

Unidade III

Nesta unidade, os conhecimentos adquiridos anteriormente serão aplicados em projetos práticos. Serão criados um aplicativo e um jogo, abordando diversos aspectos do desenvolvimento para Android. O aplicativo servirá para aplicar conceitos de interfaces, atividades e intents, enquanto o jogo permitirá explorar gráficos e animações. Por fim, o aplicativo será publicado na Google Play Store, preparando o aluno para compartilhar seus projetos com o mundo.

7 PROJETOS

Desenvolveremos dois projetos: o primeiro será um aplicativo cujo motor principal é o uso de inteligência artificial, já o segundo será um jogo tradicional, o clássico jogo da cobrinha, que foi um dos primeiros populares na época do lançamento dos celulares.

Ambos precisam ser simples, pois são os primeiros funcionais em um mundo complexo. Todavia, devem ser funcionais a fim de oferecer espaço para um desenvolvimento maior por parte do aluno após o término da disciplina.

7.1 Projeto de um app

Para estudar o processo de criação de um aplicativo, vamos desenvolver um aplicativo Android que, utilizando a API Gemini, fornecerá sugestões de roteiros turísticos personalizados para o usuário. O aplicativo solicitará ao usuário o país, a cidade e o período da viagem; e então, com base nessas informações, a API Gemini entregará dados relevantes para criar um roteiro personalizado. A interface do usuário será desenvolvida utilizando Kotlin e Android Studio.

A utilização de inteligência artificial (IA) como ferramenta de API em aplicativos está se tornando uma tendência crescente. As APIs (Interface de Programação de Aplicações) de IA permitem que os aplicativos ofereçam experiências altamente personalizadas aos usuários, como recomendações de produtos e sugestões de conteúdo adaptadas às preferências individuais. Além disso, ela está sendo usada para automatizar tarefas repetitivas e complexas, desde a análise de dados até a execução de ações específicas com base em comandos de texto ou voz.

Adicionalmente, estão surgindo marketplaces especializados em soluções de IA. Eles são plataformas online nas quais desenvolvedores podem acessar e adquirir ferramentas de IA personalizadas. Assistentes e agentes de IA estão se tornando comuns tanto no ambiente de trabalho quanto no dia a dia, ajudando a gerenciar tarefas, fornecer informações e realizar ações específicas, tornando a experiência do usuário mais fluida e eficiente. No nosso caso, utilizaremos a API Gemini fornecida pelo Google.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

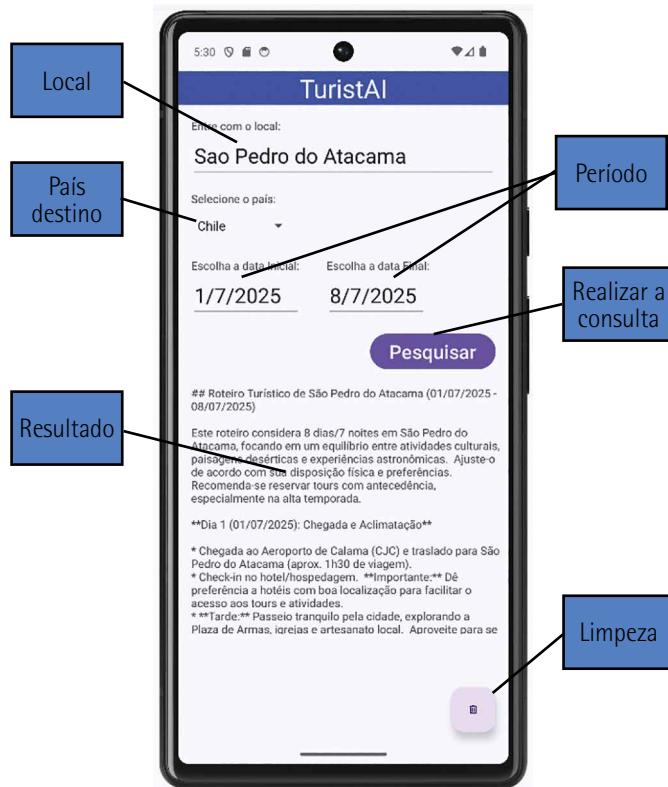


Figura 261 – A activity do aplicativo TuristAI

O princípio deste aplicativo é simples, consiste em automatizar o processo de consulta de um roteiro turístico utilizando a inteligência artificial, no caso o Gemini do Google. O nosso aplicativo irá submeter uma pergunta padrão:

```
frase="Favor elaborar um roteiro turístico para o local $local, no país $pais entre $dataini e $datafim"
```

Vamos realizar manualmente o processo que nosso aplicativo automatizará. Acessaremos o site do Gemini e faremos a pergunta:

```
frase="Favor elaborar um roteiro turístico para o local São Pedro do Atacama, no país Chile entre 1/7/2025 e 8/7/2025"
```

Unidade III

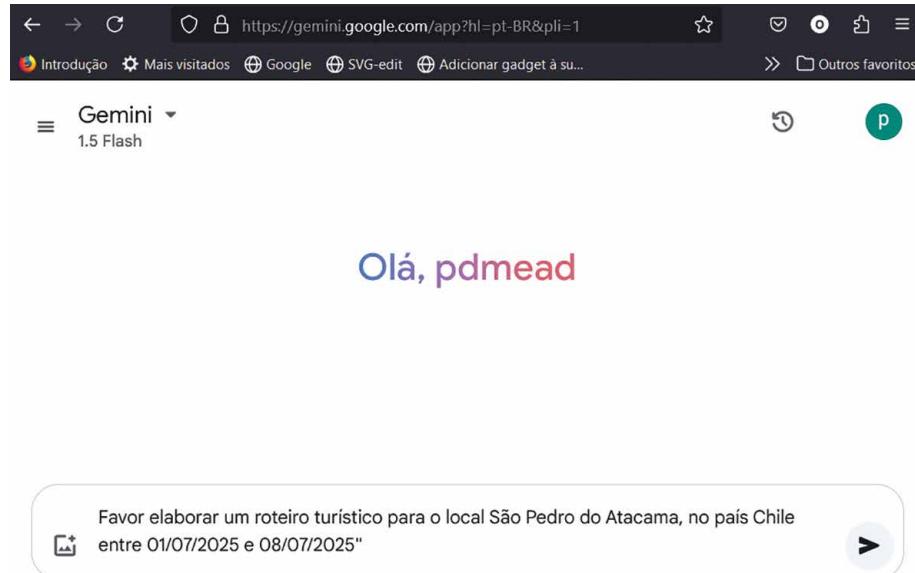


Figura 262

A pergunta é enviada ao servidor do Gemini e, após o processamento, o resultado é apresentado:

The screenshot shows the same Gemini interface after processing the query. The main content area now displays the title "Roteiro Turístico para São Pedro do Atacama: 01/07/2025 a 08/07/2025". Below this, a paragraph describes the destination: "Prepare-se para uma jornada inesquecível pelo Deserto do Atacama! Este roteiro te levará pelas paisagens mais espetaculares do Chile, com paisagens lunares, gêiseres borbulhantes, lagunas cristalinas e um céu estrelado que te deixará sem palavras.". Under the heading "Dia 1 (01/07):", there is a bulleted list: "Chegada em San Pedro de Atacama: Chegue ao aeroporto de Calama e siga para San Pedro de Atacama, a porta de entrada para o deserto." and "Acomodação: Check-in no seu hotel e aproveite para se adaptar à altitude.". At the bottom of the screen, a text input field says "Peça ao Gemini" and a microphone icon is visible.

Figura 263

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Utilizamos uma API para realizar as consultas de forma programática, eliminando a necessidade de interação do usuário com o navegador. No nosso aplicativo, o resultado da consulta é apresentado na região dedicada.

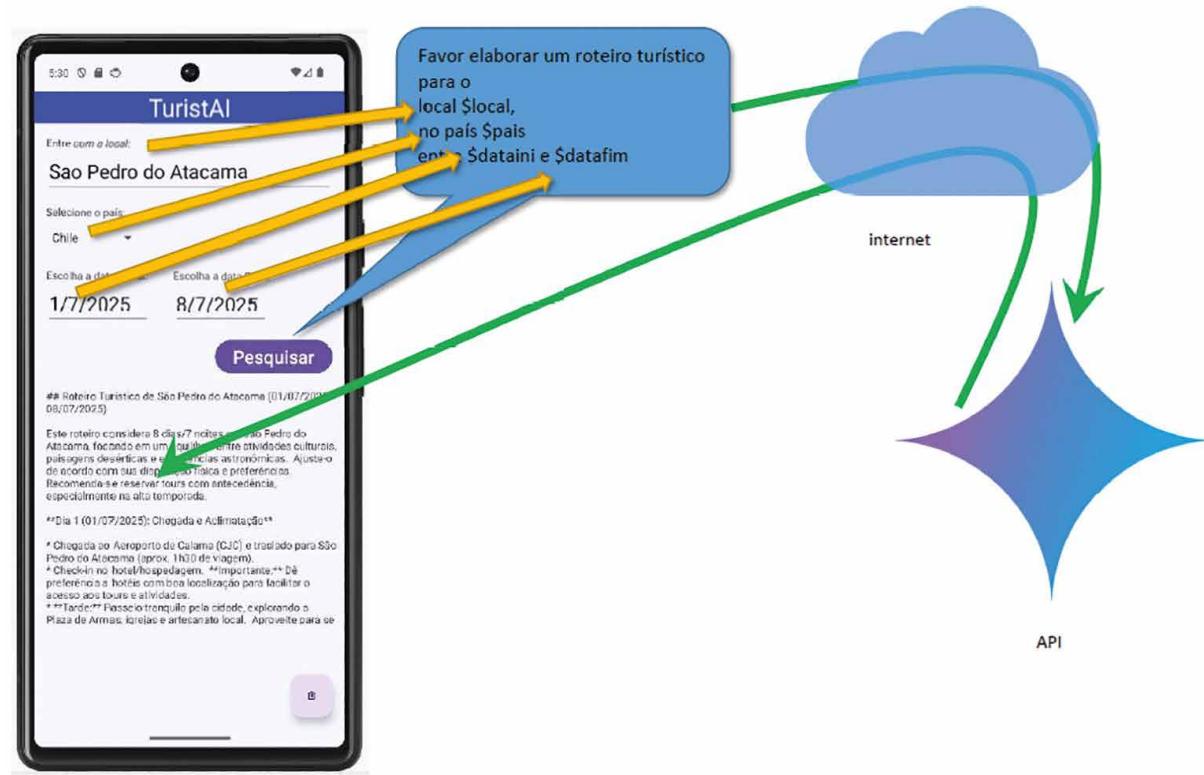


Figura 264 – Fluxo de informações entre o aplicativo e o Gemini



Uma API é um conjunto de definições e protocolos que permite que diferentes softwares se comuniquem entre si. Em termos simples, ela define as maneiras pelas quais um desenvolvedor pode interagir com um serviço ou sistema, facilitando a integração e a troca de dados entre diferentes aplicações.

Por exemplo, quando você faz um cadastro no seu smartphone, ele pode usar uma API para obter dados da sua residência, bastando digitar o seu CEP. As APIs são essenciais para a construção de software moderno, pois permitem que desenvolvedores utilizem funcionalidades de outros serviços sem precisar entender ou acessar o seu código-fonte.

7.1.1 Criar o projeto na plataforma Android

Neste primeiro projeto, utilizaremos a técnica do Views, pois a tela será estática, com campos em localizações fixas. Isso permitirá uma interface de usuário simples e direta, em que os elementos não mudam de posição, garantindo boa usabilidade para o usuário.

Conforme visto em 6.1 Iniciando o projeto com Views, vamos criar um projeto:

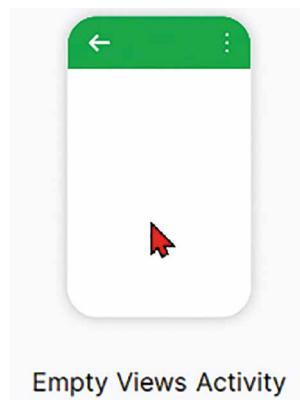


Figura 265

Em Configuração Inicial criaremos um projeto, definiremos o nome do aplicativo, no caso 'App TuristAI' por exemplo, o pacote, aquele sugerido pelo Android Studio (com.example.appturistai), a linguagem de programação (Kotlin) e o mínimo nível de API que seu aplicativo suportará.

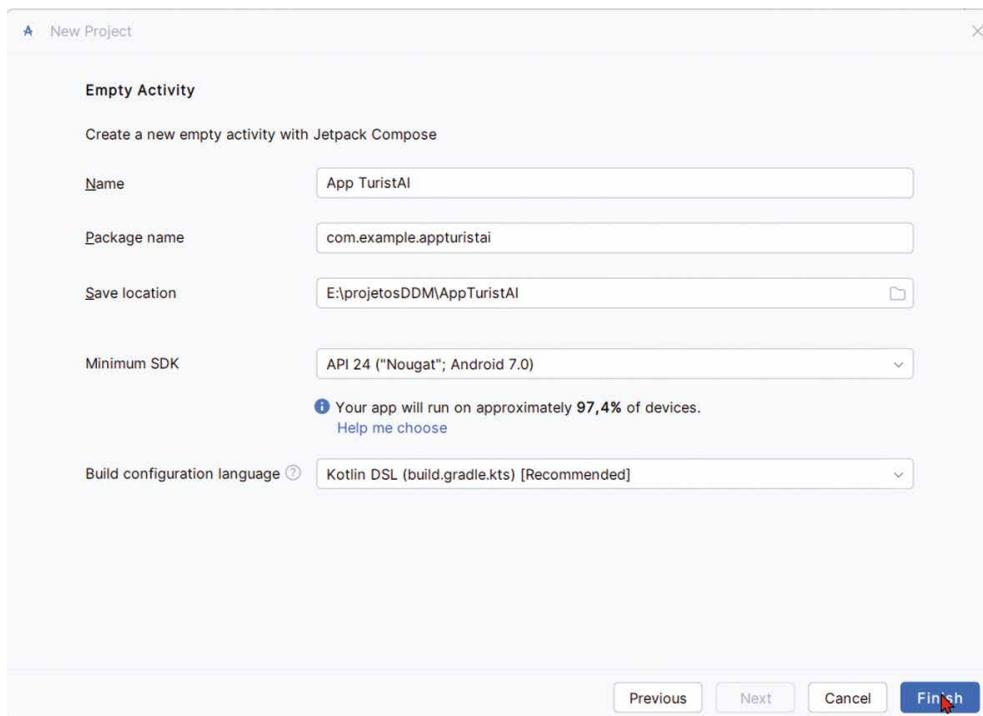


Figura 266



Uma prática interessante é salvar o projeto no GitHub. Trata-se de uma plataforma online que serve como um repositório para armazenar e gerenciar código-fonte de projetos de software. Ele utiliza o sistema de controle de versão Git para rastrear todas as alterações feitas no código, permitindo que desenvolvedores colaborem em projetos, revisem o histórico de alterações e trabalhem em diferentes versões do código simultaneamente. Além disso, oferece ferramentas para gerenciar tarefas, discutir ideias e criar comunidades em torno de projetos de software. Ele é como um Google Drive ou OneDrive para programadores, mas com foco em código e colaboração.

O Android Studio oferece suporte integrado ao GitHub, facilitando o gerenciamento de versões e a colaboração em projetos de desenvolvimento de aplicativos Android. Com ele, você pode clonar repositórios do GitHub, criar repositórios, fazer commits, push e pull de alterações diretamente da interface do IDE. Além disso, permite a integração com outras ferramentas de controle de versão.

Inicialmente, siga estes passos simples para criar sua conta:

- **Acesse o site do GitHub:** abra seu navegador e digite <https://github.com/> na barra de endereço.
- **Clique em Sign up:** na página inicial, você encontrará um botão proeminente que diz "Sign up" (Inscrever-se). Clique nele para iniciar o processo de criação da conta.
- Preencha as informações.
 - **Endereço de e-mail:** insira seu endereço de e-mail válido.
 - **Nome de usuário:** escolha um nome de usuário único que irá identificá-lo na plataforma. Escolha um que seja fácil de lembrar e o represente ou seus projetos.
 - **Senha:** crie uma senha forte e segura, certifique-se de que ela tenha pelo menos 15 caracteres ou, se tiver no mínimo um número e uma letra minúscula, oito caracteres.
 - **Informações adicionais:** você pode optar por fornecer informações adicionais, como seu nome completo e organização (opcional).

- **Verifique sua conta:** o GitHub enviará um e-mail de verificação para o endereço fornecido. Clique no link de verificação no e-mail para ativar sua conta.
- **Personalize seu perfil:** após a verificação, você será direcionado para a sua página de perfil. Personalize-a adicionando uma foto, uma breve descrição e links para seus outros projetos.

O Android Studio oferece integração nativa com o GitHub, facilitando o gerenciamento de seus projetos. Confira o passo a passo para começar a utilizar a ferramenta:

- Configurar o Git no Android Studio:
 - **Habilitar o controle de versão:** vá em VCS > Enable Version Control Integration.
 - **Selecionar o Git:** na janela que se abrir, escolha Git como o sistema de controle de versão.
- Conectar ao GitHub:
 - **Criar um repositório:** se você ainda não tem um repositório no GitHub para seu projeto, crie um por meio da interface web do GitHub.
 - **Conectar o projeto ao repositório:** no Android Studio, vá em VCS > Import into Version Control > Share Project on GitHub.
 - Preencher as informações:
 - **URL do repositório:** cole a URL do repositório criado no GitHub.
 - **Nome do diretório:** especifique o diretório local onde seu projeto está salvo.
 - **Outras opções:** configure outras opções, como nome do ramo remoto e se o repositório é privado ou público.
- Realizar o primeiro commit:
 - **Adicionar arquivos:** selecione todos os arquivos do projeto que você deseja adicionar ao controle de versão.
 - **Commit:** clique com o botão direito nos arquivos selecionados e escolha Git > Add. Em seguida, vá em VCS > Commit.

- **Escrever uma mensagem de commit:** descreva brevemente as alterações que você fez.
- **Commit:** clique em Commit.
- Enviar as alterações para o GitHub:
 - **Push:** vá em VCS > Git > Push.
 - **Selecionar o ramo:** escolha o ramo local que você deseja enviar para o GitHub.
 - **Confirmar:** clique em Push.
- Comandos Git úteis no Android Studio:
 - **Pull:** baixar as últimas alterações do repositório remoto.
 - **Fetch:** baixar informações sobre os ramos remotos sem mesclá-los.
 - **Merge:** mesclar alterações de um ramo em outro.
 - **Branch:** criar um ramo.
 - **Checkout:** mudar para um ramo diferente.

Parece longo, mas no dia a dia torna-se quase automático o processo de salvamento.

Documentação oficial do GitHub: <https://shre.ink/MhpX>

7.1.2 Registrar o projeto

Abordaremos as etapas para a publicação do seu aplicativo no capítulo 8, Publicação de Aplicativos e Jogos. Por enquanto, é importante considerar algumas informações essenciais caso você planeje publicar no Google Play. Elas nem sempre são necessárias no início do projeto, mas é fundamental localizá-las em seu projeto.

As principais informações estão no arquivo `AndroidManifest.xml`. O `AndroidManifest.xml` é um arquivo XML essencial em todos os aplicativos Android, localizado na raiz do conjunto de fontes do projeto (figura 267). Ele serve como uma espécie de carteira de identidade do app, fornecendo informações cruciais ao sistema operacional. Nele, são declarados os componentes do app (atividades, serviços, broadcast receivers e content providers), as permissões necessárias para o funcionamento, os filtros de intent, o sistema operacional Android e a Google Play Store, bem como outras configurações

Unidade III

importantes. Trata-se de um mapa que guia o Android a entender a estrutura e as necessidades do seu aplicativo, garantindo que ele seja executado corretamente e de forma segura.

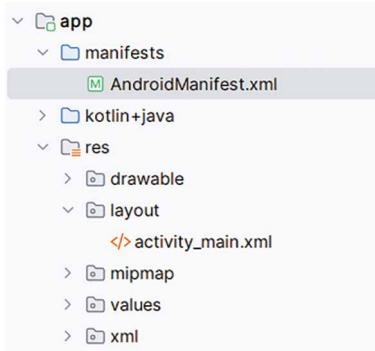


Figura 267 – Localização do AndroidManifest.xml

O arquivo inicial está assim:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="App TuristAI"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AppTuristAI"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Figura 268

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

No momento, vamos nos ater a três linhas:



Figura 269

A linha android:label indica como será o rótulo do nosso aplicativo que aparecerá na tela do dispositivo. Para alterá-lo, clique em label e verifique onde está a String com o nome ou se está digitada direto.

O exemplo indica que o rótulo está em @string/app_name.



Figura 270

Portanto, significa que está na pasta:

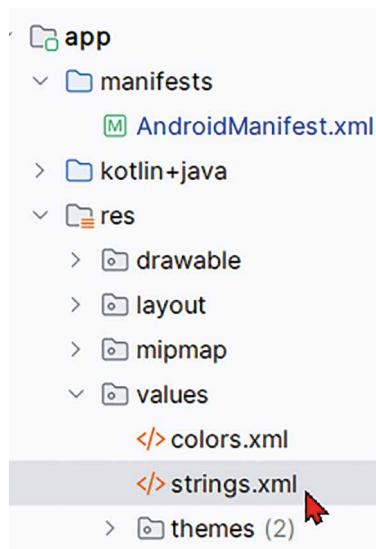


Figura 271

Verificando o conteúdo do xml, temos:



Figura 272

Unidade III

Vamos alterar para:

```
<resources>
    <string name="app_name">TuristAI</string>
</resources>
```

Figura 273

Voltando às três chaves do AndroidManifest.xml, as outras duas são os ícones que representarão o nosso aplicativo no sistema operacional, também conhecido como ícone de launcher (figura 274). O primeiro para aqueles que exibem o ícone redondo e o outro para os quadrados.

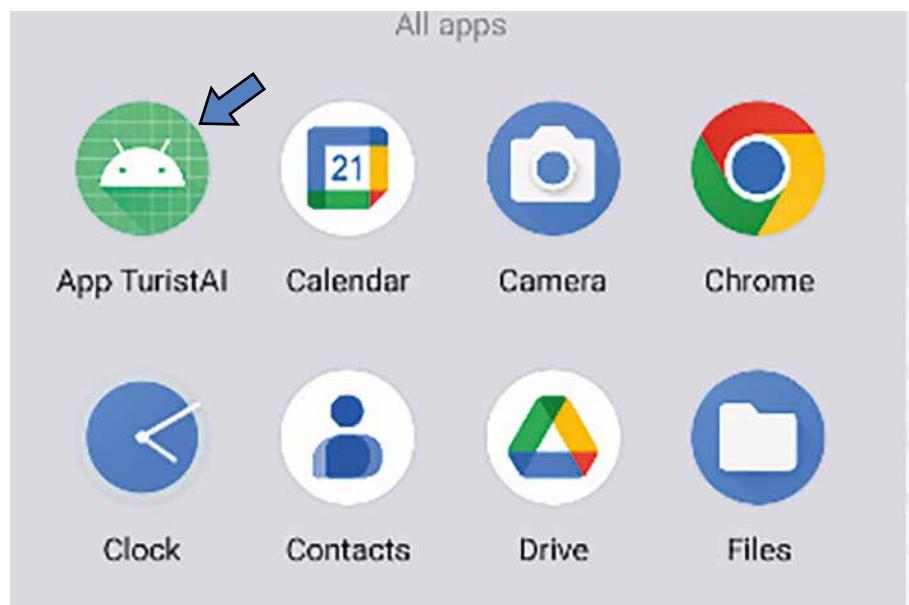


Figura 274 – Ícone do aplicativo

Se quisermos alterar para:

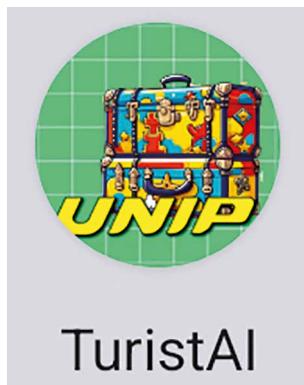


Figura 275

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Para criar um ícone personalizado no Android Studio, siga estes passos:

- Prepare seus arquivos de imagem:
 - **Formatos:** os formatos mais comuns são PNG e SVG. O SVG é recomendado para ícones vetoriais, pois se adapta a diferentes resoluções sem perda de qualidade.
 - **Tamanhos:** crie versões do seu ícone para diferentes densidades de tela (mdpi, hdpi, xhdpi, xxhdpi, xxxhdpi). Isso garante que ele se adapte a diversos dispositivos.
 - **Máscara:** certifique-se de que seu ícone tenha uma máscara transparente para que se adapte ao formato circular de alguns dispositivos.

Consta na sequência imagem criada para o projeto:

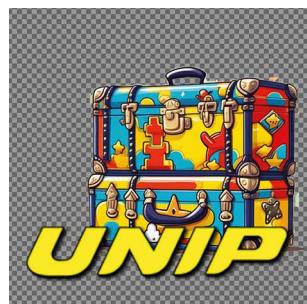


Figura 276

- Abra o Image Asset Studio:
 - No Android Studio, abra o seu projeto.

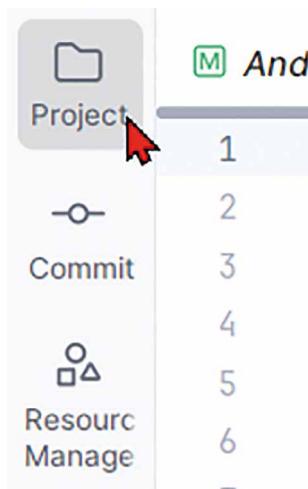


Figura 277

Unidade III

- Vá para Res > mipmap (a pasta onde estão seus ícones).
- Clique com o botão direito e selecione New > Image Asset.

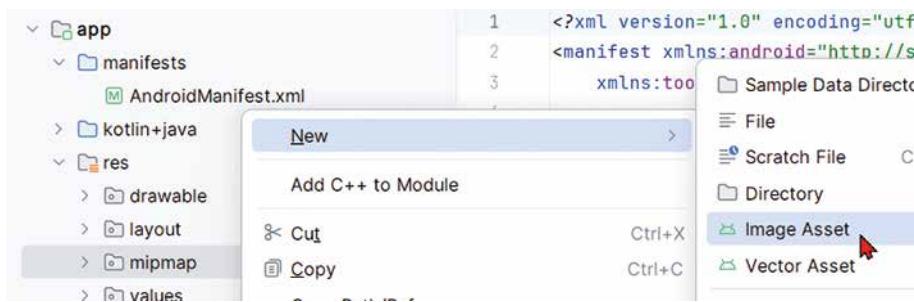


Figura 278

- Configure o Image Asset Studio:

- **Fonte do ícone:** escolha se você quer importar um arquivo de imagem existente ou usar um ícone do Material Design.
- **Tipo de ícone:** selecione Launcher Icons (Ícones de Launcher).
- Configurações:
 - **Nome do recurso:** defina o nome do recurso do seu ícone (por exemplo, ic_launcher).
 - **Diretório:** selecione a pasta mipmap onde os ícones serão salvos.
 - **Formato:** escolha o formato desejado (PNG ou SVG).
 - **Densidades:** selecione as densidades para as quais você quer gerar os ícones.
 - **Configurar ícone:** carregue seu arquivo de imagem ou selecione um ícone do Material Design.

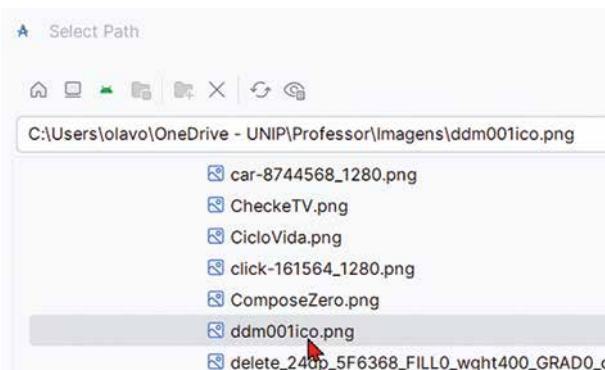


Figura 279

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- **Foreground Layer:** configure a camada frontal do ícone (opcional).
- **Background Layer:** configure a camada de fundo do ícone (opcional).

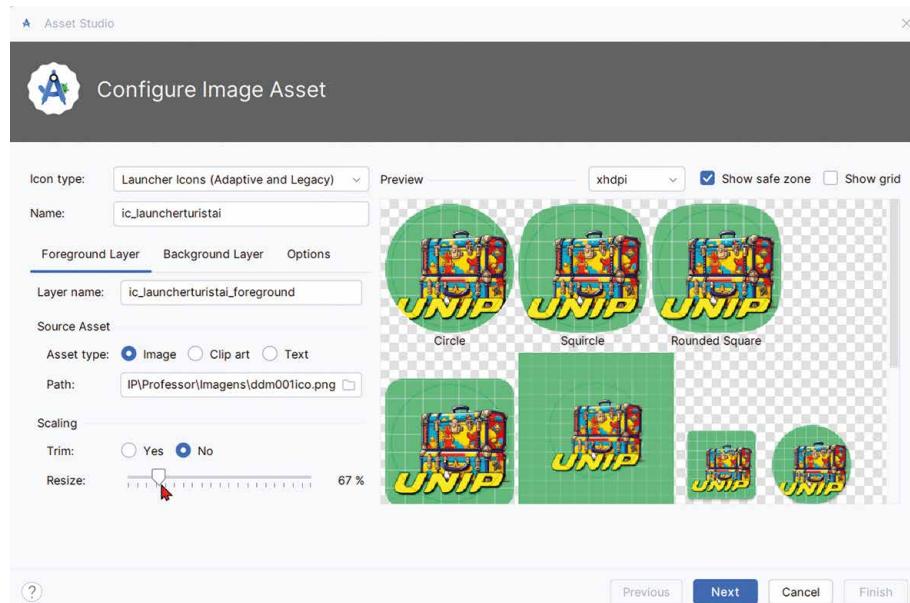


Figura 280

No nosso caso, somente alteramos Name, Path e escala.

- Gere os ícones:
 - Clique em Finish para gerar os ícones nas diferentes densidades.
- Verifique os resultados:
 - Na pasta mipmap, você encontrará os ícones gerados.

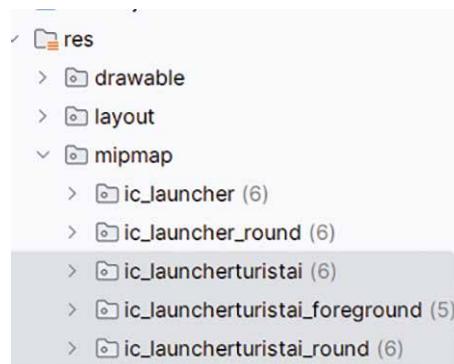


Figura 281

Verifique se os ícones estão com a qualidade desejada e estão sendo renderizados corretamente no emulador ou em um dispositivo físico.

- Altere o AndroidManifest.xml:

```
    android:icon="@mipmap/ic_laucherturistai"  
    android:label="@string/app_name"  
    android:roundIcon="@mipmap/ic_laucherturistai_round"
```

Figura 282

Verificando no emulador a tela de carregamento:



Figura 283

O ícone na área de trabalho:

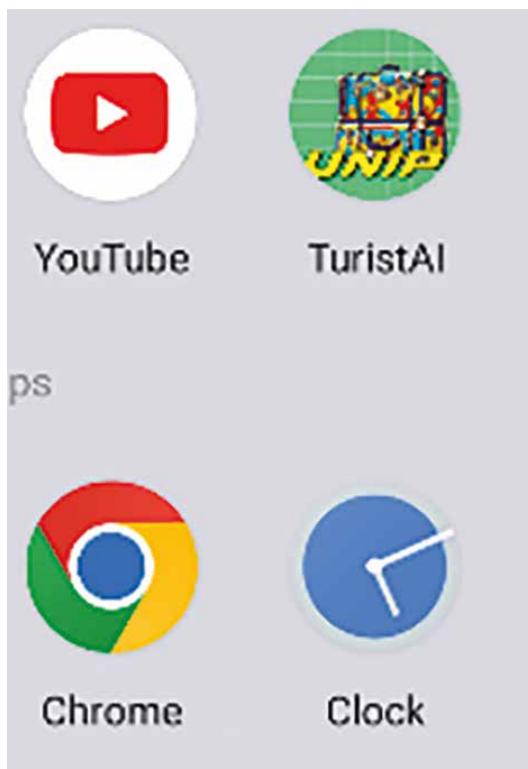


Figura 284

Desta forma, o nosso aplicativo já tem identidade própria.

7.1.3 Implementar as activities

O nosso aplicativo possui apenas uma activity, ela é organizada a fim de facilitar a navegação do usuário para o planejamento de sua viagem. O usuário informa o local e país de destino, escolhe as datas da viagem e inicia a pesquisa por roteiros turísticos. A área de resultados deve exibir o roteiro sugerido e há um botão para limpar os dados inseridos.

Vamos analisar os elementos da tela detalhadamente (figura 285).

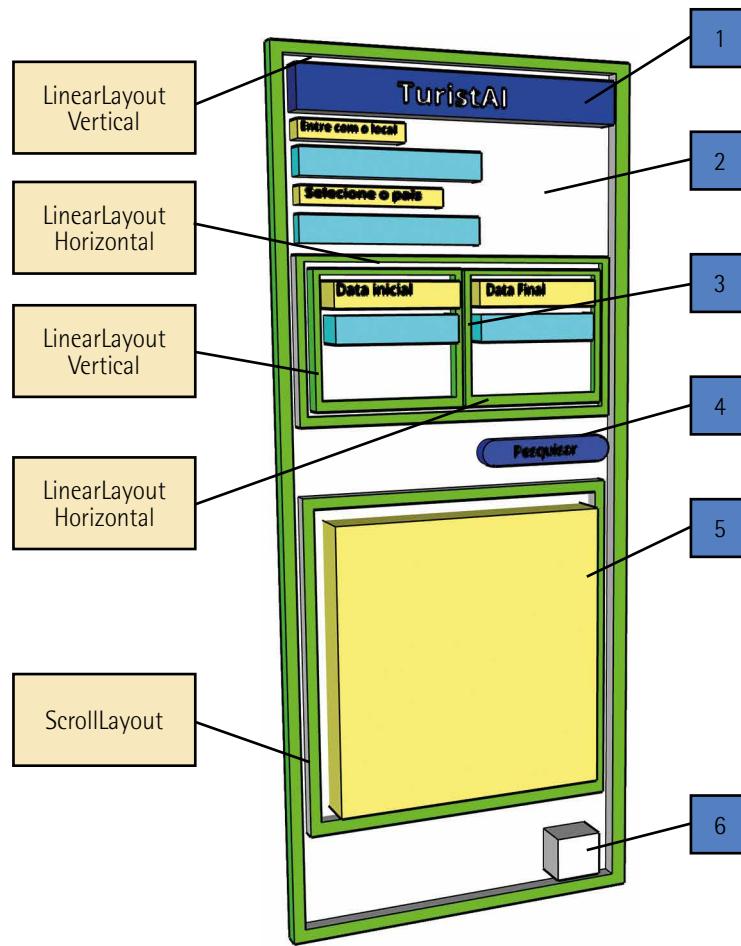


Figura 285

- Cabeçalho:
 - Um banner azul na parte superior com o logo do aplicativo "TuristAI" centralizado e escrito em branco.
- Local e país:
 - Um campo de texto onde o usuário pode digitar o local desejado para a viagem.
 - Um texto explicativo "Entre com o local:"
 - Um spinner (caixa de seleção) onde o usuário pode escolher o país de destino.
 - Um texto explicativo "Selezione o país:"

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- Datas da viagem:
 - Dois campos de texto onde o usuário pode selecionar as datas de início e final da viagem.
 - Os campos não permitem digitação direta, mas abrem um calendário para escolha da data.
 - Um texto explicativo "Escolha a Data Inicial:" para o primeiro campo.
 - Um texto explicativo "Escolha a Data Final:" para o segundo campo.
- Botão de pesquisa:
 - Um botão azul com o texto "Pesquisar" localizado no canto direito da tela.
- Área de resultados:
 - Uma caixa de rolagem vertical que deve exibir o roteiro sugerido pelo aplicativo após a pesquisa.
 - Inicialmente, essa área exibe apenas o texto "Seu roteiro".
- Botão flutuante de ação:
 - Um botão circular flutuante na parte inferior direita da tela com uma lixeira como ícone.
 - A função desse botão serve para limpar os dados inseridos pelo usuário.

O layout da activity principal do aplicativo TuristAI é composto por um LinearLayout principal com orientação vertical, que organiza os demais elementos da tela em uma única coluna.

```
<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:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    .

    .

    .

</LinearLayout>
```

No novo projeto, o Layout deve ser convertido para o novo Layout.



Lembrete

Para converter os tipos de layout, deixe a tela na visão Split.

Ative o Component Tree:

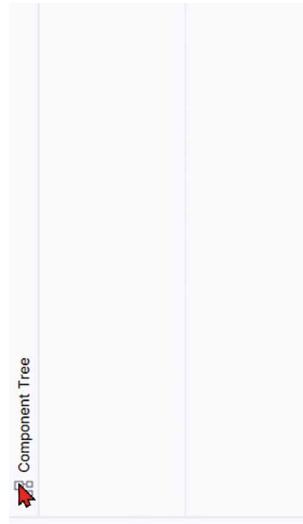


Figura 286

Clique com a direita em main e selecione Convert View:

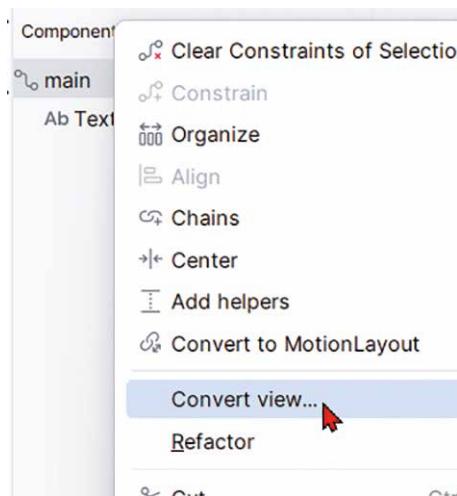


Figura 287

Escolha LinearLayout e clique no botão Apply.

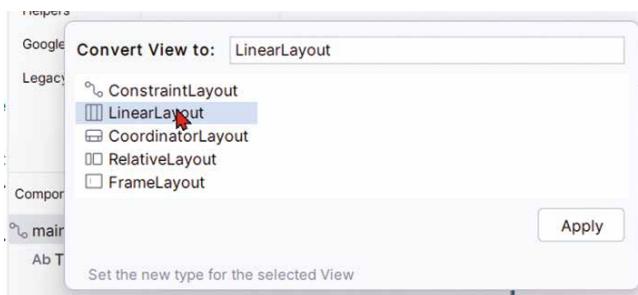


Figura 288

Acrescente o atributo orientation e defina vertical.

```
    android:orientation="vertical"
```

Figura 289

- Cabeçalho:

- Um **TextView** na parte superior preenche a largura da tela (match_parent) e exibe o logo do aplicativo TuristAI centralizado.
- O fundo do TextView é azul (#3F51B5) e o texto é branco (@color/white).

Para montar o cabeçalho, o texto do Hello World pode ser alterado.

```
<TextView  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:background="#3F51B5"  
    android:text="TuristAI"  
    android:textAlignment="center"  
    android:textColor="@color/white"  
    android:textSize="32dp" />
```



Figura 290

- Local e país:
 - Um **TextView** com o texto "Entre com o local:" indica o campo de entrada seguinte.

```
<TextView  
    android:id="@+id/textView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="16dp"  
    android:layout_marginTop="16dp"  
    android:text="Entre com o local:" />
```



Figura 291

No **EditText** (campo de entrada de texto), o campo tem o ID @+id/etLocal, largura preenchendo o pai ('fill_parent') e altura ajustável ao conteúdo ('wrap_content'). Ele possui margens laterais de 16dp e uma tonalidade de fundo neutra. A dica exibida no campo é "Local" e a frequência de hifenização é normal. O tipo de entrada é texto e o tamanho do texto é 26dp. Este campo é utilizado para que o usuário insira o nome de um local.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
<EditText  
    android:id="@+id/etLocal"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="16dp"  
    android:layout_marginEnd="16dp"  
    android:backgroundTint="@color/material_dynamic_neutral80"  
    android:hint="Local"  
    android:hyphenationFrequency="normal"  
    android:inputType="text"  
    android:textSize="26dp" />
```



Figura 292

Um **TextView** com o texto "Selecione o país:" indica o spinner a seguir.

```
<TextView  
    android:id="@+id/textView4"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="16dp"  
    android:layout_marginTop="16dp"  
    android:text="Selecione o país:" />
```



Figura 293

Coloca-se um **Spinner** (caixa de seleção) com o ID `@+id/spinner`, largura ajustável ao conteúdo ('wrap_content') e altura fixa de 48dp. Ele possui margens laterais de 16dp. O Spinner é usado para apresentar uma lista suspensa de opções que o usuário pode selecionar, funcionando de maneira semelhante a um menu dropdown. As margens laterais garantem que o componente esteja devidamente espaçado em relação aos outros elementos do layout.

```
<Spinner  
    android:id="@+id/spinner"  
    android:layout_width="wrap_content"  
    android:layout_height="48dp"  
    android:layout_marginStart="16dp"  
    android:layout_marginEnd="16dp" />
```



Figura 294

Datas da viagem

Este grupo de elementos possui um posicionamento mais sofisticado. O texto da data inicial está posicionado acima do campo de entrada do calendário, enquanto as datas finais da viagem estão dispostas ao lado do conjunto inicial. Desta forma, os layouts ficam:

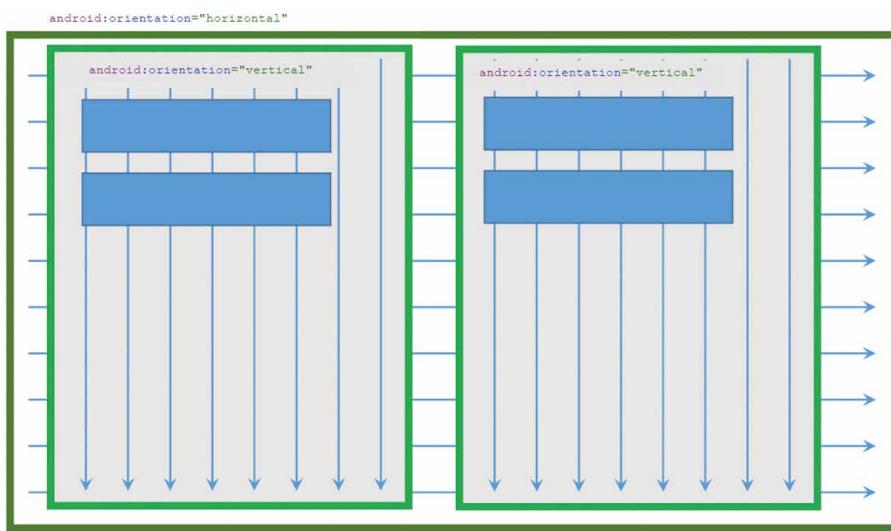


Figura 295

```
<LinearLayout  
    android:orientation="horizontal">  
    <LinearLayout  
        android:orientation="vertical">  
            <TextView  
                android:text="Escolha a data Inicial:" />  
            <EditText
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
        android:hint="Data Inicial" />
    </LinearLayout>

    <LinearLayout
        android:orientation="vertical">
        <TextView
            android:text="Escolha a data Final:" />
        <EditText
            android:hint="Data Final"
        />
    </LinearLayout>
</LinearLayout>
```

Um LinearLayout horizontal organiza os elementos referentes às datas da viagem.

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

Um primeiro LinearLayout vertical agrupa:

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

Um **TextView** com o texto "Escolha a data Inicial:".

```
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:text="Escolha a data Inicial:" />
```



Figura 296

Um **EditText** é onde o usuário seleciona a data de início da viagem. Ele possui propriedades que impedem a digitação direta (android:clickable="false", android:cursorVisible="false", android:focusable="false", android:focusableInTouchMode="false"). Ao clicar neste campo, será aberto um calendário para escolha da data.

```
<EditText  
    android:id="@+id/etDataIni"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="16dp"  
    android:layout_marginEnd="16dp"  
    android:clickable="false"  
    android:cursorVisible="false"  
    android:focusable="false"  
    android:focusableInTouchMode="false"  
    android:hint="Data Inicial"  
    android:inputType="datetime|date"  
    android:textSize="26sp" />
```



Figura 297

O LinearLayout Vertical é encerrado.

```
</LinearLayout>
```

Temos agora um segundo **LinearLayout** ao lado do primeiro, vertical similar ao primeiro, agrupando:

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

Um **TextView** com o texto "Escolha a data Final:"

```
<TextView  
    android:id="@+id/textView3"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:text="Escolha a data Final:" />
```



Figura 298

Um **EditText** similar ao anterior, em que o usuário seleciona a data final da viagem.

```
<EditText
    android:id="@+id/etDataFim"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginEnd="16dp"
    android:clickable="false"
    android:cursorVisible="false"
    android:focusable="false"
    android:focusableInTouchMode="false"
    android:hint="Data Final"
    android:inputType="datetime|date"
    android:textSize="26sp" />
```



Figura 299

Aqui os dois LinearLayouts são fechados.

```
</LinearLayout>
</LinearLayout>
```

Unidade III

- Botão de pesquisa:

- Um **Button** (botão) com o texto "Pesquisar" posicionado na parte direita da tela (android:layout_gravity="right").

```
<Button  
    android:id="@+id/btOK"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="right"  
    android:layout_marginTop="16dp"  
    android:layout_marginEnd="16dp"  
    android:text="Pesquisar"  
    android:textSize="24sp" />
```



Figura 300

Área de resultados

Como o texto resultante geralmente é maior que o tamanho de uma caixa de texto, é necessário colocá-lo em um layout que permita deslizar o conteúdo para cima ou para baixo, possibilitando a visualização de partes que não estão inicialmente visíveis, como no caso do ScrollView.

Um **ScrollView** (área com scroll vertical) preenche a maior parte da tela (android:layout_weight="0.9").

```
<ScrollView  
    android:layout_width="match_parent"  
    android:layout_height="0dp"  
    android:layout_weight="0.9"  
    android:layout_marginStart="16dp"  
    android:layout_marginTop="16dp"  
    android:layout_marginBottom="50dp">  
  
</ScrollView>
```

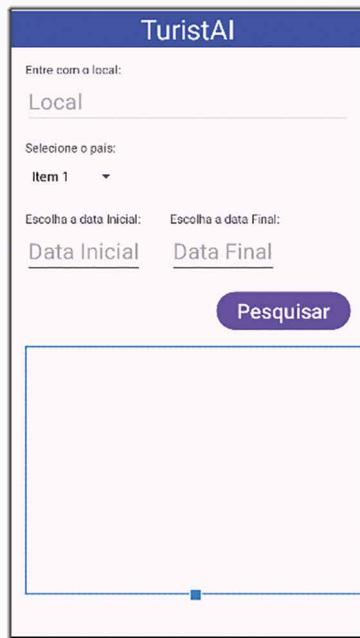


Figura 301

No ScrollView, um **TextView** exibe o texto "Seu roteiro" como a dica inicial. Essa área deve exibir o roteiro turístico sugerido pelo aplicativo após a pesquisa.

```
<TextView  
    android:id="@+id/txtRetorno"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:text="Seu roteiro" />
```

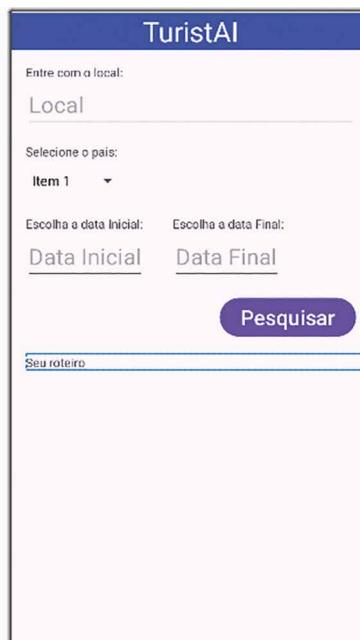


Figura 302

Unidade III

- Botão flutuante de ação (FAB):

- A imagem do botão representa uma lixeira (`android:src="@drawable/ic_lixo"`), indicando a função provável de limpar os dados inseridos pelo usuário.

Para a limpeza dos campos, é necessário criar um ícone de lata de lixo a ser colocado no FAB (Floating Action Button). Já vimos como criar um ícone em 7.1.2, vamos repetir o processo para criar o ícone `ic_lixo`.



Figura 303

Na pasta res do projeto, clique com o botão da direita e escolha New/Image Asset.

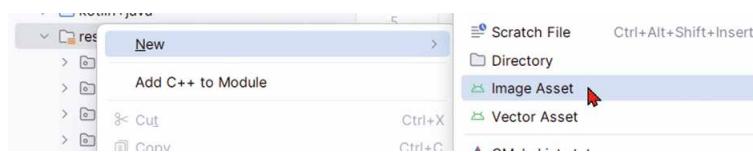


Figura 304

Renomeamos e inserimos a nova imagem em path.

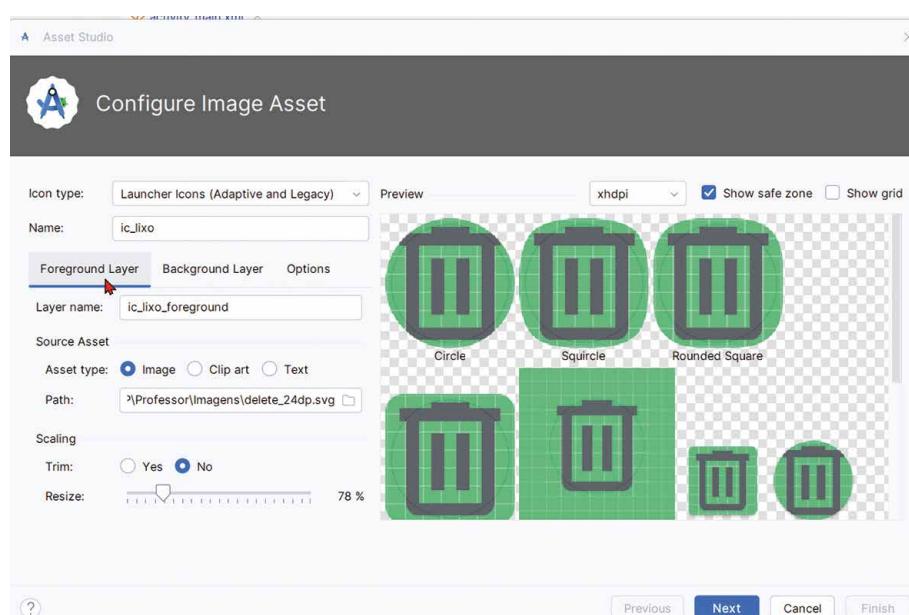


Figura 305

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Na aba Background Layer, diminua o resize para 0%, desta forma o fundo será transparente.

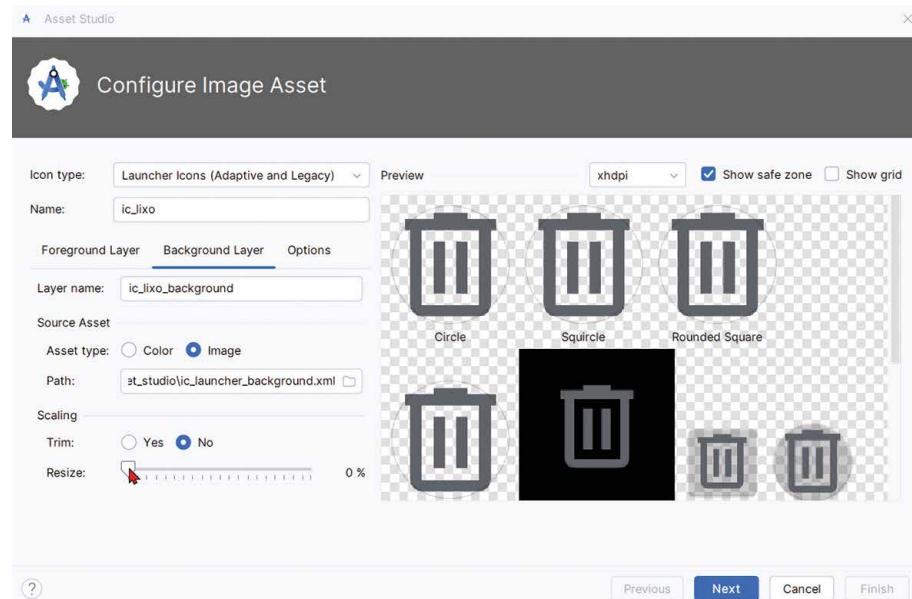


Figura 306

Clique em next e finalize.

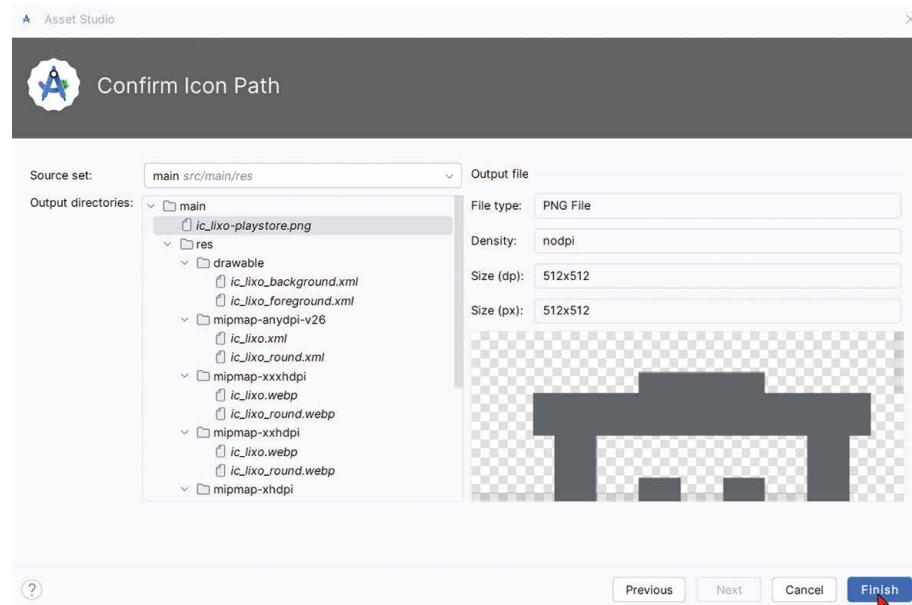


Figura 307

Após a feitura do ícone, vamos à criação do botão flutuante.

*Um **FloatingActionButton** (botão flutuante de ação) posicionado na parte inferior direita da tela (android:layout_gravity="bottom|right").

Unidade III

Ao digitar, escolha a partir da sugestão apresentada pela IDE.

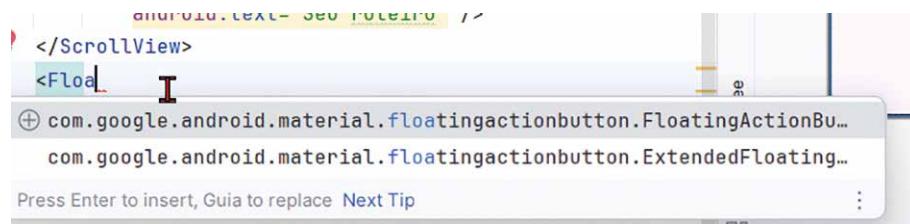


Figura 308

Podemos atestar que a motivação por trás da utilização de com.google.android.material.floatingactionbutton.FloatingActionButton em vez de apenas FloatingActionButton é garantir que você está usando a versão específica do componente fornecida pela biblioteca Material Components for Android.

```
<com.google.android.material.floatingactionbutton.FloatingActionButton  
    android:id="@+id/fabLixo"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="bottom|right"  
    android:layout_margin="16dp"  
    android:contentDescription="Limpar"  
    android:src="@drawable/ic_lixo_foreground" />
```

A tela final será:

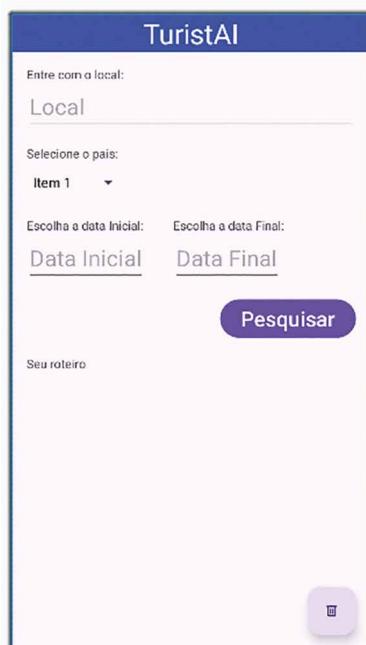


Figura 309

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

E a sua estrutura e o emulador:

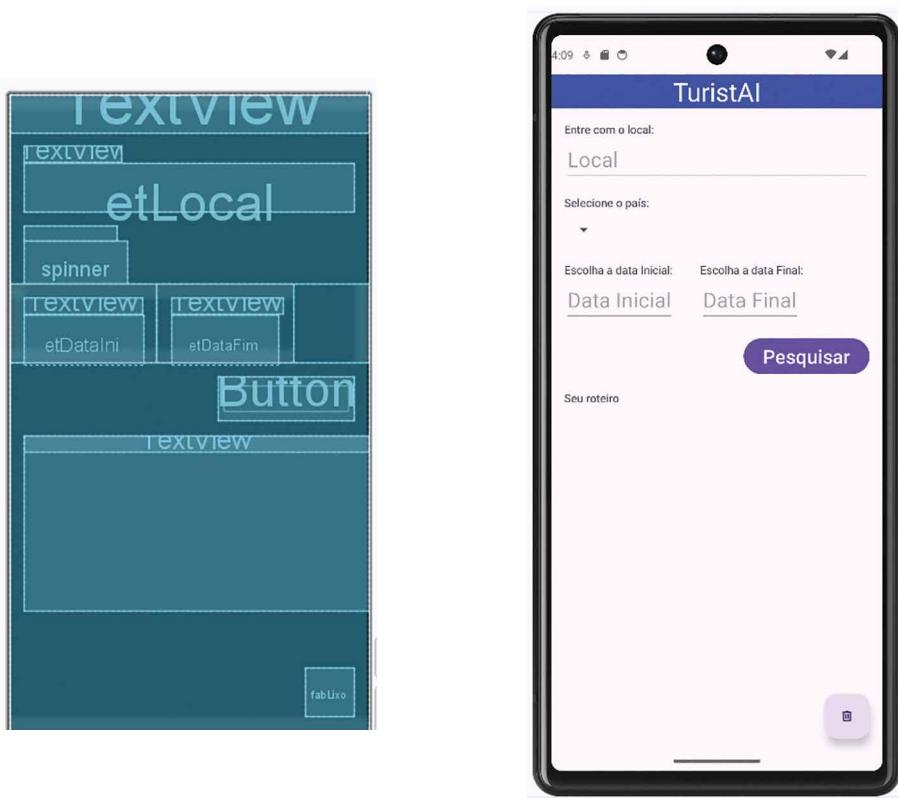


Figura 310

Com o Layout montado, vamos passar para a fase de programação do nosso aplicativo. Prepararemos o ambiente da edição do MainActivity. Caso a aba do programa não esteja aberta, basta abrir a janela de projetos e localizar a pasta do Kotlin e clicar em MainActivity.

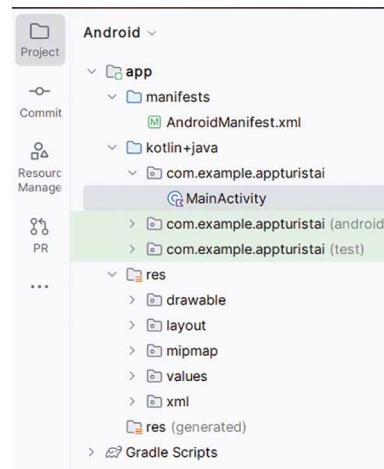


Figura 311

Inicialmente o nosso MainActivity.kt permanece intocado:

```
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 312

- Declaração da classe e método onCreate:

- **A linha class MainActivity:** AppCompatActivity() declara uma classe chamada MainActivity, que herda de AppCompatActivity. Isso indica que MainActivity é uma atividade que pode ser exibida na tela do dispositivo Android.
- **class MainActivity:** AppCompatActivity().
- O método onCreate é chamado quando a atividade é criada pela primeira vez. Ele é responsável por inicializar a atividade e configurar seus componentes visuais.

```
override fun onCreate(savedInstanceState: Bundle?)
```

- Inicialização da Activity:

- **super.onCreate(savedInstanceState):** chama o método onCreate da classe-pai para lidar com tarefas comuns de inicialização.
- **enableEdgeToEdge():** (provavelmente) habilita um recurso que permite exibir o conteúdo da Activity de ponta a ponta na tela, sem a barra de status.
- **setContentView(R.layout.activity_main):** define o layout da Activity como activity_main.xml. Este arquivo XML estabelece a estrutura visual da Activity usando componentes de interface do usuário (UI).
- **ViewCompat.setOnApplyWindowInsetsListener:** trata das áreas da tela que ficam ocultas pelas barras do sistema (status bar e barra de navegação). Ele garante que o conteúdo da Activity tenha um espaçamento adequado para não sobrepor essas barras.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
{  
    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  
    }  
}
```

- Variáveis de trabalho:

– **frase**: uma variável do tipo String chamada frase e a inicializa com uma string vazia. Essa variável será utilizada para armazenar a frase que será enviada para a API Gemini para gerar o roteiro turístico.

```
var frase: String = "" //frase de envio ao API
```

– **país**: variável usada para armazenar o país selecionado pelo usuário em um componente da interface chamado Spinner.

```
var pais: String = "" //armazenar o país selecionado no Spinner
```

O trecho ficará:

```
//Variáveis de Trabalho  
var frase: String = "" //frase de envio ao API  
var pais: String = "" //armazenar o país selecionado no Spinner
```

- Configuração entrada do local:

– **etLocal**: representa o campo de texto onde o usuário digita o local desejado. É uma variável do tipo EditText chamada etLocal. Um EditText é um campo de texto onde o usuário pode digitar informações (EditText).

```
var etLocal: EditText //campo para digitar o local
```

findViewById(R.id.etLocal): associa a variável etLocal ao elemento da interface com o ID etLocal. Isso significa que etLocal agora representa o campo de texto que você definiu no seu layout XML com o ID etLocal.

```
etLocal = findViewById(R.id.etLocal) //localiza o campo com o id etLocal no layout
```

Configuração do botão de pesquisa

O botão de pesquisa será utilizado para testar a cada etapa da construção da atividade, portanto, os tratamentos dos componentes em seguida deverão ser posicionados antes.

- **btOk**: uma variável do tipo Button chamada btOk. Um Button é um botão que, quando clicado, executa uma ação (Button).

```
var btOk: Button
```

- **findViewById(R.id.btOk)**: associa a variável btOk ao botão com o ID btOk no seu layout XML

```
btOk = findViewById(R.id.btOk)
```

- **btOk.setOnClickListener { ... }**: associa um listener para o botão btOk, define o comportamento do botão quando ele é clicado. Quando o botão é pressionado, o código dentro das chaves é executado.

```
btOk.setOnClickListener {
```

Clicar no botão de pesquisa (btOk) aciona as seguintes ações:

- Quando o botão é clicado, o texto que o usuário digitou no campo etLocal é obtido e armazenado na variável local.
- **etLocal.text.toString()**: obtém o texto digitado no campo etLocal e o armazena na variável local.

```
var local: String = etLocal.text.toString() //armazenar o local digitado
```

- **frase**: atualiza a variável frase com a mensagem que inclui o local digitado. A variável é atualizada com uma string que inclui o conteúdo "Favor elaborar um roteiro turístico para o local" seguido do valor armazenado em local. Essa string mais tarde será enviada para a API a fim de gerar o roteiro.

```
. frase = "Favor elaborar um roteiro turístico para o local $local"
```

- **Toast.makeText(this, frase, Toast.LENGTH_LONG).show()**: exibe uma mensagem curta na tela (Toast) com o conteúdo da variável frase. Isso serve como confirmação para o usuário de que a solicitação foi enviada.

```
Toast.makeText(this, frase, Toast.LENGTH_LONG).show()
```

Este trecho de código configura a parte da interface do usuário responsável por coletar informações do usuário (local e país) e, quando o botão Pesquisar é clicado, monta uma frase que será enviada para uma API a fim de gerar um roteiro turístico personalizado.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
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
        }
        //Variáveis de Trabalho
        var frase: String = "" //frase de envio ao API
        var pais: String = "" //armazenar o país selecionado no Spinner

        //Variáveis de local:
        var etLocal: EditText //campo para digitar o local
        etLocal = findViewById(R.id.etLocal) //localiza o campo com o id etLocal no layout

        //=====Tratamento do botão Pesquisar
        var btOk: Button
        btOk = findViewById(R.id.btOk)
        btOk.setOnClickListener {
            var local: String = etLocal.text.toString() //armazenar o local digitado
            frase = "Favor elaborar um roteiro turístico para o local $local"
            Toast.makeText(this, frase, Toast.LENGTH_LONG).show()
        }
    }
}
```

Figura 313

Ao digitar o local, temos o Toast exibindo o resultado:

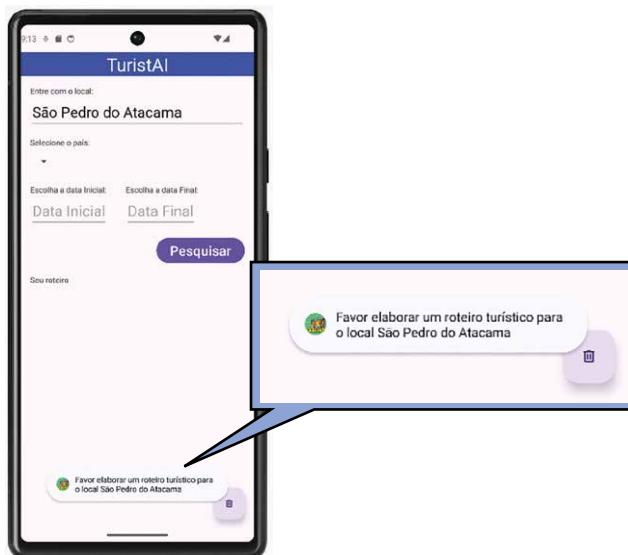


Figura 314

Configuração da lista de países

Este trecho de código lida com a exibição e seleção de países em um spinner na Activity principal do seu aplicativo de roteiro turístico.

- **países**: declaração de um array de strings que define a lista de países disponíveis para o usuário selecionar. No exemplo, ele contém Brasil, Argentina, Chile, Uruguai e Paraguai.

```
var países = arrayOf("Brasil", "Argentina", "Chile", "Uruguai", "Paraguai")
```

Para que o Spinner funcione corretamente, ele precisa de um adapter. Um adapter é uma ponte entre o Spinner e a fonte de dados que ele exibe. Ele converte os dados do array ou de outra fonte de dados em visualizações que podem ser exibidas no Spinner. Existem diferentes tipos de adapters, como ArrayAdapter, CursorAdapter, entre outros, dependendo da fonte de dados. No caso do ArrayAdapter, ele pega um array de strings (ou outros tipos de dados) e cria uma visualização para cada item do array, que é então exibida no Spinner. O adapter também define como cada item deve ser exibido.

O Spinner é um componente de interface de usuário que permite ao usuário selecionar um item de uma lista suspensa. Ele é semelhante a um menu dropdown em outras plataformas. Para usá-lo, você precisa definir um array de itens que serão exibidos e associá-lo a um adapter, que converte os itens do array em visualizações que podem ser exibidas no Spinner. O Spinner é útil para economizar espaço na tela, pois exibe apenas o item selecionado até que o usuário clique nele para ver a lista completa. Além disso, é possível configurar um listener para capturar eventos de seleção, permitindo que seu aplicativo responda quando o usuário escolhe um item da lista.

Nos processos para o tratamento do Spinner são necessárias as seguintes etapas:

- Criar o Spinner.
- Criar e configurar o adapter.
- Associar o adapter ao Spinner.
- Tratar a seleção do usuário.
- Tratar do item não selecionado.

Criar o Spinner

- **listaPaises**: variável do tipo Spinner que será usada para exibir a lista de países e permitir ao usuário selecionar um país.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
var listaPaises: Spinner
```

- **findViewById(R.id.spinner)**: associa a variável listaPaises ao componente Spinner no layout XML que possui o ID spinner.

```
listaPaises = findViewById(R.id.spinner)
```

Criar e configurar o Adapter

- **adapter = ArrayAdapter(this, android.R.layout.simple_spinner_item, paises)**: cria um adaptador do tipo ArrayAdapter chamado 'adapter'. Ele é específico para Spinners e sabe como exibir dados simples como strings.

- **this**: refere-se à Activity atual (MainActivity).
- **android.R.layout.simple_spinner_item**: define o layout padrão do Android para exibir cada item no Spinner.
- **paises**: é a lista de países criada anteriormente que será usada para preencher o Spinner.

```
var adapter = ArrayAdapter(this, android.R.layout.simple_spinner_item, paises)
```

- **setDropDownViewResource**: define para que o layout do 'adapter' usado exiba os itens quando o dropdown do Spinner é expandido para o padrão android.R.layout.simple_spinner_dropdown_item.

```
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
```

Associar o adapter ao spinner

- **listaPaises.adapter=adapter**: associa o Adapter ao Spinner para que ele possa exibir os dados. Com isso, o Spinner saberá como exibir os itens da lista de países (paises).

```
listaPaises.adapter = adapter
```

Tratar a seleção do usuário

- **listapaises.onItemSelectedListener**: listaPaises é a variável que representa o Spinner que exibe a lista de países. O método onItemSelectedListener define um listener para o Spinner. Um listener é um objeto que escuta eventos específicos, neste caso, eventos de seleção de itens no Spinner.

```
listaPaises.onItemSelectedListener =
```

- **object**: AdapterView.OnItemSelectedListener: uma instância anônima de uma classe que implementa a interface AdapterView.OnItemSelectedListener é criada. Esta interface possui dois métodos que precisamos implementar: onItemSelected e onNothingSelected.

```
object : AdapterView.OnItemSelectedListener()
```

- **override fun onItemSelected**: método chamado sempre que um item é selecionado no Spinner. Vamos detalhar os seus parâmetros.

```
override fun onItemSelected()
```

Vamos detalhar os parâmetros deste método.

- **parent: AdapterView<*>?**: este parâmetro representa o AdapterView (neste caso, o Spinner) em que a seleção ocorreu. O tipo AdapterView<*>? indica que pode ser qualquer tipo de AdapterView e ser nulo.

```
parent: AdapterView<*>?,
```

- **view: View?**: este parâmetro representa a View no AdapterView que foi clicada. Pode ser nulo.

```
view: View?,
```

- **position: Int**: este parâmetro representa a posição do item selecionado no Adapter. É um valor inteiro que começa em 0 para o primeiro item.

```
position: Int,
```

- **id: Long**: este parâmetro representa o ID do item selecionado. Em muitos casos, este valor é o mesmo que position, mas pode ser diferente dependendo do Adapter usado.

```
id: Long
```

```
listaPaises.onItemSelectedListener = object :  
    AdapterView.OnItemSelectedListener {  
        override fun onItemSelected(  
            parent: AdapterView<*>?,  
            view: View?,  
            position: Int,  
            id: Long  
        )
```

Dentro do método onItemSelected:

- **pais = paises[position]**: recupera o país selecionado acessando a lista paises na posição indicada por position.

```
pais = paises[position]
```

Tratar do item não selecionado

- **override fun onNothingSelected**: método chamado quando nenhum item é selecionado. No exemplo fornecido, ele não foi implementado, mas poderia ser usado para definir um comportamento padrão quando nenhuma seleção é feita. Ele somente está implementado, pois o AdapterView exige a presença deste método.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- TODO("Nada implementado") indica que a funcionalidade para tratar a situação de nenhum item selecionado ainda não foi implementada. Você pode personalizá-la para exibir uma mensagem de erro ou comportamento desejado.

```
override fun onNothingSelected(parent: AdapterView<*>?) {  
    TODO("Nada implementado")  
}
```

Antes de continuarmos, vamos alterar a frase do botão pesquisar:

```
frase = "Favor elaborar um roteiro turístico para o local $local, no país $pais"
```



Figura 315

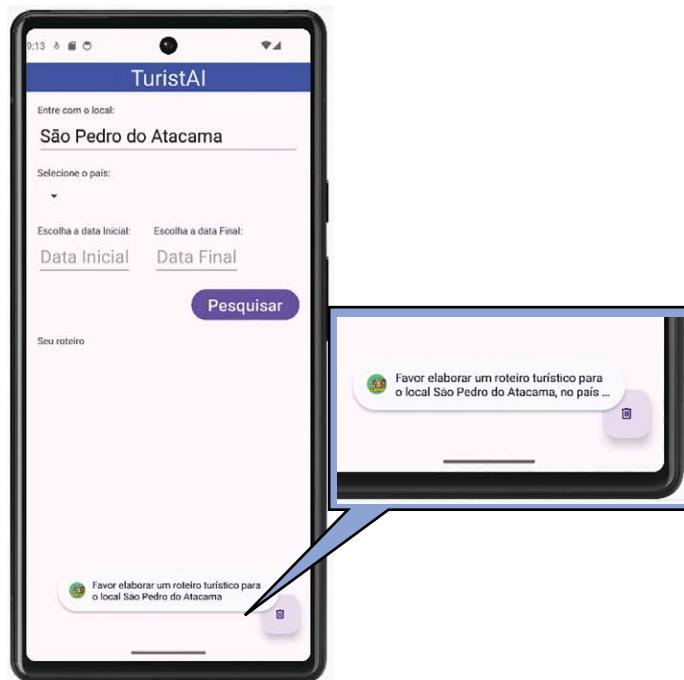


Figura 316

A implementação fica:

```
//=====Tratamento da lista dePaíses
//Variáveis de listaPaises:
var países = arrayOf("Brasil", "Argentina", "Chile", "Uruguai", "Paraguai")
var listaPaises: Spinner
listaPaises = findViewById(R.id.spinner)
//O spinner precisa receber um adapter para que ele possa exibir os dados
var adapter = ArrayAdapter(this, android.R.layout.simple_spinner_item, países)
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
//listaPaises.adapter = adapter
listaPaises.adapter = adapter
listaPaises.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
    override fun onItemSelected(
        parent: AdapterView<*>?,
        view: View?,
        position: Int,
        id: Long
    ) {
        //retorna a posição do item selecionado
        pais = países[position]
        Toast.makeText(this@MainActivity, "País Selecionado: $pais", Toast.LENGTH_SHORT)
            .show()
    }
    override fun onNothingSelected(parent: AdapterView<*>?) {
        TODO("Nada implementado")
    }
}
}
```

Figura 317

Lembrando que a implementação fica antes do Tratamento do Botão:

- **Tratamento de data inicial e final:** selecionando datas com DatePickerDialoga:

– **etDataIni:** representa o campo de texto onde o usuário insere a data inicial da viagem (EditText).

```
var etDataIni: EditText
```

– **dataini:** armazena a data inicial da viagem como string.

```
var dataini: String=""
```

– **etDataFim:** representa o campo de texto onde o usuário insere a data final da viagem (EditText).

```
var etDataFim: EditText
```

– **datafim:** armazena a data final da viagem como string.

```
var datafim: String=""
```

– **cal:** objeto 'Calendar' usado para manipulação de datas.

```
var cal: Calendar
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- **findViewById(R.id.etDataIni)**: vincula a variável etDataIni ao campo de texto no layout XML que possui o ID etDataIni. Isso permite que o código manipule os campos de texto programaticamente.

```
etDataIni = findViewById(R.id.etDataIni)      //etDataIni é o id do objeto Data Inicial
```

- **findViewById(R.id.etDataFim)**: vincula a variável etDataFim ao campo de texto no layout XML que possui o ID etDataFim.

```
etDataFim = findViewById(R.id.etDataFim)      //etDataFim é o id do objeto Data Final
```

- **Calendar.getInstance()**: inicializa a variável cal com uma instância do Calendar, configurada para a data e hora atuais.

```
cal = Calendar.getInstance()                  //cal é o objeto Calendar
```

- **setOnClickListener**: define um listener para o evento onClick para os campos de texto etDataIni e etDataFim. Quando o usuário clica nesses campos, o método showDatePickerDialog é chamado.

```
etDataIni.setOnClickListener {  
    showDatePickerDialog(this, etDataIni, cal)  
}  
etDataFim.setOnClickListener {  
    showDatePickerDialog(this, etDataFim, cal)  
}
```

Os parâmetros passados na chamada do método são:

- **context**: contexto da Activity atual (MainActivity), ou seja, this.
- **editText**: campo de texto associado ao clique (etDataIni ou etDataFim).
- **cal**: é o objeto Calendar que representa data e hora atuais.

É necessário criar a função showDatePickerDialog. Ela será criada na classe MainActivity no mesmo nível do onCreate.

Unidade III

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        /* Código do OnCreate
    }
    fun showDatePickerDialog(context: Context, editText: EditText, cal: Calendar) {
        val ano = cal.get(Calendar.YEAR)
        val mes = cal.get(Calendar.MONTH)
        val dia = cal.get(Calendar.DAY_OF_MONTH)
        val datePickerDialog = DatePickerDialog(
            context,
            { view, ano, mes, dia ->
                val dataSelecionada = "$dia/${mes + 1}/${ano}"
                editText.setText(dataSelecionada)
                Toast.makeText(context, dataSelecionada, Toast.LENGTH_SHORT).show()
            }, ano, mes, dia
        )
        datePickerDialog.show()
    }
}
```

Figura 318

- **fun showDatePickerDialog(context: Context, editText: EditText, cal: Calendar)**: a função cria e exibe um diálogo do tipo DatePickerDialog. Este diálogo permite ao usuário selecionar uma data usando seletores de dia, mês e ano.

```
fun showDatePickerDialog(context: Context, editText: EditText, cal: Calendar)
```

- **cal.get(Calendar.YEAR)**: obtém o ano atual do objeto Calendar.
- **cal.get(Calendar.MONTH)**: obtém o mês atual do objeto Calendar.
- **cal.get(Calendar.DAY_OF_MONTH)**: obtém o dia atual do objeto Calendar.

```
val ano = cal.get(Calendar.YEAR)
val mes = cal.get(Calendar.MONTH)
val dia = cal.get(Calendar.DAY_OF_MONTH)
```

- **DatePickerDialog(Context, OnDateSetListener, Year, Month, DayOfMonth)**: cria uma instância do DatePickerDialog.

O DatePickerDialog é um componente de interface de usuário no Android que permite aos usuários selecionar uma data (dia, mês e ano) através de uma caixa de diálogo. Quando ele é exibido, mostra um calendário ou uma interface de rolagem na qual o usuário pode escolher a data desejada. Este componente é útil para garantir que os usuários selecionem uma data válida e um formato correto, facilitando a entrada de dados em aplicativos que requerem informações de data.

Ao criar um DatePickerDialog, você pode especificar a data inicial que será exibida, bem como definir um listener para capturar a data selecionada pelo usuário. Quando o usuário escolhe uma data e confirma, o listener é acionado, permitindo que o aplicativo processe a data escolhida.

Os parâmetros são:

- **Context**: o contexto atual da aplicação, geralmente a atividade. Aqui é o valor recebido pela variável contexto, que é o parâmetro da função showDatePickerDialog.
- **OnDateSetListener**: um listener que é chamado quando o usuário seleciona uma data. Este listener define o que acontece quando a data é escolhida. { view, ano, mes, dia -> ... }: um lambda que define o que acontece quando uma data é selecionada.
- **Year**: o ano inicial que será exibido no diálogo. Ano atual obtido no qual a variável foi criada e recebeu o valor de cal.get(Calendar.YEAR).
- **Month**: o mês inicial que será exibido no diálogo. Note que os meses são indexados a partir de 0 (Janeiro é 0, Fevereiro é 1 etc.). Mês atual obtido no qual a variável foi criada.
- **DayOfMonth**: o dia inicial que será exibido no diálogo. Dia atual obtido no qual a variável foi criada.

O OnDateSetListener é {view, ano, mes, dia -> ...}: um lambda que define o que acontece quando uma data é selecionada.

O corpo do Lambda é dado por:

- **val dataSelecionada = "\$dia/\${mes + 1}/\$ano"**: formata a data selecionada como uma string.
- **editText.setText(dataSelecionada)**: define o texto do campo de texto com a data selecionada.

```
val datePickerDialog = DatePickerDialog(  
    context,  
    { view, ano, mes, dia ->  
        val dataSelecionada = "$dia/${mes + 1}/$ano"  
        editText.setText(dataSelecionada)  
    }, ano, mes, dia  
)
```

- **datePickerDialog.show()**: exibição do DatePickerDialog.
 - datePickerDialog.show()

Unidade III

Assim, ao clicar em Data Inicial ou Data Final no emulador, temos:



Figura 319

No main então o código:

```
//=====Tratamento da data inicial e final
var etDataIni: EditText
var etDataFim: EditText
var dataini: String=""
var datafim: String=""
var cal: Calendar

etDataIni = findViewById(R.id.etDataIni) //etDataIni é o id do objeto Data Inicial
etDataFim = findViewById(R.id.etDataFim) //etDataFim é o id do objeto Data Final
cal = Calendar.getInstance()           //cal é o objeto Calendar
etDataIni.setOnClickListener {
    showDatePickerDialog(this, etDataIni, cal)
}
etDataFim.setOnClickListener {
    showDatePickerDialog(this, etDataFim, cal)
}
```

Figura 320

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Na classe MainActivity:

```
fun showDatePickerDialog(context: Context, editText: EditText, cal: Calendar) {  
    val ano = cal.get(Calendar.YEAR)  
    val mes = cal.get(Calendar.MONTH)  
    val dia = cal.get(Calendar.DAY_OF_MONTH)  
    val datePickerDialog = DatePickerDialog(  
        context,  
        { view, ano, mes, dia ->  
            val dataSelecionada = "$dia/${mes + 1}/$ano"  
            editText.setText(dataSelecionada)  
        }, ano, mes, dia  
    )  
    datePickerDialog.show()  
}
```

Figura 321

Vamos alterar o listener no botão para:

```
btOk.setOnClickListener {  
    var local: String = etLocal.text.toString() //armazenar o local digitado  
    dataini = etDataIni.text.toString()  
    datafim = etDataFim.text.toString()  
    frase = " $local, $pais , $dataini e $datafim"  
    Toast.makeText(this, frase, Toast.LENGTH_LONG).show()  
}
```

Temos:

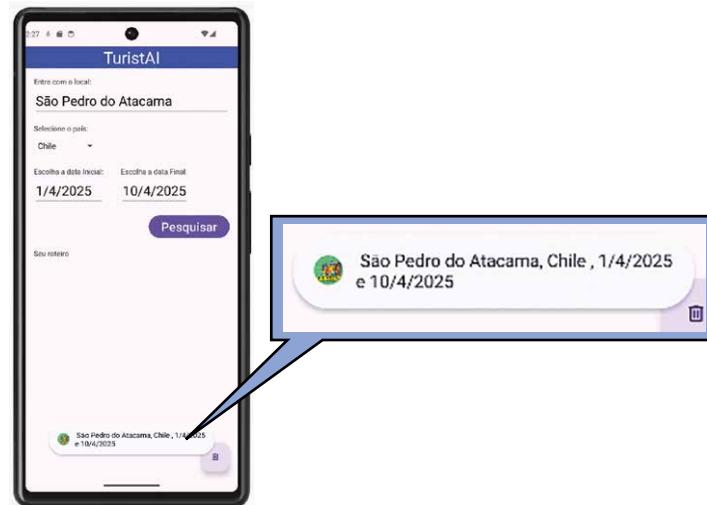


Figura 322

Desta forma, temos todas as informações necessárias para montar a nossa frase a ser utilizada para a consulta no Gemini:

```
frase =  
    "Favor elaborar um roteiro turístico para o local $local, no país $pais  
entre $dataini e $datafim"
```

Declaração da saída de texto

Precisamos preparar o TextView txtSaida para poder receber o resultado da consulta ao Gemini. Antes do listener do botão, crie:

```
//=====Definindo o TextView de saída  
var txtSaida: TextView  
txtSaida = findViewById(R.id.txtRetorno)
```

IA – implementar a API Gemini

É inegável que a capacidade de processamento e armazenamento de um dispositivo móvel, por mais avançado que seja, possui limitações. Para oferecer funcionalidades complexas e abrangentes, como a geração de informações turísticas detalhadas em tempo real para qualquer lugar do mundo no nosso aplicativo, é necessário recorrer a recursos externos. Assim, as APIs desempenham um papel fundamental, permitindo que nosso aplicativo se conecte a serviços em nuvem, como no nosso caso a API Gemini do Google.

Ao utilizar a API Gemini, deixamos a tarefa de processar e gerar informações turísticas para servidores com alto poder de processamento e utilizar a IA, enquanto o aplicativo móvel se concentra em apresentar os dados para o usuário. Essa abordagem não apenas otimiza o desempenho do aplicativo, mas também garante que o usuário tenha acesso a um banco de dados vasto e atualizado de informações turísticas, independentemente de sua localização.



Saiba mais

Para mais opções de configuração, a página para desenvolvedores do Google oferece um tutorial bastante acessível. Acessa-a em:

GOOGLE AI FOR DEVELOPERS. *Tutorial: primeiros passos com a API Gemini.* [s.d.]. Disponível em: <https://shre.ink/MVFH>. Acesso em: 6 mar 2025.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

A integração da API Gemini em um aplicativo Android Studio para realizar consultas de texto envolve algumas etapas:

- Configurar o ambiente de desenvolvimento.
- Obter uma chave de API.
- Implementar a lógica de consulta.
- Obter a resposta.

Para configurar o ambiente, vamos acessar o site citado previamente (<https://shre.ink/MVFH>). Utilizá-lo é importante, pois as versões do API atualizadas estarão nele.



Figura 323

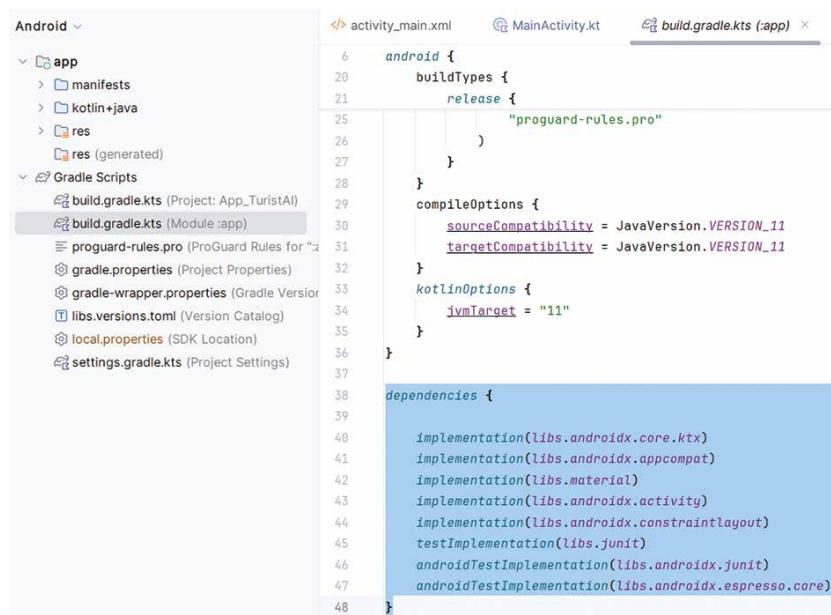
Escolha Android e a linguagem Kotlin.

Configurar o ambiente de desenvolvimento

Vamos adicionar a dependência do SDK ao projeto Android Studio, o que significa incluir no seu projeto as bibliotecas e ferramentas necessárias para utilizar um Software Development Kit (SDK) específico. Isso é feito editando o arquivo build.gradle do seu aplicativo para incluir as dependências do SDK, que são pacotes de código que fornecem funcionalidades adicionais. Ao adicionar a dependência de uma API, você está permitindo que seu projeto utilize os recursos dela, como métodos e classes, facilitando a integração e o desenvolvimento de funcionalidades avançadas sem precisar escrever todo o código do zero.

Unidade III

Abra o projeto existente, no arquivo build.gradle (nível do aplicativo).



The screenshot shows the Android Studio interface. On the left, the project structure is visible with 'app' selected. The main area displays the contents of the 'build.gradle.kts' file. The code is as follows:

```
6 android {
10     buildTypes {
11         release {
15             "proguard-rules.pro"
16         }
17     }
18 }
19
20 compileOptions {
21     sourceCompatibility = JavaVersion.VERSION_11
22     targetCompatibility = JavaVersion.VERSION_11
23 }
24
25 kotlinOptions {
26     jvmTarget = "11"
27 }
28
29 dependencies {
30     implementation(libs.appcompat)
31     implementation(libs.material)
32     implementation(libs.core.ktx)
33     implementation(libs.constraintlayout)
34     testImplementation(libs.junit)
35     androidTestImplementation(libs.espresso.core)
36     androidTestImplementation(libs.junit)
37 }
```

Figura 324

No site do developer, localize o script da dependência.

Adicionar a dependência do SDK ao seu projeto

1. No arquivo de configuração Gradle do **módulo (nível do app)** (como `<project>/<app-module>/build.gradle.kts`), adicione a dependência SDK da IA do Google para Android:



```
Kotlin Java
dependencies {
    // ... other androidx dependencies

    // add the dependency for the Google AI client SDK for Android
    implementation("com.google.ai.client.generativeai:generativeai:0.9.0")
}
```

Figura 325

Copie e cole nas dependências do seu projeto:



```
dependencies {

    implementation(libs.appcompat)
    implementation(libs.material)
    implementation(libs.core.ktx)
    implementation(libs.constraintlayout)
    implementation(libs.junit)
    androidTestImplementation(libs.junit)
    androidTestImplementation(libs.espresso.core)

    // add the dependency for the Google AI client SDK for Android
    implementation("com.google.ai.client.generativeai:generativeai:0.9.0")
}
```

Figura 326

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Sincronize o Gradle e espere o processo terminar.



Figura 327

ou

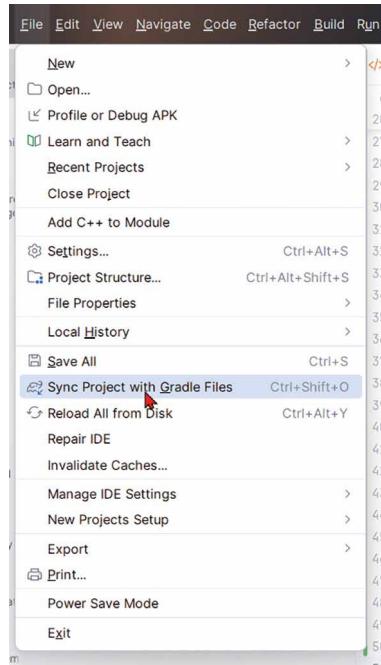


Figura 328

Obter uma chave de API

Para utilizar a API Gemini, é necessário possuir uma chave de API. A chave de API do Gemini é necessária porque atua como um identificador único que autentica e autoriza o acesso à API. Caso ainda não tenha uma, você pode criá-la no Google AI Studio.

Para criar a chave, clique no botão do site do developers que você será direcionado ao site do API Google.



Figura 329

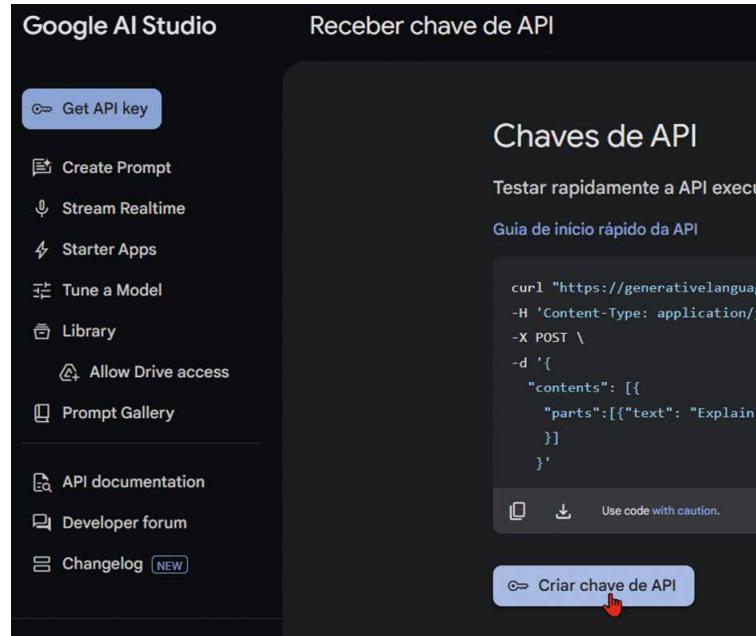


Figura 330

Preencha as informações e, ao obter a chave, copie-a para um lugar seguro:



Figura 331

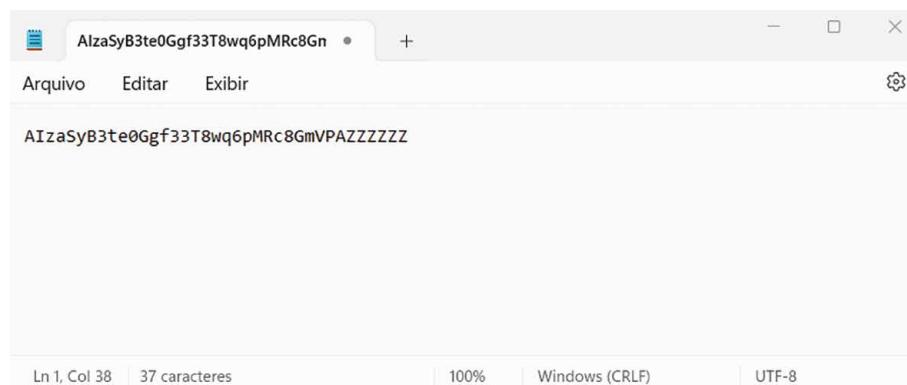


Figura 332

Implementar a lógica de consulta

Definição da função

- **fun chamaGemini(txtSaida: TextView, frase: String):** define uma função chamada chamaGemini que recebe dois parâmetros: txtSaida (um TextView onde o resultado será exibido) e frase (uma String que será usada como prompt para a API).

```
fun chamaGemini(txtSaida: TextView, frase: String)
```

Essa função permite que você envie uma frase para a API Gemini, receba uma resposta gerada e a exiba em um TextView no seu aplicativo Android.

Instância do modelo generativo

O texto da instância pode ser obtido no site do developers:

Gerar texto com base em uma entrada somente de texto

Quando a entrada do comando incluir apenas texto, use um modelo Gemini 1.5 ou o Modelo Gemini 1.0 Pro com `generateContent` para gerar uma saída de texto:

The screenshot shows a snippet of Kotlin code for generating text from a prompt using the Gemini model. The code is as follows:

```
Kotlin Java  
generateContent() é uma função de suspensão e precisa ser chamados de um escopo de corوتina. Se você não conhece as corوتinas, leia Corوتinas do Kotlin no Android.  
val generativeModel = GenerativeModel(  
    // The Gemini 1.5 models are versatile and work with both text-only and multimodal prompts  
    modelName = "gemini-1.5-flash",  
    // Access your API key as a Build Configuration variable (see "Set up your API key" above)  
    apiKey = BuildConfig.apiKey  
)  
  
val prompt = "Write a story about a magic backpack."  
val response = generativeModel.generateContent(prompt)  
print(response.text)
```

Figura 333

- **var generativeModel = GenerativeModel(...):** cria uma instância da classe GenerativeModel com dois parâmetros:
 - **modelName = "gemini-1.5-flash":** especifica o nome do modelo Gemini a ser usado.
 - **apiKey = "sua_chave_de_API":** fornece a chave de API necessária para autenticação.

```
var generativeModel =  
    GenerativeModel(  
        // Specify a Gemini model appropriate for your use case  
        modelName = "gemini-1.5-flash",  
        // Access your API key as a Build Configuration variable (see "Set up  
        // your API key" above)  
        apiKey = "AIzaSyB3te0Ggf33T8wq6pMRc8GmVXXXXXXXXXX"  
    )
```

Definição do prompt:

- **val prompt = frase**: atribui a string frase ao prompt, que será usado para gerar o conteúdo.

```
val prompt = frase
```

Lançamento da coroutine:

- **MainScope().launch { ... }**: inicia uma coroutine no escopo principal, permitindo a execução de código assíncrono.

```
MainScope().launch {
```



Coroutine é uma construção de programação assíncrona que permite a execução de tarefas de forma concorrente sem bloquear a thread principal. Em Kotlin, coroutines são usadas para simplificar o código assíncrono, tornando-o mais legível e fácil de manter. Elas permitem que você escreva código que parece síncrono, mas na verdade é executado de forma assíncrona, permitindo operações como chamadas de rede ou acesso a banco de dados sem bloquear a interface do usuário. Coroutines são gerenciadas pelo Kotlin Coroutine Library, que fornece um conjunto de APIs para iniciar, pausar e retomar tarefas de maneira eficiente e controlada.

Um código assíncrono é um tipo de programação no qual as operações podem ser executadas de forma independente do fluxo principal do programa, permitindo que outras tarefas continuem a ser processadas enquanto uma operação está em andamento. Isso é especialmente útil para tarefas que podem levar tempo, como chamadas de rede, leitura de arquivos ou acesso a bancos de dados, pois evita que a interface do usuário fique bloqueada ou não responsiva. Em vez de esperar que uma operação termine, o código assíncrono possibilita que o programa continue executando outras tarefas e, quando a operação assíncrona for concluída, um callback ou uma coroutine pode ser usado para processar o resultado.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Geração de conteúdo:

- **val response = generativeModel.generateContent(prompt)**: chama o método generateContent do modelo gerativo, passando o prompt e recebendo a resposta.

```
val response = generativeModel.generateContent(prompt)
```

Atualização da UI:

- **txtSaida.setText(response.text)**: define o texto do TextView txtSaida com o texto gerado pela API.

```
txtSaida.setText(response.text)
```

Assim, a função completa fica:

```
fun chamaGemini(txtSaida: TextView, frase: String) {  
    var generativeModel =  
        GenerativeModel(  
            // Specify a Gemini model appropriate for your use case  
            modelName = "gemini-1.5-flash",  
            // Access your API key as a Build Configuration variable (see  
            apiKey = "AIzaSyB3te0Ggf33T8wq6pMrc8GmVPXXXXXX"  
        )  
    val prompt = frase  
    MainScope().launch {  
        val response = generativeModel.generateContent(prompt)  
        txtSaida.setText(response.text)  
    }  
}
```

Figura 334

No botão implemente a chamada à função chamaGemini.

```
chamaGemini(txtSaida, frase)
```

Ao clicar em Pesquisar, temos:

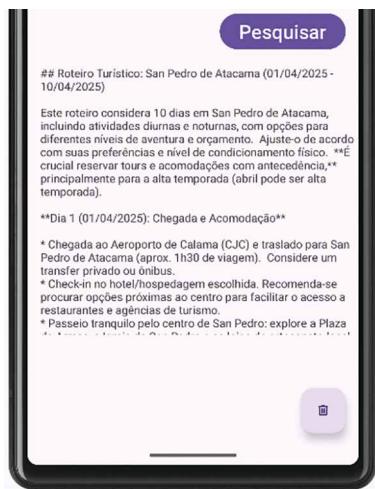


Figura 335

Configuração do botão de limpar

O botão de limpeza é um FAB (Floating Action Button) ou botão de ação flutuante. Este é um elemento visual comum em aplicativos Android, que se destaca na interface do usuário, geralmente com um ícone circular ou quadrado, e serve para realizar ações principais. Quando o usuário clica no botão FAB, ele limpa o conteúdo de quatro campos de texto: etLocal, etDataIni, etDataFim e txtSaida da activity.

- **var fabLixo: View:** cria uma variável do tipo View chamada fabLixo. Ela será usada para representar o botão FAB no código.

```
var fabLixo: View
```

- **fabLixo = findViewById(R.id.fabLixo):** busca o elemento com o ID fabLixo no layout XML do aplicativo e o associa à variável fabLixo. Isso significa que, a partir desse momento, a variável fabLixo representa o botão FAB definido no layout.

```
fabLixo = findViewById(R.id.fabLixo)
```

- **fabLixo.setOnClickListener { ... }:** define o que acontece quando o usuário clica no botão FAB. O código nas chaves será executado.

```
fabLixo.setOnClickListener {
```

- **etLocal.setText(""):** limpa o conteúdo do campo de texto com o ID etLocal.
- **etDataIni.setText(""):** limpa o conteúdo do campo de texto com o ID etDataIni.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- `etDataFim.setText("")`: limpa o conteúdo do campo de texto com o ID etDataFim.
- `txtSaida.setText("")`: limpa o conteúdo do campo de texto com o ID txtSaida.

```
etLocal.setText("")  
etDataIni.setText("")  
etDataFim.setText("")  
txtSaida.setText("")
```

Assim, o código correspondente ao botão de limpeza fica:

```
var fabLixo: View  
fabLixo = findViewById(R.id.fabLixo)  
fabLixo.setOnClickListener {  
    etLocal.setText("")  
    etDataIni.setText("")  
    etDataFim.setText("")  
    txtSaida.setText("")  
}
```

Figura 336

* Clicar no botão de limpar ('fabLixo') executa a limpeza do texto de todos os campos de entrada (local, data inicial, data final) e do campo de texto de saída.

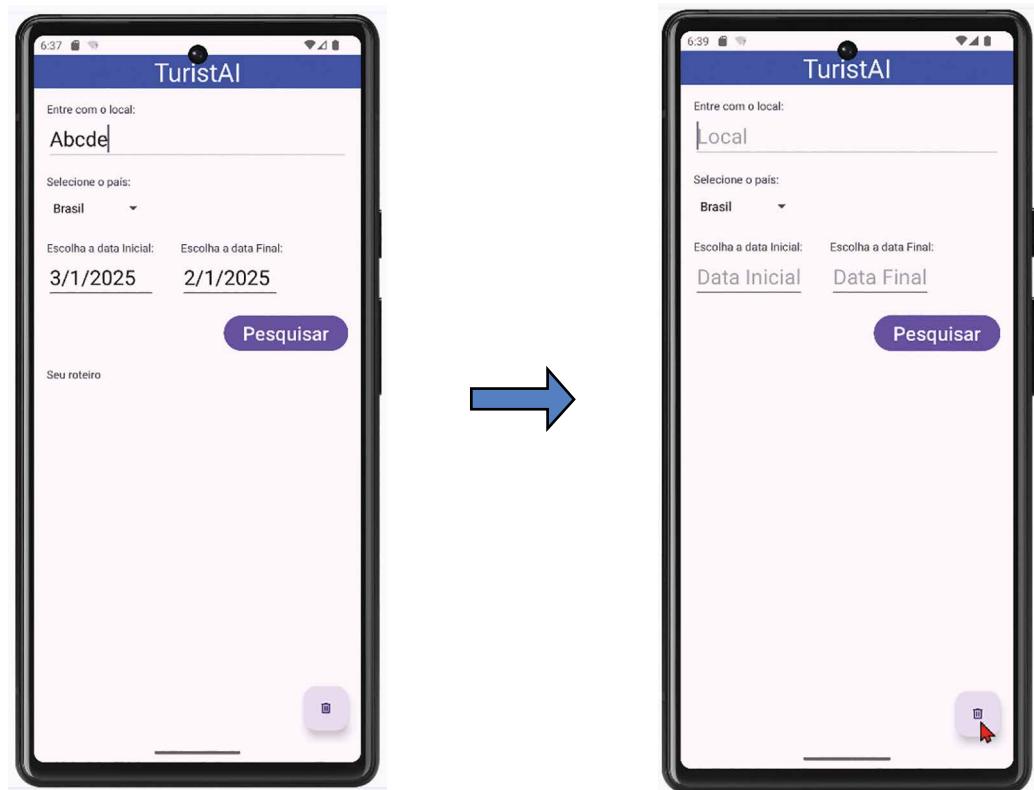


Figura 337

O código completo é:

activity_main.xml:

```
<?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:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#3F51B5"
        android:text="TuristaAI"
        android:textAlignment="center"
        android:textColor="@color/white"
        android:textSize="32dp" />
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:text="Entre com o local:" />
    <EditText
        android:id="@+id/etLocal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="16dp"
        android:backgroundTint="@color/material_dynamic_neutral80"
        android:hint="Local"
        android:hyphenationFrequency="normal"
        android:inputType="text"
        android:textSize="26dp" />
    <TextView
        android:id="@+id/textView4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:text="Selecione o país:" />
    <Spinner
        android:id="@+id/spinner"
        android:layout_width="wrap_content"
        android:layout_height="48dp"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="16dp" />
```

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:text="Escolha a data Inicial:" />

    <EditText
        android:id="@+id/etDataIni"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="16dp"
        android:clickable="false"
        android:cursorVisible="false"
        android:focusable="false"
        android:focusableInTouchMode="false"
        android:hint="Data Inicial"
        android:inputType="datetime|date"
        android:textSize="26sp" />
</LinearLayout>

<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:text="Escolha a data Final:" />
    <EditText
        android:id="@+id/etDataFim"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="16dp"
        android:clickable="false"
        android:cursorVisible="false"
        android:focusable="false"
        android:focusableInTouchMode="false"
        android:hint="Data Final"
        android:inputType="datetime|date"
        android:textSize="26sp" />
</LinearLayout>
```

Unidade III

```
</LinearLayout>
<Button
    android:id="@+id/btOk"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="right"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="16dp"
    android:text="Pesquisar"
    android:textSize="24sp" />
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="0.9"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:layout_marginBottom="50dp">
    <TextView
        android:id="@+id/txtRetorno"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Seu roteiro" />
</ScrollView>

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fabLixo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|right"
    android:layout_margin="16dp"
    android:contentDescription="Limpar"
    android:src="@drawable/ic_lixo_foreground" />
</LinearLayout>
```

MainActivity.kt:

```
package com.example.appturistai

import android.app.DatePickerDialog
import android.content.Context
import android.os.Bundle
import android.view.View
import android.widget.AdapterView
import android.widget.ArrayAdapter
import android.widget.Button
import android.widget.EditText
import android.widget.Spinner
import android.widget.TextView
import android.widget.Toast
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat
import com.google.ai.client.generativeai.GenerativeModel
import kotlinx.coroutines.MainScope
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
import kotlinx.coroutines.launch
import java.util.Calendar

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
}
//Variáveis de Trabalho
var frase: String = "" //frase de envio ao API
var pais: String = "" //armazenar o país selecionado no Spinner

//Variáveis de local:
var etLocal: EditText //campo para digitar o local
etLocal = findViewById(R.id.etLocal) //localiza o campo com o id etLocal
no layout
//=====Tratamento da lista dePaíses
//Variáveis de listaPaises:
var paises = arrayOf("Brasil", "Argentina", "Chile", "Uruguai",
"Paraguai")
var listaPaises: Spinner
listaPaises = findViewById(R.id.spinner)
//O spinner precisa receber um adapter para que ele possa exibir os
dados
var adapter = ArrayAdapter(this,
android.R.layout.simple_spinner_item, paises)
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
//listaPaises.adapter = adapter
listaPaises.adapter = adapter
listaPaises.onItemSelectedListener = object :
AdapterView.OnItemSelectedListener {
    override fun onItemSelected(
        parent: AdapterView<*>?,
        view: View?,
        position: Int,
        id: Long
    ) {
        //retorna a posição do item selecionado
        pais = paises[position]
        Toast.makeText(this@MainActivity, "País Selecionado: $pais",
Toast.LENGTH_SHORT)
            .show()
    }
    override fun onNothingSelected(parent: AdapterView<*>?) {
        TODO("Nada implementado")
    }
}
//=====Tratamento da data inicial e final
```

Unidade III

```
var etDataIni: EditText
var etDataFim: EditText
var dataini: String=""
var datafim: String=""
var cal: Calendar

etDataIni = findViewById(R.id.etDataIni) //etDataIni é o id do objeto
Data Inicial
etDataFim = findViewById(R.id.etDataFim) //etDataFim é o id do objeto
Data Final
cal = Calendar.getInstance() //cal é o objeto Calendar
etDataIni.setOnClickListener {
    showDatePickerDialog(this, etDataIni, cal)
}
etDataFim.setOnClickListener {
    showDatePickerDialog(this, etDataFim, cal)
}
//=====Definindo o TextView de saída
var txtSaida: TextView
txtSaida = findViewById(R.id.txtRetorno)
//=====Tratamento do botão Pesquisar
var btOk: Button
btOk = findViewById(R.id.btOk)
btOk.setOnClickListener {
    var local: String = etLocal.text.toString() //armazenar o local
digitado
    dataini = etDataIni.text.toString()
    datafim = etDataFim.text.toString()
    frase =
        "Favor elaborar um roteiro turístico para o local $local, no
país $pais entre $dataini e $datafim"

    Toast.makeText(this, " $local, $pais , $dataini e $datafim" ,
Toast.LENGTH_LONG).show()

    chamaGemini(txtSaida, frase)
}
var fabLixo: View
fabLixo = findViewById(R.id.fabLixo)
fabLixo.setOnClickListener {
    etLocal.setText("")
    etDataIni.setText("")
    etDataFim.setText("")
    txtSaida.setText("")
}
fun showDatePickerDialog(context: Context, editText: EditText, cal:
Calendar) {
    val ano = cal.get(Calendar.YEAR)
    val mes = cal.get(Calendar.MONTH)
    val dia = cal.get(Calendar.DAY_OF_MONTH)
    val datePickerDialog = DatePickerDialog(
        context,
        { view, ano, mes, dia ->
            val dataSelecionada = "$dia/${mes + 1}/$ano"
            editText.setText(dataSelecionada)
        }
    )
}
```

```
        }, ano, mes, dia
    )
    datePickerDialog.show()
}
fun chamaGemini(txtSaida: TextView, frase: String) {
    var generativeModel =
        GenerativeModel(
            // Specify a Gemini model appropriate for your use case
            modelName = "gemini-1.5-flash",
            // Access your API key as a Build Configuration variable (see
            "Set up your API key" above)
            apiKey = "AIzaSyB3te0Ggf33T8wq6pMRC8GmVPAGZB8ul4w"
        )
    val prompt = frase
    MainScope().launch {
        val response = generativeModel.generateContent(prompt)
        txtSaida.setText(response.text)
    }
}
```

Teste do aplicativo

O processo de teste no Android Studio é essencial para garantir que seu aplicativo funcione corretamente e sem bugs. Constanam a seguir os principais aspectos do processo de teste:

Tipos de testes

- Testes Locais (Unit Tests):
 - **Localização:** src/test/java/.
 - **Execução:** na JVM local do seu computador.

Uso: para testar unidades de código que não dependem do framework Android. São rápidos e ideais para testar lógica de negócios.

- Testes Instrumentados (Instrumented Tests):
 - **Localização:** src/androidTest/java/.
 - **Execução:** em um dispositivo físico ou emulador.

Uso: para testar a interação com a interface do usuário e componentes que dependem do framework Android. Eles têm acesso às APIs de instrumentação, permitindo controlar o app durante os testes.

Configuração e execução de testes

- Configuração:
 - **Gradle:** o Android Studio usa o Gradle para gerenciar e executar testes. Você pode configurar dependências de teste no arquivo 'build.gradle'.
 - **Diretórios de teste:** os testes são organizados em diretórios específicos ('src/test/java/' para testes locais e 'src/androidTest/java/' para testes instrumentados).
- Execução:
 - **IDE:** você pode executar testes diretamente no Android Studio. Basta clicar com o botão direito no arquivo de teste ou na classe de teste e selecionar Run.
 - **Linha de comando:** também é possível executar testes usando a linha de comando com o comando './gradlew test' para testes locais e './gradlew connectedAndroidTest' para testes instrumentados.
- Análise de resultados:
 - **Resultados:** os resultados dos testes são exibidos diretamente no Android Studio, mostrando quais testes passaram, falharam ou foram ignorados.
 - **Relatórios:** o Android Studio gera relatórios detalhados que ajudam a identificar e corrigir problemas.
- Testes avançados:
 - **Testes de integração contínua:** para projetos maiores, você pode configurar testes como parte de um pipeline de integração contínua, garantindo que cada alteração no código seja testada automaticamente.
 - **Testes de variantes de build:** você pode criar testes específicos para diferentes variantes de build, garantindo que todas as versões do seu app sejam testadas adequadamente.



Saiba mais

A fim de obter mais informações sobre os testes avançados, acesse:

DEVELOPERS. *Configuração de testes avançados*. [s.d.]c. Disponível em: <https://shre.ink/MNnp>. Acesso em: 6 mar. 2025.

7.2 Protótipo de um jogo

Agora vamos construir um jogo, ou melhor recriá-lo, o jogo da cobrinha. Conhecido como Snake, tem raízes nos arcades de 1976, sob o nome Blockade. No entanto, foi nos celulares Nokia que ele se tornou um fenômeno cultural (Benedetti, 2021). Lançado inicialmente no Nokia 6110, em 1997, conquistou milhões de jogadores por ser simples e viciante.



Figura 338 – Nokia 6110

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

A sua popularidade explodiu com o lançamento do Nokia 3310 em 2000, vendendo mais de 126 milhões de unidades. Desenvolvido por Taneli Armanto, o jogo foi brilhantemente adaptado para as limitações dos celulares da época, como telas monocromáticas e controles simples. A mecânica era viciante: controlar uma cobra que crescia ao comer itens, evitando obstáculos.

O legado do Snake é inegável. Além de divertir milhões de pessoas, ele pavimentou o caminho para os jogos mobile como conhecemos hoje. Em novembro de 2012, o Museu de Arte Moderna de Nova York anunciou que adicionou Snake à sua coleção de jogos eletrônicos notáveis (MoMA, [s.d.]). Sua influência pode ser vista em inúmeros jogos modernos, que continuam a explorar a fórmula simples e eficaz do comer e crescer.

O objetivo do jogo era controlar uma cobra que cresce ao comer alimentos. Na primeira versão, é necessário evitar colidir com as bordas e o corpo. A partir da segunda versão, a restrição foi removida, fazendo com que a cobra reapareça na borda oposta ao sair da tela. O desafio então passa a ser evitar colidir com o próprio corpo. A cobra se move continuamente e o jogador direciona seus movimentos com toques na tela. A cada alimento consumido, ela aumenta de tamanho, tornando as manobras mais desafiadoras.

Na elaboração de nosso jogo utilizaremos o Jetpack Compose, pois ele nos permitirá criar interfaces de usuário de forma mais intuitiva e eficiente sem a necessidade de nos preocupar com a construção de views e xml antes da codificação. Com o Jetpack Compose, podemos definir a interface do jogo de maneira declarativa, o que facilita a visualização e manutenção do código simultaneamente.

Portanto, vamos desenvolver o projeto em que trabalharemos:

- Inicie um novo projeto e escolha Empty Activity:

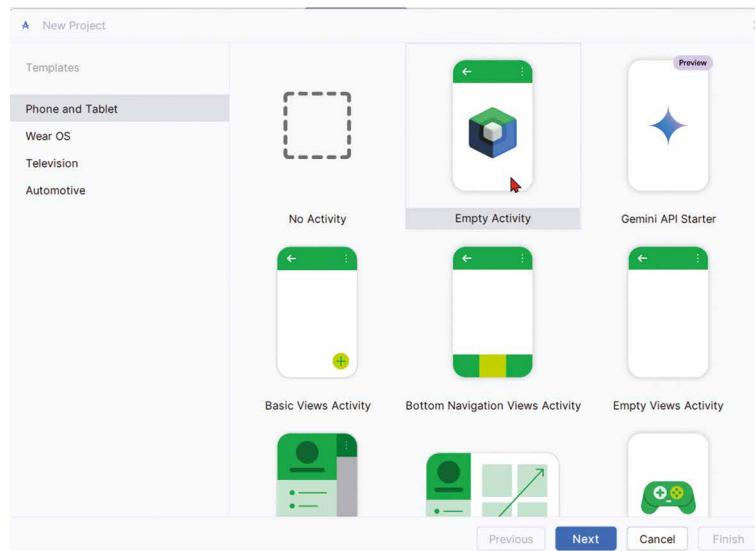


Figura 339

- Dê um nome ao projeto e mantenha todos os outros parâmetros.

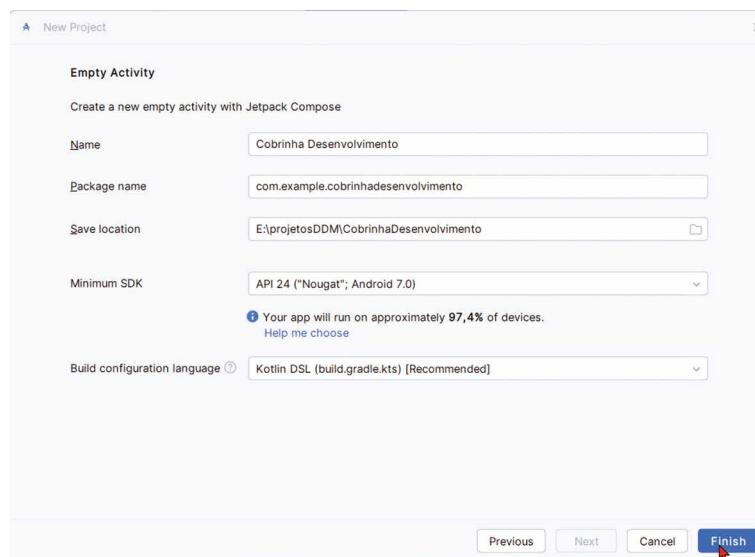


Figura 340

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- Espere com calma todo o Gradle ser montado e deixe a tela em modo Split.

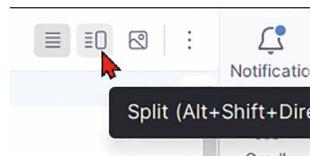


Figura 341

- Monte a visualização clicando em "Build & Refresh" ou combinando as teclas "Control-Shift-F5".



Caso aconteça o erro:

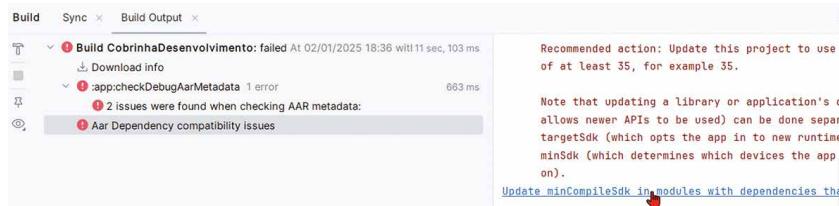


Figura 342

Clique em Update...

Clique em Do Refactor e espere o Gradle remontar.

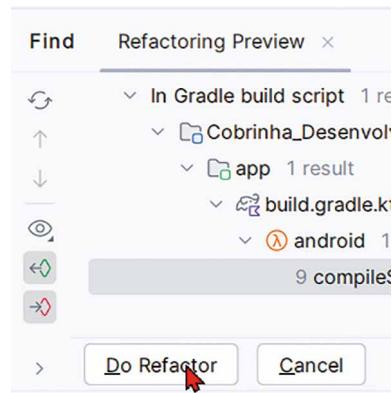


Figura 343

Para o emulador, recomenda-se utilizar o seu dispositivo ligado via cabo USB para ser mais dinâmico, mas pode-se usar o emulador sem problema.

7.2.1 Criar o cenário do jogo

O layout do jogo é composto por uma caixa na parte superior que serve como display de informações. Durante o jogo, ela exibe o placar, permitindo acompanhar a pontuação em tempo real. Ao final de cada partida, a mesma caixa mostra a mensagem "Game Over", indicando o fim da rodada. Abaixo do display, está localizado o campo de jogo propriamente dito, onde a ação se desenrola. Na parte inferior da tela, encontramos os controles: os botões direcionais para mover o personagem e os botões de Start para iniciar uma nova partida e Reset para reiniciar a partida atual () .

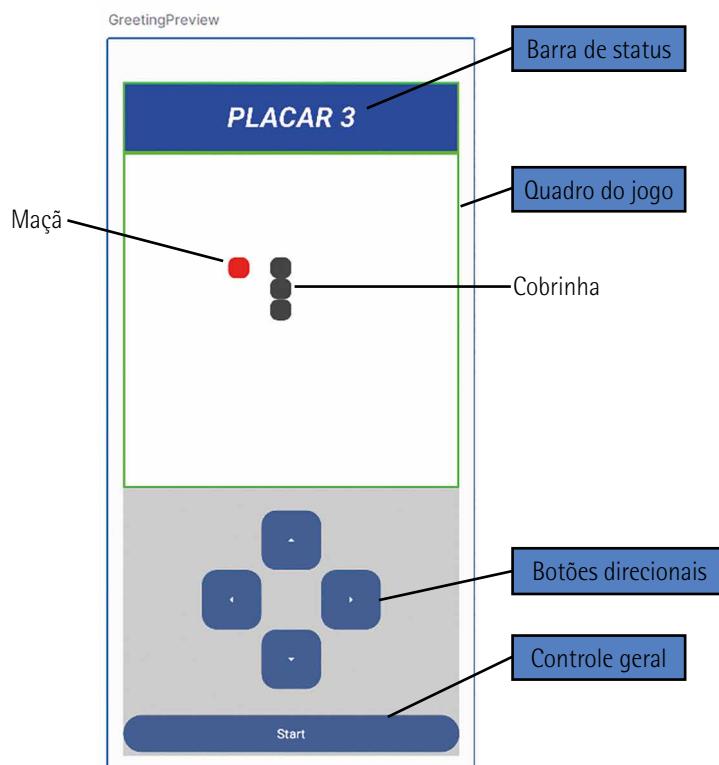


Figura 344 – Layout final do jogo

Configurando o projeto

Conforme previamente dito, utilizando o Jetpack Compose podemos montar a lógica e a nossa tela, uma activity e a lógica simultaneamente. Poderíamos deixar tudo em apenas um lugar, mas para facilitar a montagem, vamos separar a lógica em um novo arquivo.

- No projeto, criaremos uma pasta de Kotlin Class File. Para isso, siga o caminho Project/, clique com o botão direito na pasta onde está o MainActivity.kt, depois em new/ e selecione Kotlin Class File.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

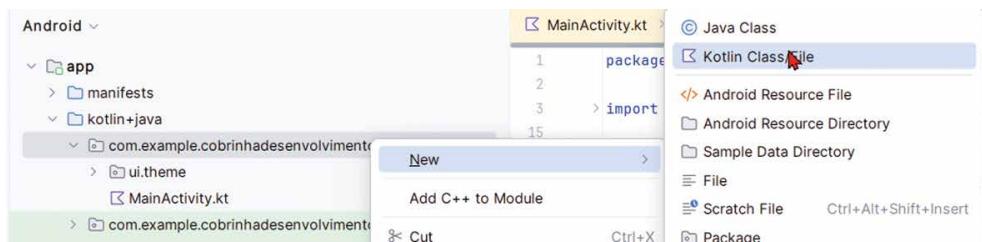


Figura 345

- Na nova janela, nomeie como Logica e escolha a opção File.

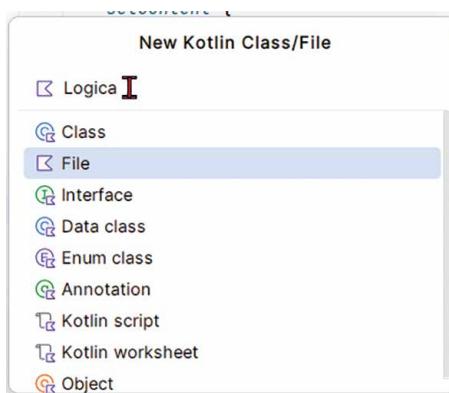


Figura 346

- Observe que a nova pasta está criada:

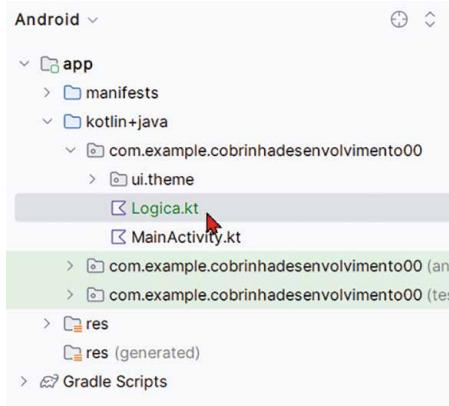


Figura 347

Para melhor organizar o nosso projeto, definimos que as telas do aplicativo estarão na MainActivity e a lógica de funcionamento isolada em uma pasta dedicada ao funcionamento do aplicativo. Antes de nos aprofundarmos no desenvolvimento do jogo, vamos observar os fundamentos da programação de aplicativos com Jetpack Compose e Kotlin.

Unidade III

- Na pasta Logica faremos um programa simples. Quando a função roda() de uma instância de Game for executada, o contador aumentará de zero a 1000.

```
class Game () {  
  
    // variáveis  
  
    var contador = 0  
    // ação  
  
    fun roda () {  
        while (contador < 1000) {  
            contador++  
        }  
    }  
}
```

Vamos desmembrar o código:

- **class Game{}:** essa linha declara uma nova classe chamada Game. As chaves {} delimitam o corpo da classe em que as propriedades e métodos são definidos.
- **var contador = 0:** dentro da classe é declarada uma propriedade chamada contador do tipo Int (número inteiro) e inicializada com o valor 0. Essa propriedade será utilizada para contar as iterações do jogo.
- **fun roda{}:** essa linha declara uma função (método) na classe chamada roda. Essa função não recebe nenhum parâmetro nem retorna qualquer valor.
- **while(contador<1000){}**: na função roda existe um loop while. Ele continua executando enquanto a condição contador<1000 for verdadeira. Em outras palavras, o código dentro do loop será executado repetidamente até que o valor de contador seja igual ou maior que 1000.
- **contador++:** no loop, a cada iteração, o valor da variável contador é incrementado em 1. Isso significa que a cada vez que o loop é executado, o valor de contador fica mais próximo de 1000.

Antes de nos aprofundarmos no desenvolvimento do nosso jogo, vamos observar os fundamentos da programação de aplicativos com Jetpack Compose e Kotlin. Criaremos um componente composable chamado JogoCobrinha na MainActivity para exibir o valor do contador. Essa tela será a base para as futuras funcionalidades do nosso jogo.

A função JogoCobrinha é um Composable no Jetpack Compose que representará um simples jogo da cobrinha. Dentro da função, a variável game armazenará uma referência a um Objeto da classe Game criada na memória, e um texto é inicializado como uma string vazia. A função roda do objeto game é chamada para atualizar o estado do jogo. Em seguida, a variável texto é atualizada para exibir uma mensagem que inclui o valor do contador do jogo, indicando o progresso ou a pontuação atual. Finalmente, o texto é exibido na tela usando o Composable Text. Este exemplo demonstra como integrar lógica de jogo e interface de usuário em uma aplicação Compose.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
@Composable
fun JogoCobrinha() {

    // variáveis
    val game = Game()
    var texto = ""

    // controle

    game.roda()

    // tela
    texto="Jogo da Cobrinha ${game.contador} "
    Text(text = texto)
}
```

Analizando elemento a elemento:

- **@Composable fun JogoCobrinha()**: essa linha declara uma função composta chamada JogoCobrinha. Funções compostas são a base da construção de interfaces de usuário no Jetpack Compose, permitindo criar elementos visuais de forma declarativa.
- **val game = Game()**: uma instância da classe Game é criada e armazenada na variável game. Essa classe, não mostrada no código, provavelmente contém a lógica principal do jogo, como o movimento da cobra, a geração de comida e o cálculo da pontuação.
- **var texto = ""**: uma variável texto é declarada para armazenar uma string que será exibida na tela. Inicialmente, ela está vazia.
- **game.roda()**: essa linha chama o método roda() da classe Game, que provavelmente inicia o loop principal do jogo, atualizando o seu estado a cada frame.
- **texto = "Jogo da Cobrinha \${game.contador}"**: o valor da variável texto é atualizado para incluir uma mensagem com o texto "Jogo da Cobrinha" e o valor do atributo contador da classe Game, que provavelmente representa a pontuação do jogador.
- **Text(text = texto)**: por fim, o componente Text é usado para exibir o texto armazenado na variável texto na tela. Esse componente é responsável por renderizar o texto na interface do usuário.

Altere a função Greeting para chamar a nossa função jogoCobrinha:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        JogoCobrinha()
    }
}
```

Execute o emulador e observe que na tela somente apareceu o número final: 1000.

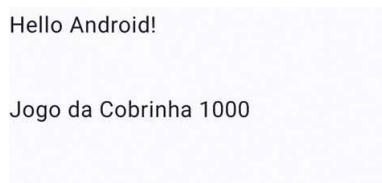


Figura 348

Isso ocorre porque o processamento é realizado em uma única linha de execução (thread). Primeiramente, o programa chama a função roda, que é executada até que o contador atinja o valor de mil. Em seguida, o texto é renderizado na tela, exibindo o valor mil.

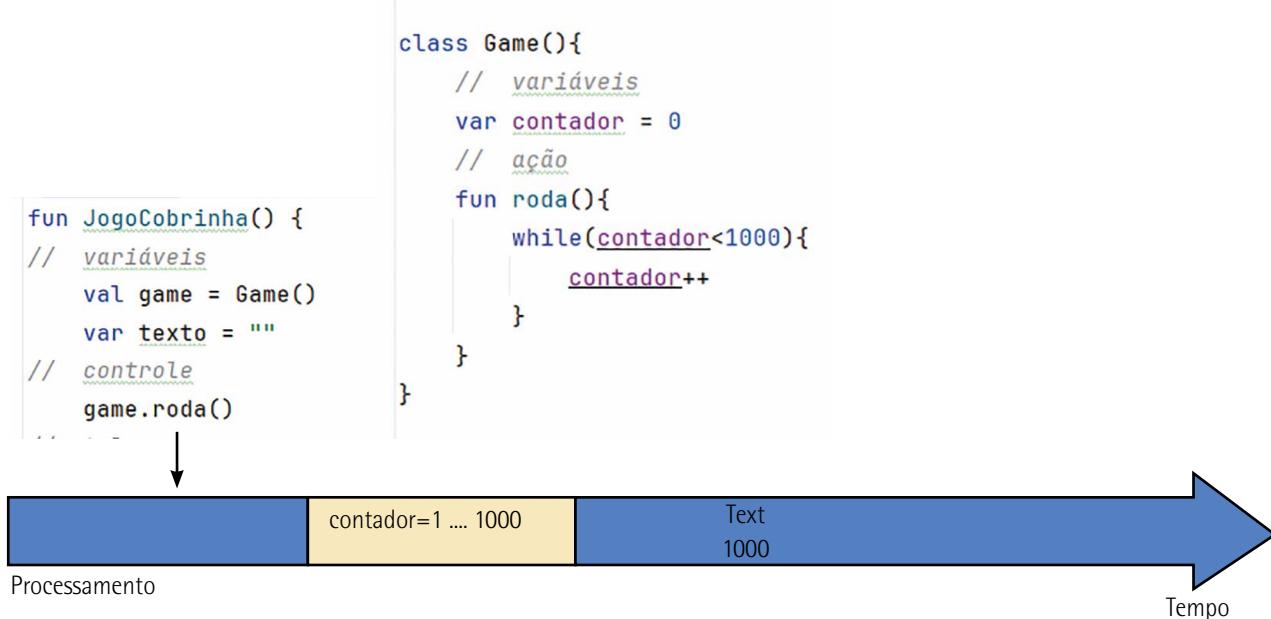


Figura 349 – O processamento acontece em uma linha de execução, sequencialmente

Se o jogo for montado desta forma, a lógica e a exibição não estarão funcionando em sincronia. A tela somente será atualizada após o final da função roda. Para contornar este problema, temos uma técnica que se chama corrotinas.

Corrotinas

Corrotinas são estruturas de programação que permitem a execução de tarefas de forma concorrente em um único fluxo de controle. Elas são especialmente úteis para simplificar o código assíncrono, tornando-o mais legível e fácil de manter.

Em termos simples, corrotinas permitem que você pause e retome a execução de funções em pontos específicos, sem bloquear o programa principal que está em execução. Isso é particularmente útil em

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

aplicações que precisam realizar operações demoradas, como atualização de tela, chamadas de rede ou acesso a banco de dados, sem travar a interface do usuário. Possibilitam ainda que várias rotinas funcionem simultaneamente.

Constam a seguir alguns de seus conceitos-chave:

- **Lançamento de corrotinas:** você pode lançar uma corrotina usando funções como launch ou async.
- **Suspensão:** funções de suspensão (suspend functions) podem ser pausadas e retomadas em pontos específicos.
- **Escopos de corrotinas:** escopos (CoroutineScope) definem o contexto no qual as corrotinas são executadas, ajudando a gerenciar seu ciclo de vida.

Vamos implementar a corrotina no nosso exemplo.

- Na função JogoCobrinha() substitua a instanciação da classe Game (val game = Game()) por:

```
val game = remember{Game()}
```

É criada uma instância da classe Game e armazenada na variável game. A função remember é usada para garantir que a instância Game seja preservada durante recomposições, ou seja, quando a interface de usuário é redesenhada.

- Envolva a chamada função roda() com o LaunchedEffect.

```
// controle
LaunchedEffect(game) {
    game.roda()
}
```

Aqui, quando a composição é iniciada, este bloco de código é executado, chamando a função roda do objeto game. O LaunchedEffect é utilizado para lançar corrotinas no escopo da composição, iniciando uma corrotina assim que o valor de game muda, como na criação. Isso aciona a função roda() da classe Game, que é responsável por executar a lógica do jogo.

Nosso código na MainActivity.kt neste momento deve estar da seguinte maneira:

```
@Composable
fun JogoCobrinha() {
// variáveis
    val game = remember{Game()}
    var texto = ""

// controle
    LaunchedEffect(game) {
        game.roda()
```

Unidade III

```
}

// tela
texto="Jogo da Cobrinha ${game.contador} "
Text(text =texto)
}
```

Ao executá-lo, aparentemente o programa não funcionou, pois o contador permaneceu no zero.

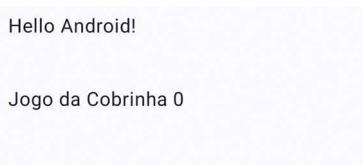


Figura 350

A corrotina foi criada quando a função roda foi chamada no LaunchedEffect. Nesse momento, temos duas execuções ocorrendo simultaneamente, mas o valor do texto não é atualizado devido a problemas de sincronização entre os processamentos, permanecendo em seu estado inicial.

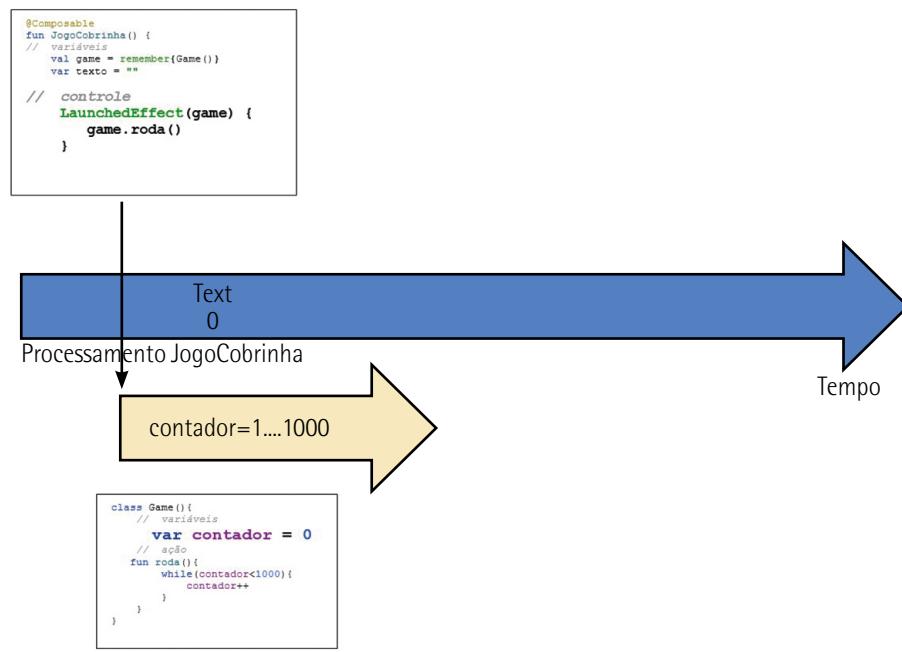


Figura 351 – Uma nova linha de execução, mas não há retorno de valor

Precisamos que a variável atualize automaticamente a interface do usuário. Neste caso, utiliza-se o estado mutável ou mutableStateOf. Ele cria um estado observável para contador. Isso significa que o Compose monitora constantemente o valor de contador e, quando ele é modificado, o Compose dispara uma recomposição, atualizando a interface do usuário de acordo com o novo valor.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Para tornar a variável contador mutável, vamos modificar a classe Game. No arquivo Logica.kt, altere a declaração da variável contador para:

```
// variáveis  
var contador by mutableStateOf(0)
```

A alteração da declaração var contador by mutableStateOf(0) agora criará uma variável contador como um estado mutável com valor inicial 0. O uso de mutableStateOf permite que o Compose observe mudanças no valor de contador e atualize automaticamente a interface do usuário. Isso é útil para acompanhar o valor da variável, tornando o contador observável e permitindo que a UI seja atualizada quando ele mudar.



Às vezes, a importação das bibliotecas não é feita automaticamente, o que pode causar erros em instruções relacionadas a essas bibliotecas. Por exemplo, isso ocorre após a última alteração:

```
class Game(){  
    // variáveis  
    var contador by mutableStateOf(0)  
    private set
```

Figura 352

Neste caso, verifique se as seguintes importações estão presentes:

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

Se não estiverem, adicione-as manualmente.

Ao executar, temos como resultado uma tela mostrando o valor 1000.

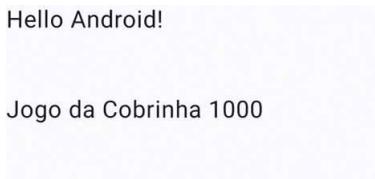


Figura 353

Desta vez, o processamento da thread da função rodou enviou todas as atualizações para a interface do usuário. No entanto, o laço da instrução while executou tão rapidamente que a taxa de atualização da UI foi inferior à taxa de mudança do estado, resultando na exibição apenas do estado final.

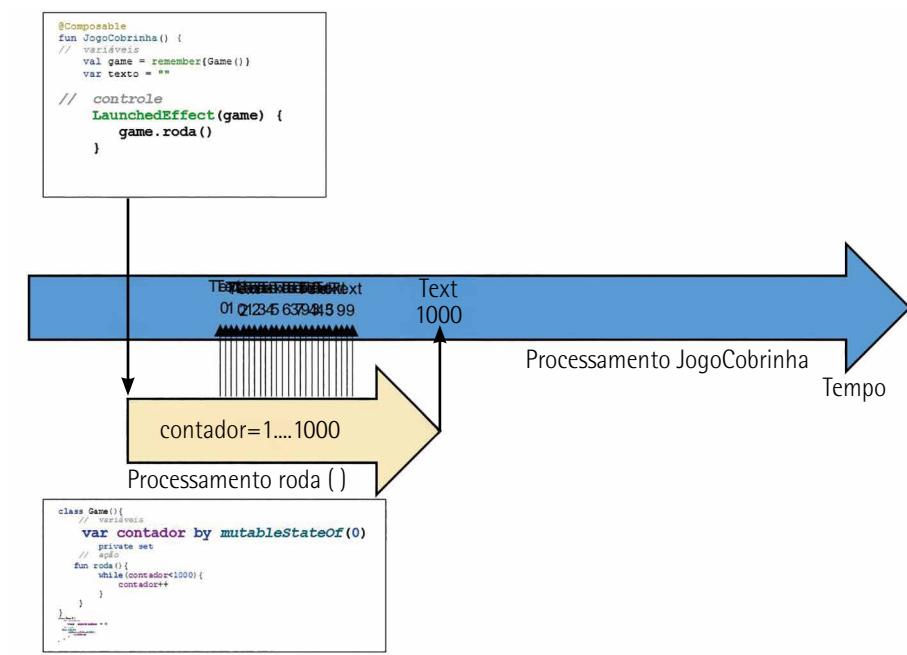


Figura 354 – O mutableState envia as atualizações muito rapidamente sobrecarregando a UI.

Como vimos, uma das características de uma corوتina é a possibilidade da suspensão de um processamento. Aqui utilizaremos a função `delay()`, ela é usada em corوتinas para pausar a execução por um período sem bloquear a thread em que a corوتina está sendo executada.

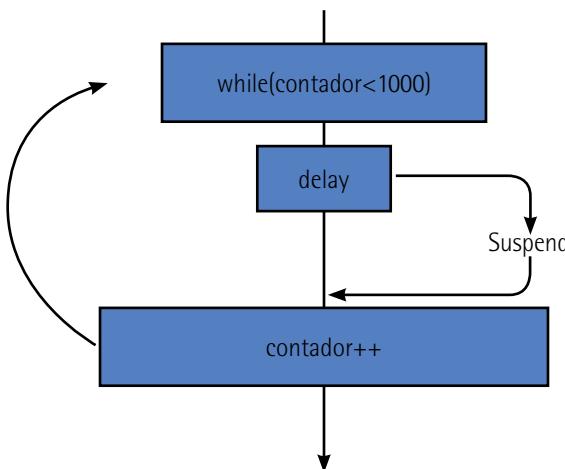


Figura 355 – A função `delay` causa a suspensão de uma corوتina sem afetar as outras corوتinas

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Vamos alterar a função roda(). Para que o delay funcione, é necessário informar ao sistema indicando que a função pode ser pausada e retomada.

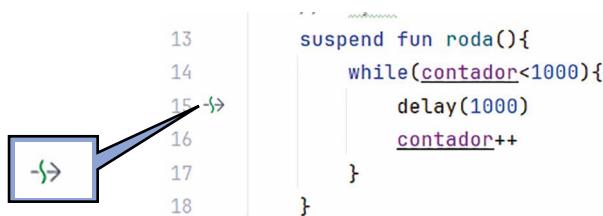
```
suspend fun roda() {
```

Incluiremos na função roda a instrução de delay().

```
suspend fun roda() {  
    while(contador<1000) {  
        delay(1000)  
        contador++  
    }  
}
```



Note que um símbolo foi acrescentado na tela do editor, indicando que existe um ponto de suspensão.



```
13  
14  
15 ->  
16  
17  
18
```

```
suspend fun roda(){  
    while(contador<1000){  
        delay(1000)  
        contador++  
    }  
}
```

Figura 356

Execute e observe que agora o valor avança de um em um a cada segundo.

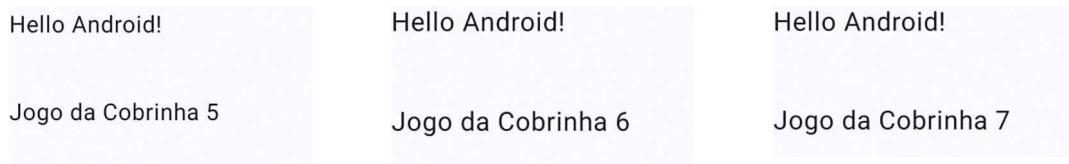


Figura 357

Unidade III

Ao adicionar o delay, o laço itera a cada segundo, permitindo tempo suficiente para que a atualização da variável mutável contador seja refletida na interface do usuário.

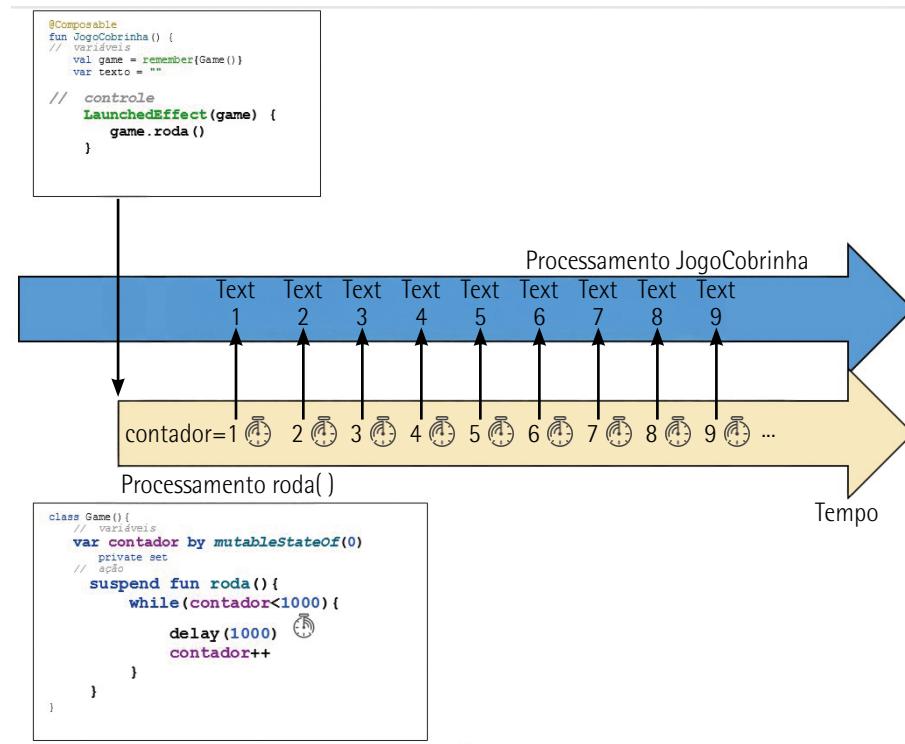


Figura 358 – Cada atualização da variável contador pode ser refletida na UI

Para evitar a execução desnecessária de corrotinas e garantir que elas sejam iniciadas apenas quando certas condições são atendidas, um if envolvendo um LaunchedEffect é usado para condicionar a execução da coroutines a uma determinada condição. Isso significa que a coroutines somente será lançada se a condição especificada no if for verdadeira.

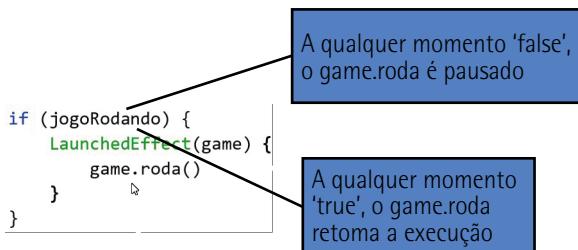


Figura 359

Para controlar a execução do programa, criaremos um botão que pausará o contador.

- Adicionaremos uma nova variável chamada `jogoRodando`. A ideia é que o contador apenas comece a funcionar quando essa variável for definida como `true`, assim iniciaremos com `false`.

```
var jogoRodando by remember { mutableStateOf(false) }
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- Como vimos, precisamos envolver o LaunchedEffect com um if para pausar ou retomar a execução.

```
if (jogoRodando) {
    LaunchedEffect(game) {
        game.roda()
    }
}

fun JogoCobrinha() {

    // variáveis
    var jogoRodando by remember { mutableStateOf(false) }
    val game = remember{Game()}
    var texto = ""

    // controle
    if (jogoRodando) {
        LaunchedEffect(game) {
            game.roda()
        }
    }

    // tela
    texto="Jogo da Cobrinha ${game.contador}"
    Text(text =texto)
}
```

Como no Jetpack Compose a programação pode ser feita com a interface, vamos adicionar um botão que, ao ser clicado, altere o valor de jogoRodando para false se estiver true e vice-versa. Para isso, utilizaremos a operação lógica de negação (!). A legenda do botão automaticamente é trocada, alternando entre Start e Pause.

```
Button(
    onClick = { jogoRodando = !jogoRodando },
    modifier = Modifier.fillMaxWidth(),
) {
    Text(if (jogoRodando) "Pause" else "Start")
}
```

Como estamos desenvolvendo nosso jogo e discutindo novos conceitos, vamos construindo nossa tela. Para garantir que o jogo se adapte a qualquer formato de tela dos dispositivos móveis, envolveremos a parte visual com BoxWithConstraints. Este componente desempenha um papel crucial na criação de layouts flexíveis e responsivos no Jetpack Compose.

Ao envolver a interface do usuário (UI) com BoxWithConstraints, você permite que ela se adapte dinamicamente a diferentes tamanhos de tela e orientações, garantindo uma experiência consistente em diversos dispositivos. O BoxWithConstraints fornece informações precisas sobre a largura e altura disponíveis para a UI, possibilitando a criação de layouts que se ajustam de forma inteligente ao espaço disponível. Nele já vamos deixar uma margem de 16.dp.

Note que ocorreu um erro.

```
BoxWithConstraints (Modifier.padding(16.dp)) {
    texto = "Jogo da Cobrinha ${game.contador} "
    Text(text = texto)
    Button(
        onClick = { jogoRodando = !jogoRodando },
        modifier = Modifier.fillMaxWidth(),
    ) {
        Text(if (jogoRodando) "Pause" else "Start")
    }
}
```

Figura 360

Com a finalidade de corrigir o problema, utiliza-se a anotação `@SuppressLint("UnusedBoxWithConstraintsScope")` para suprimir um aviso do lint do Android Studio, uma ferramenta de análise estática que verifica o código em busca de possíveis problemas e sugere melhorias. Essa anotação é necessária ao usar BoxWithConstraints para evitar avisos do compilador sobre o escopo não utilizado. Isso ocorre porque BoxWithConstraints fornece um escopo que permite acessar as dimensões mínimas e máximas disponíveis para o layout, mas, em alguns casos, você não precisa usar essas informações diretamente no código. Para evitar esses avisos, a anotação é adicionada.

Portanto, antes da anotação `@Composable`, coloque:

```
@SuppressLint("UnusedBoxWithConstraintsScope")
```

O código fica:

```
@SuppressLint("UnusedBoxWithConstraintsScope")
@Composable
fun JogoCobrinha() {
    // variáveis
    var jogoRodando by remember { mutableStateOf(false) }
    val game = remember{Game()}
    var texto = ""

    // controle
    if (jogoRodando) {
        LaunchedEffect(game) {
            game.roda()
        }
    }
    // tela
    BoxWithConstraints() {
        texto = "Jogo da Cobrinha ${game.contador} "
        Text(text = texto)
        Button(
            onClick = { jogoRodando = !jogoRodando },
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
        modifier = Modifier.fillMaxWidth(),
    ) {
    Text(if (jogoRodando) "Pause" else "Start")
}
}
}
```

No nosso preview, temos:



Figura 361

Se observamos direito, o Botão Start e o Texto "Jogo da Cobrinha" estão sobrepostos. Para corrigir, vamos envolver os dois componentes com uma Column, organizando-os verticalmente e já definindo que ela terá um fundo cinza claro, com os seus elementos posicionados a partir do topo, um após o outro e centralizados horizontalmente.

```
BoxWithConstraints() {
Column(
    modifier = Modifier.background(Color.LightGray),
    verticalArrangement = Arrangement.Top,
    horizontalAlignment = Alignment.CenterHorizontally
) {
    texto = "Jogo da Cobrinha ${game.contador} "
    Text(text = texto)
    Button(
        onClick = { jogoRodando = !jogoRodando },
        modifier = Modifier.fillMaxWidth(),
    ) {
        Text(if (jogoRodando) "Pause" else "Start")
    }
}
}
```

Agora a nossa tela está correta, vamos verificar no Preview:



Figura 362

Podemos testar e verificar que a contagem pode ser interrompida clicando no botão Pause.



Saiba mais

As corrotinas são importantíssimas, pois permitem que várias ações ocorram simultaneamente. Caso o aluno queira se aprofundar, consulte os sites oficiais do Android Studio e da linguagem Kotlin.

DEVELOPERS. *Introdução a corrotinas no Android Studio*. 21 maio 2024. Disponível em: <https://shre.ink/MRBq>. Acesso em: 6 mar. 2025.

KOTLIN. *Coroutines*. 18 out. 2022. Disponível em: <https://shre.ink/MRBq>. Acesso em: 6 mar. 2025.

7.2.2 Desenhar o objeto principal

Nosso jogo se passa em um tabuleiro de 16x16 quadradinhos, em que a cobrinha se movimenta e os alimentos aparecem.

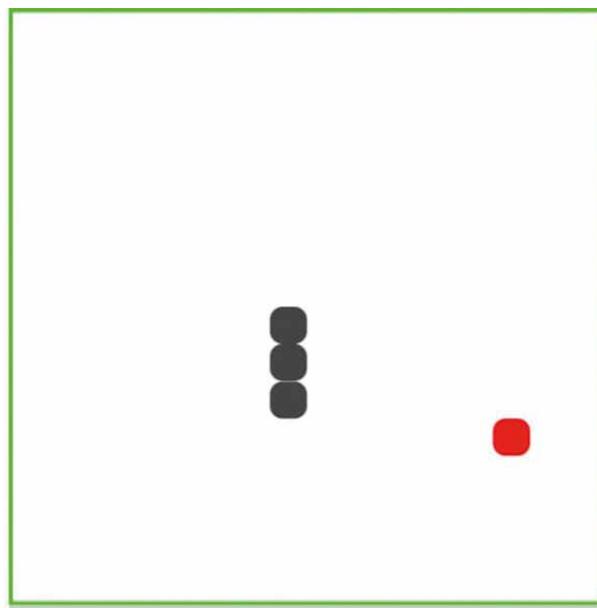


Figura 363

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Como o Compose permite, vamos inserir o quadro em nossa activity.

```
BoxWithConstraints(){  
    Column() {  
        texto = "Jogo da Cobrinha ${game.contador}"  
        Text(text = texto)  
        Button(  
            onClick = { jogoRodando = !jogoRodando }  
            modifier = Modifier.fillMaxWidth(),  
        ) {  
            Text(if (jogoRodando) "Pause" else "Start")  
        }  
    }  
}
```

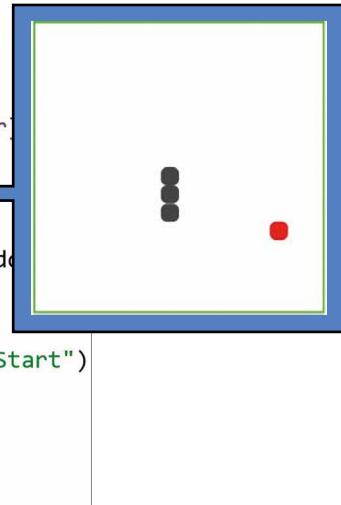


Figura 364

Inicialmente, nosso quadro será novamente envolvido com o BoxWithConstraints, pois ele nos fornecerá, conforme veremos mais tarde, valores úteis para a flexibilidade do jogo.

- Na função JogoCobrinha, entre o Text e o Button acrescente um BoxWithConstraints.

```
    texto = "Jogo da Cobrinha ${game.contador}"  
    Text(text = texto)
```

```
BoxWithConstraints() {}
```

```
Button(
```

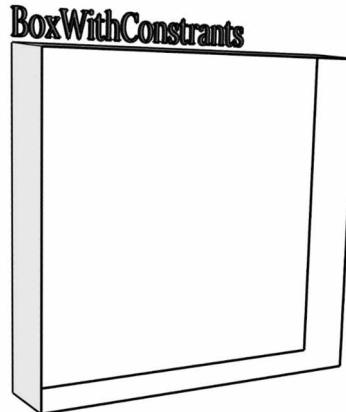


Figura 365

Neste BoxWithConstraints insira um Box quadrado, que é basicamente um contêiner quadrado com um fundo branco e uma borda verde, tendo o tamanho largura disponível. Modifique-o para que a altura e a largura sejam maxWidth, formando um quadrado. O maxWidth é uma propriedade que fornece a largura máxima disponível para o conteúdo no BoxWithConstraints. Isso é útil pois permite criar layouts adaptáveis em que você pode ajustar o design com base no espaço disponível na tela e ainda obter valores flexíveis baseados nas diferentes dimensões das telas dos dispositivos. Deixe também a borda verde e o fundo branco:

```
BoxWithConstraints() {  
    Box(Modifier  
        .size(maxWidth)  
        .background(Color.White)  
        .border(2.dp, Color.Green)) {  
    }  
}
```

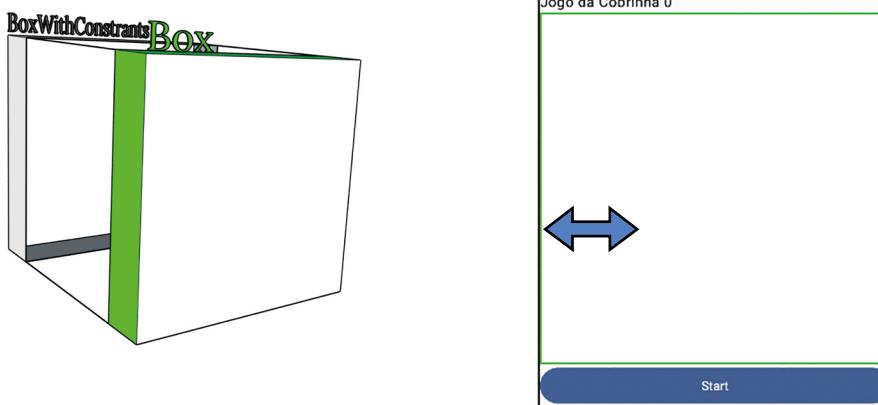


Figura 366

No Box que forma o quadro do nosso jogo, criaremos um box quadrado, que é deslocado 30 dp para a direita e 30 dp para baixo a partir de sua posição original. Vamos deixá-lo com um tamanho fixo de 30 dp por 30 dp e um fundo cinza escuro. Além disso, o fundo do Box é estilizado com uma forma pequena definida por Shapes().small, que geralmente aplica cantos arredondados ou outras características de forma. Este Box será a cabeça da nossa cobrinha.

```
Box(  
    Modifier  
        .size(maxWidth)  
        .background(Color.White)  
        .border(2.dp, Color.Green)  
) {  
}  
  
Box(  
    Modifier  
        .size(30.dp)  
        .background(shapes().small)  
        .border(2.dp, Color.Black)
```

```

    modifier = Modifier
        .offset(x = 30.dp, y = 30.dp)
        .size(30.dp)
        .background(
            Color.DarkGray,
            Shapes().small
        )
    )
}

```

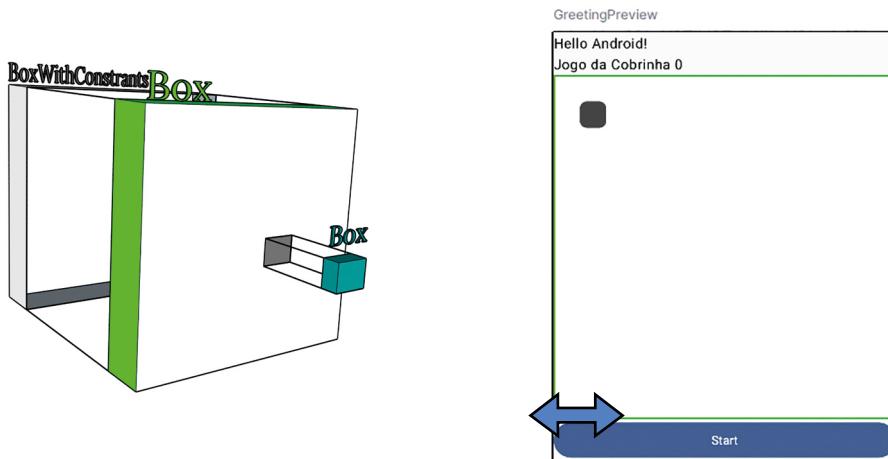


Figura 367

Observação

No Jetpack Compose, as unidades de medida dp e sp são fundamentais para criar interfaces de usuário consistentes e acessíveis:

- **dp (density-independent pixels)**: são pixels independentes de densidade. Um dp é uma unidade de medida que se adapta à densidade da tela, garantindo que os elementos da interface mantenham o mesmo tamanho físico em diferentes dispositivos. Isso ajuda a criar layouts que se ajustam de forma consistente em telas com diferentes resoluções.
- **sp (scale-independent pixels)**: são pixels independentes de escala. Além de serem independentes de densidade como os dp, os sp levam em consideração as preferências de escala de texto do usuário. Isso significa que o tamanho do texto ajustado com sp será escalado de acordo com as configurações de acessibilidade do dispositivo, garantindo que o texto seja legível para todos os usuários.

Além de dp e sp, existem outras unidades de medida que podem ser usadas no desenvolvimento Android:

- **px (pixels)**: corresponde aos pixels reais na tela. É uma unidade absoluta e não se adapta a diferentes densidades de tela.
- **in (polegadas)**: baseada no tamanho físico da tela. 1 polegada = 2,54 centímetros.
- **mm (milímetros)**: também baseada no tamanho físico da tela.
- **pt (pontos)**: 1/72 de polegada, usada principalmente em tipografia.

Essas unidades adicionais são menos comuns, mas podem ser úteis em situações específicas em que medidas absolutas são necessárias.

Implementação da animação

Neste momento, a tela está pronta para implementarmos a animação. Embora não nos aprofundemos em técnicas avançadas neste curso introdutório, os princípios fundamentais são os mesmos. A mecânica de mover a cabeça da cobrinha no nosso jogo é similar aos princípios da física. Utilizamos o modificador offset para posicionar o Box (que representa a cabeça) em coordenadas específicas no contêiner. Ao variar o valor do offset ao longo do tempo em intervalos dados pelo delay(), simulamos o movimento. Essa variação do offset pode ser comparada à equação horária da física, na qual a posição de um objeto muda em função do tempo.

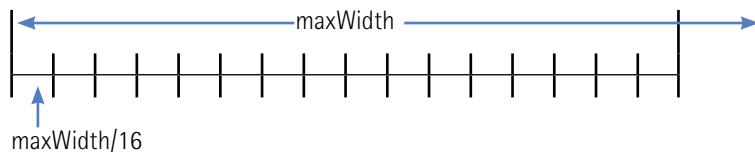
No nosso aplicativo, vamos alterar o offset do Box cinza para:

```
Box(  
    modifier = Modifier  
        .offset(x = game.contador.dp, y = 30.dp)  
        .size(30.dp)  
        .background(  
            Color.DarkGray,  
            Shapes().small  
        )  
    )
```

Para ativar o emulador, clique em Start. Observe que o ponto se move lentamente.

Como queremos que nosso campo seja 16 x 16, dividimos a largura em pixels do nosso dispositivo por 16 para definir a dimensão padrão. Como os dispositivos têm tamanhos diferentes, utilizamos o BoxWithConstraints, que retorna a propriedade maxWidth com o tamanho máximo do dispositivo. Assim, cada dimensão do ponto será 1/16 da largura máxima.

Quadro 5



Criaremos uma variável chamada dimensãoPonto, que armazenará 1/16 do valor da largura da tela do dispositivo e deixá-la logo abaixo do início do bloco do primeiro BoxWithConstraints.

```
// tela
BoxWithConstraints () {
    val dimensaoPonto = maxWidth/16
    Column () {
```

Ajustaremos o tamanho do Box para que ele seja um quadrado de tamanho dimensoPonto e novamente alterar o offset do Box:

```
.offset(x = dimensaoPonto * game.contador , y = dimensaoPonto * 7)
.size( dimensaoPonto )
```

Quando executarmos novamente no emulador, no sentido horizontal, o ponto avançará em múltiplos da dimensão do ponto e, na vertical por enquanto, estará na linha 7. No entanto, observe que o ponto continua avançando além do nosso quadro.

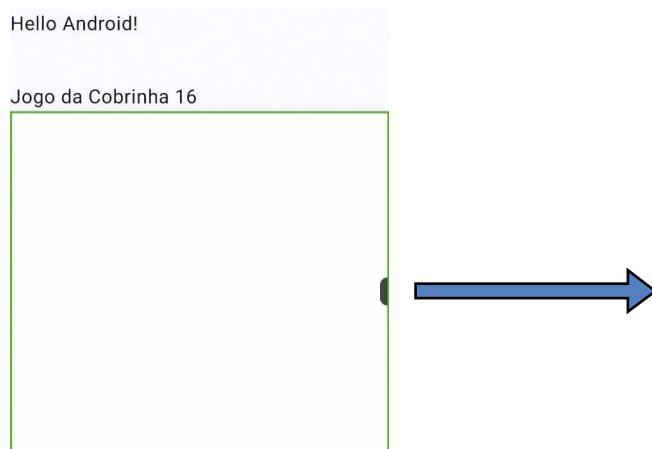
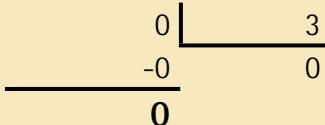
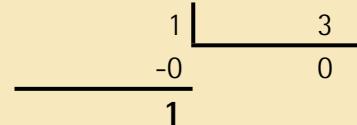
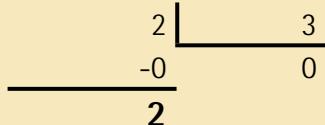
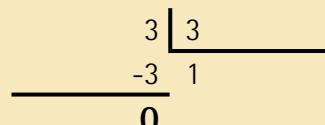
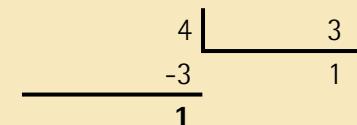
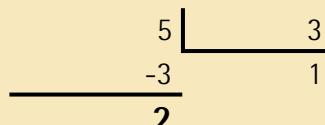
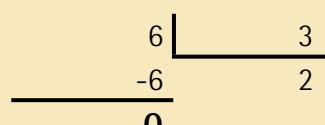
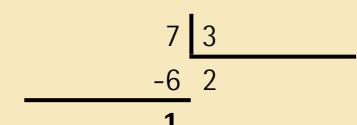


Figura 368

Para resolver o avanço indevido, utilizaremos a matemática a nosso favor. Existe uma operação matemática extremamente útil conhecida como módulo, representada no Kotlin pelo símbolo %, que calcula o resto da divisão. Veremos o comportamento do Mod (quadro 6).

Quadro 6

0 % 3 resulta em 0 Porque: 	1 % 3 resulta em 1 Porque: 	2 % 3 resulta em 2 Porque: 
3 % 3 resulta em 0 Porque: 	4 % 3 resulta em 1 Porque: 	5 % 3 resulta em 2 Porque: 
6 % 3 resulta em 0 Porque: 	7 % 3 resulta em 1 Porque: 	...

Na operação Mod, observe que, ao dividir um número, quando ele alcança seu múltiplo, o resto da divisão volta a zero. A cada unidade, o resto aumenta de um em um até alcançar o próximo múltiplo, voltando a zero novamente. Assim, vamos alterar na função roda para que a posição volte a zero ao chegar no 16, que é o tamanho do nosso quadro. Precisamos reorganizar os comandos para ficarem mais coesos sem perder a sequência lógica e aumentar a velocidade diminuindo o tempo de espera dos 500 milissegundos.

```
class Game() {
    // variáveis
    var contador by mutableStateOf(0)
    private set
    // ação
    suspend fun roda() {
        while(contador<1000) {
            contador++
            contador=contador % 16
            delay(500)
        }
    }
}
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Desta forma, o nosso ponto retorna ao início quando alcança o limite esquerdo.

Vamos testar com o passo negativo para ver se o ponto ao alcançar a parede esquerda surge na parede direita. Alterando o contador para:

```
contador--
```

Neste ponto, vemos que o quadrado sai imediatamente da tela, desaparecendo do quadro. Para corrigir, vamos somar 16 ao nosso cálculo, assim o valor do contador sempre estará entre zero e quinze.

```
while(contador<1000) {
    contador++
    contador=(contador + 16) % 16
    delay(500)
}
```

Veja que agora o movimento funciona em ambas as direções. Com o programa funcionando, para continuar o desenvolvimento e termos uma melhor compreensão, renomearemos a variável contador.

Com o objetivo de melhorar a compreensão e a continuidade na programação, mudaremos o nome da variável contador para posicaoX na lógica. Além disso, alteraremos a condição para true, pois o contador nunca chegaria a 1000.

```
class Game(){
    // variáveis
    var posicaoX by mutableStateOf(0)
    // ação
    suspend fun roda(){
        while(true){
            posicaoX++
            posicaoX=(posicaoX + 16) % 16
            delay(500)
        }
    }
}
```

Figura 369

Também faremos a troca nos nomes no MainActivity.kt.

```
// tela
BoxWithConstraints(){
    val dimensaoPonto = maxWidth/16
    Column() {
        ! texto = "Jogo da Cobrinha ${game.posicaoX} "
        Text(text = texto)
        //Quadro do jogo
        BoxWithConstraints(){
            Box(
                Modifier
                    .size(maxWidth)
                    .background(Color.White)
                    .border(2.dp, Color.Green))
            {
                Box(
                    modifier = Modifier
                        .offset(x = dimensaoPonto * game.posicaoX, y = dimensaoPonto * 7)
                        .size(30.dp)
```

Figura 370

Antes de continuarmos para o próximo item, o nosso programa está assim:

MainActivity.kt

```
package .....

import android.annotation.SuppressLint
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.BoxWithConstraints
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.offset
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.material3.Button
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Shapes
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import com.example.cobrinhadese...
```

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

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

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

@SuppressWarnings("UnusedBoxWithConstraintsScope")
@Composable
fun JogoCobrinha() {
// variáveis
    var jogoRodando by remember { mutableStateOf(false) }
    val game = remember{Game()}
    var texto = ""

// controle
```

Unidade III

```
if (jogoRodando) {
    LaunchedEffect(game) {
        game.roda()
    }
}
// tela
BoxWithConstraints() {
    val dimensaoPonto = maxWidth/16
    Column() {
        texto = "Jogo da Cobrinha ${game.posicaoX} "
        Text(text = texto)
        //Quadro do jogo
        BoxWithConstraints() {
            Box(
                Modifier
                    .size(maxWidth)
                    .background(Color.White)
                    .border(2.dp, Color.Green))
            {
                Box(
                    modifier = Modifier
                        .offset(x = dimensaoPonto * game.posicaoX, y =
dimensaoPonto * 7)
                    .size(30.dp)
                    .background(
                        Color.DarkGray,
                        Shapes().small
                    )
                )
            }
        }
        //Liga e desliga o jogo
        Button(
            onClick = { jogoRodando = !jogoRodando },
            modifier = Modifier.fillMaxWidth(),
        ) {
            Text(if (jogoRodando) "Pause" else "Start")
        }
    }
}
```

Logica.kt

```
package com.example.cobrinhadesenvolvimento

import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import kotlinx.coroutines.delay

class Game() {
    // variáveis
```

```
var posicaoX by mutableStateOf(0)
    private set
// ação
suspend fun roda() {
    while(true){
        posicaoAtual++
        posicaoAtual=(posicaoX + 16) % 16
        delay(500)
    }
}
```

7.2.3 Mover o objeto através do comando do usuário

Atualmente estamos trabalhando apenas com a direção X, mantendo o valor de Y constante em 7. Para implementar a mudança de direção, precisaremos também considerar a direção Y. Portanto, será necessário criar uma variável similar à nossa variável posicaoX, chamada posicaoY.

```
var posicaoY by mutableStateOf(0)
```

Vamos acrescentar o processamento da posicaoY:

```
class Game(){
    // variáveis
    var posicaoX by mutableStateOf(0)
    var posicaoY by mutableStateOf(0)
    // ação
    suspend fun roda(){
        while(true){
            posicaoX++
            posicaoY++
            posicaoX=(posicaoX + 16) % 16
            posicaoY=(posicaoY + 16) % 16
            delay(500)
        }
    }
}
```

Figura 371

Unidade III

No Box móvel agora substituíremos o valor 7 do offset de Y pelo game.posicaoY.

```
Box(  
    modifier = Modifier  
        .offset(x = dimensaoPonto * game.posicaoX, y = dimensaoPonto * game.posicaoY)  
        .size(30.dp)  
        .background(  
            Color.DarkGray,  
            Shapes().small  
        )  
)
```

Figura 372

Observe que, ao executar o emulador, o ponto se move na diagonal.

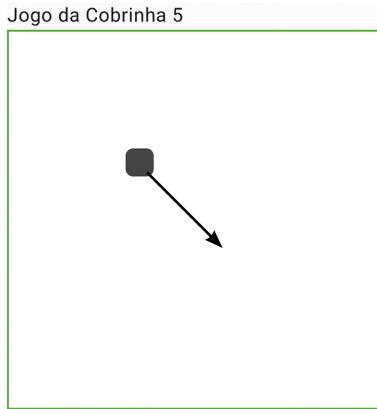


Figura 373

É necessário ter controle sobre a direção em que o ponto avançará. Seria interessante possuir uma variável que armazenasse dois valores: um para a direção horizontal e outro para a vertical. O Kotlin oferece uma solução prática para isso: a classe Pair. Um Pair nos permite agrupar dois valores, como as coordenadas (x, y) da posição. Isso é particularmente útil em situações nas quais precisamos retornar dois valores de uma função ou armazenar pares de dados sem a necessidade de criar uma classe específica para isso.

Na classe Game, crie a variável direcaoAtual do tipo Pair com o valor inicial (1,0).

```
var direcaoAtual by mutableStateOf(Pair(1,0))
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

A classe Pair fornece dois métodos, first e second, que nos fornecem os valores da primeira e da segunda posição do par, respectivamente. Utilizaremos esses métodos na função roda, substituindo o incremento de X, que era `++`, por `direcaoAtual.first`, e o incremento de Y por `direcaoAtual.second`.

```
suspend fun roda(){
    while(true){
        posicaoX++
        posicaoY++
        posicaoX=(posicaoX + 16) % 16
        posicaoY=(posicaoY + 16) % 16
        delay(500)
    }
}

suspend fun roda(){
    while(true){
        posicaoX=posicaoX + direcaoAtual.first
        posicaoY=posicaoY + direcaoAtual.second
        posicaoX=(posicaoX + 16) % 16
        posicaoY=(posicaoY + 16) % 16
        delay(500)
    }
}
```

Figura 374

Ao dar Start no emulador, o ponto movimenta-se para a direita. Altere os valores da variável `direcaoAtual` para $(0,1)$, $(-1,0)$ e $(0, -1)$. No emulador, o ponto movimenta-se para baixo, para esquerda e para cima, respectivamente.

Para implementar o controle do jogo, utilizaremos botões que definirão a direção do movimento.

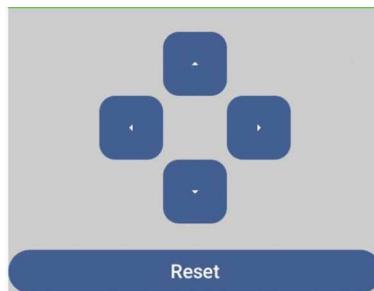


Figura 375

Os botões estarão entre o quadro do jogo e o botão de start Pause.

Unidade III

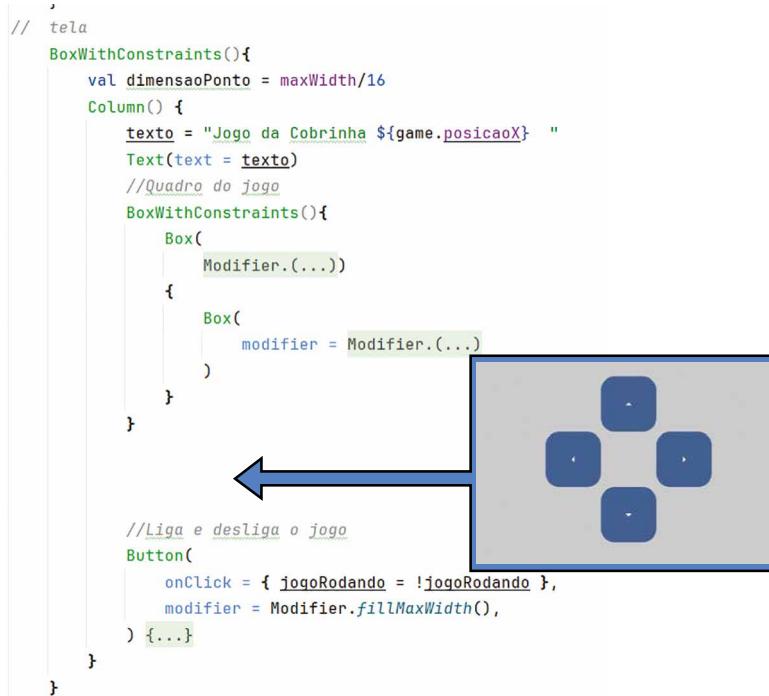


Figura 376

Inicialmente, criaremos uma variável imutável para definir um tamanho padrão para o botão. Isso é útil porque, se eventualmente quisermos alterar o tamanho, não será necessário editar os botões individualmente.

```
// tela
BoxWithConstraints() {
    val dimensaoPonto = maxWidth/16
    val tamBotao = Modifier.size(64.dp)
    Column() {
```

Usaremos o layout em coluna, pois os elementos estarão dispostos verticalmente de cima para baixo. Os elementos-filhos estarão alinhados horizontalmente ao centro dentro da coluna. A coluna terá um preenchimento (padding) de 24 dp.

```
//botões de direção
Column(horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier.padding(24.dp)) {
```



Figura 377

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Dentro, colocaremos um Botão, a ação ao clicar será { game.direcaoAtual = Pair(0, -1) }. Quando o botão é clicado, ele define a variável direcaoAtual da instância de Game para Pair(0, -1), indicando à lógica que a direção do movimento será para cima. O tamanho será o padrão definido pela variável tamBotao e o seu formato um quadrado com cantos arredondados. No bloco de código, por enquanto, será um texto escrito cima.

```
//botões de direção
Column(horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier.padding(24.dp)) {
    Button(
        onClick = { game.direcaoAtual = Pair(0, -1) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Text("Cima")
    }
}
```

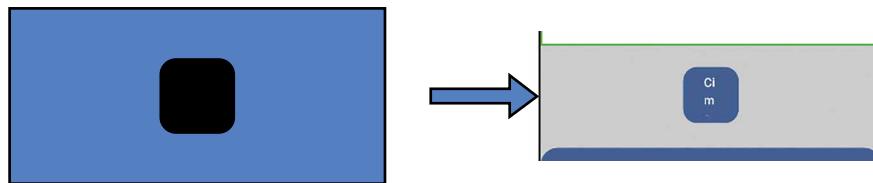


Figura 378

Abaixo do primeiro botão organizaremos os botões em uma Row, pois os botões para esquerda e para direita estarão dispostos lado a lado.

```
//botões de direção
Column(horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier.padding(24.dp)) {
    Button(
        onClick = { game.direcaoAtual = Pair(0, -1) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Text("Cima")
    }
}
Row {
```

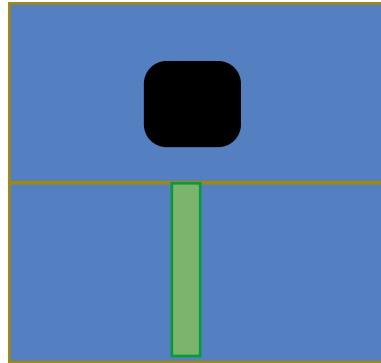


Figura 379

Dentro do Row, coloque dois botões com o mesmo design do primeiro botão, sendo o da esquerda acionando game.diracaoAtual(Pair(-1,0) e o da direita acionando direcaoAtual(Pair(1,0). Entre eles coloque um espaçador com o tamanho de um botão.

```
Row{
    Button(
        onClick = { game.diracaoAtual = Pair(-1, 0) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Text("Esquerda")
    }
    Spacer(modifier = tamBotao)
    Button(
        onClick = { game.diracaoAtual = Pair(1, 0) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Text("Direita")
    }
}
```

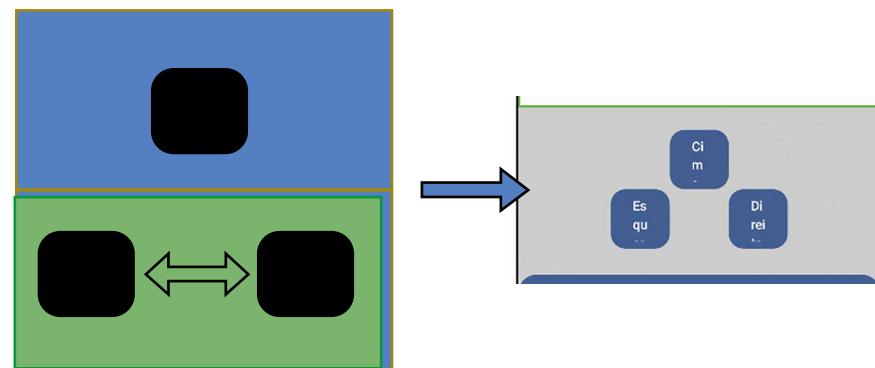


Figura 380

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Termine a criação dos botões colocando um botão abaixo da Row, em que o clique gera a ação de enviar o valor de game.direcaoAtual = Pair(0, 1).

```
Button(  
    onClick = { game.direcaoAtual=Pair(0,1) },  
    modifier = tamBotao,  
    shape = RoundedCornerShape(16.dp)  
) {  
    Text("Baixo")  
}
```

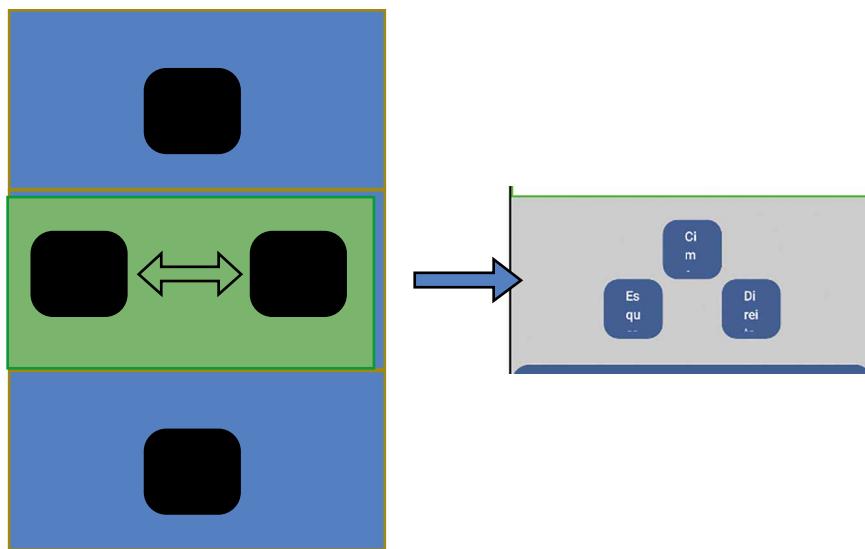


Figura 381

Colocaremos todo o grupo dos botões em uma coluna em que o botão para cima ocupará uma linha, enquanto os botões para os lados estarão alinhados lado a lado em outra linha, tendo abaixo o botão para baixo.

```
Column(horizontalAlignment = Alignment.CenterHorizontally,  
    modifier = Modifier.padding(24.dp)) {  
    Button(  
        onClick = { game.direcaoAtual = Pair(0, -1) },  
        modifier = tamBotaoMod,  
        shape = RoundedCornerShape(16.dp)  
) {  
        Text("Cima")  
    }  
    Row {  
        Button(  
            onClick = { game.direcaoAtual = Pair(-1, 0) },  
            modifier = tamBotao,  
            shape = RoundedCornerShape(16.dp)  
) {  
            Text("Esquerda")  
        }  
        Spacer(modifier = tamBotaoMod)
```

```
        Button(
            onClick = { game.direcaoAtual = Pair(1, 0) },
            modifier = tamBotao,
            shape = RoundedCornerShape(16.dp)
        ) {
            Text("Direita")
        }
    }
    Button(
        onClick = { game.direcaoAtual=Pair(0,1) },
        modifier = tamBotaoMod,
        shape = RoundedCornerShape(16.dp)
    ) {
        Text("Baixo")
    }
}
```

Neste momento, ao clicar start e nos botões, é possível dirigir o ponto.

Com o objetivo de otimizar a aparência visual da nossa interface, optamos por utilizar ícones no lugar de textos nos botões. O processo de inclusão de imagens é semelhante à adição de ícones. A biblioteca Icons do Compose disponibiliza uma ampla variedade de ícones padrão, facilitando a sua integração. No caso de imagens personalizadas, o componente Image do Compose permite a inserção de recursos gráficos a partir de diversas fontes, como arquivos locais ou endereços URL. Vamos criar os nossos ícones.

Clique com o botão esquerdo do seu mouse na pasta dos retouches (recursos), escolha New/Vector Asset.

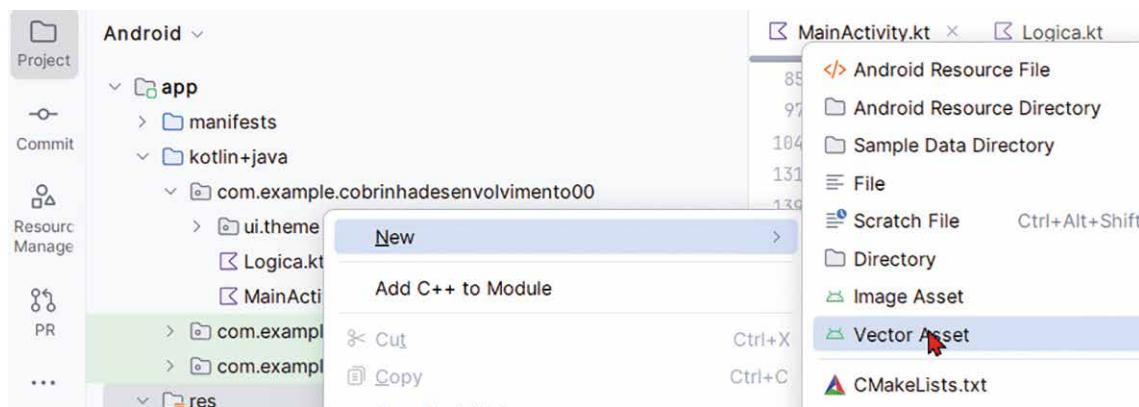


Figura 382

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Ao abrir a tela de Configure Vector Asset, clique em Clip art.

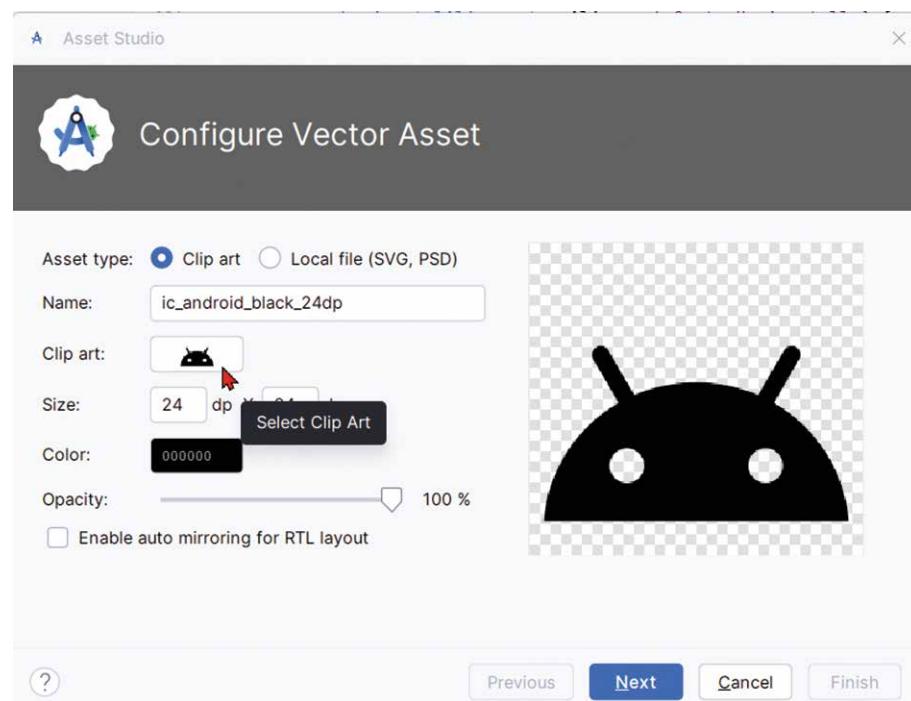


Figura 383

Ao abrir a tela, pesquise por arrow:

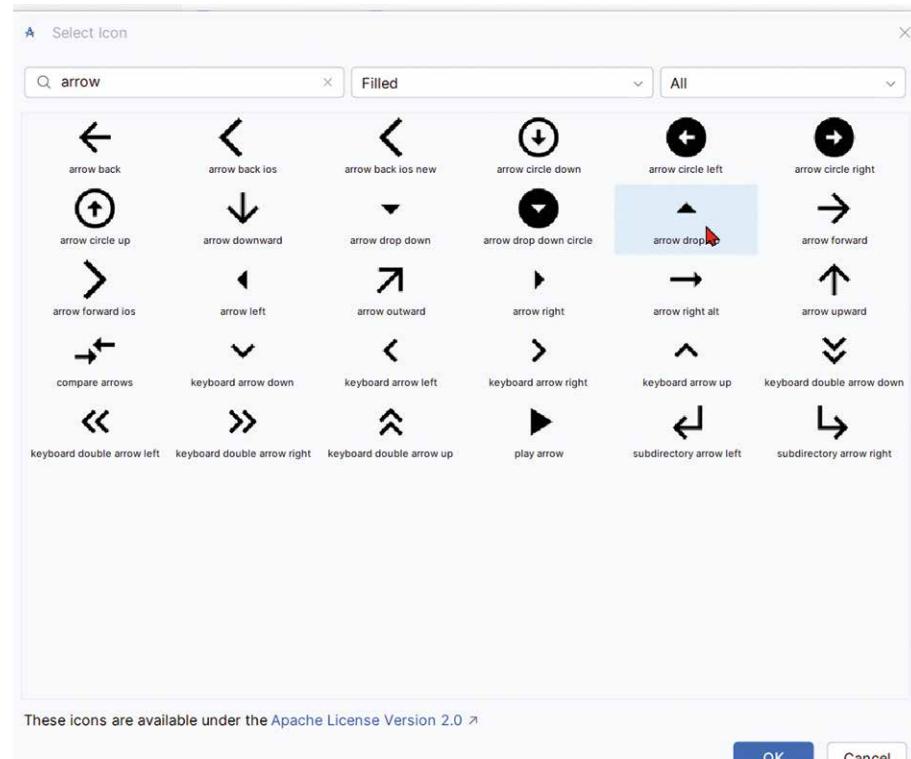


Figura 384

Unidade III

Escolha a seta para cima e clique em OK.

Renomeie para ic_seta_cima (o campo apenas aceita letras minúsculas e underline '_') e clique em Next.

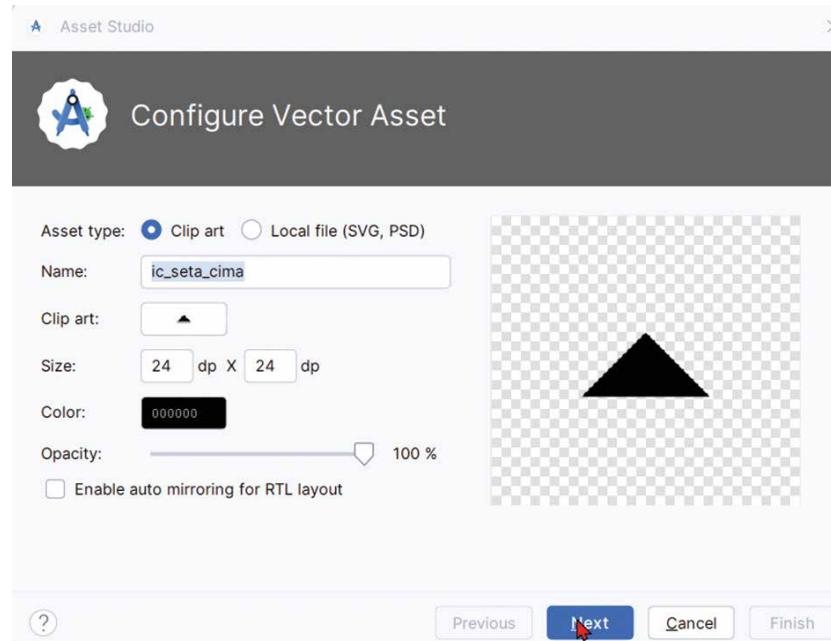


Figura 385

A última tela serve apenas para verificar onde o ícone criado está localizado. Clique em Finish para concluir.

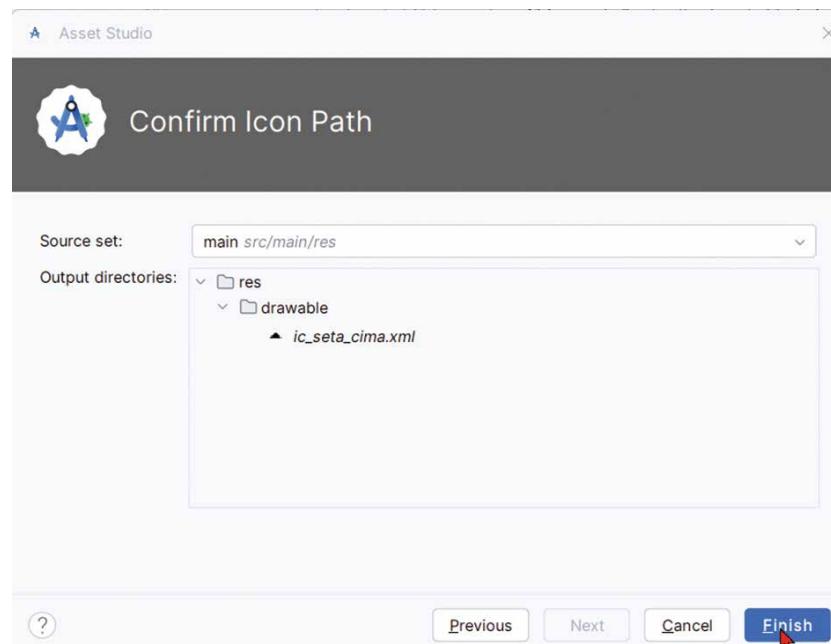


Figura 386

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Para substituir o texto do botão Cima pelo ícone criado, usaremos o componente Icon do Compose.

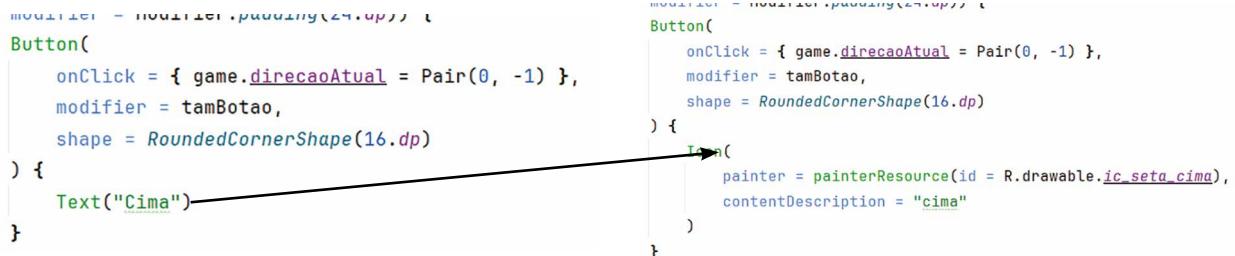


Figura 387

Repita o processo para ic_seta_esquerda, ic_seta_direita e ic_seta_baixo.

O grupo dos botões final fica:

```
//botões de direção
Column(horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier.padding(24.dp)) {
    Button(
        onClick = { game.direcaoAtual = Pair(0, -1) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Icon(
            painter = painterResource(id = R.drawable.ic_seta_cima),
            contentDescription = "cima"
        )
    }
}
Row{
    Button(
        onClick = { game.direcaoAtual = Pair(-1, 0) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Icon(
            painter = painterResource(id = R.drawable.ic_seta_esquerda),
            contentDescription = "esquerda"
        )
    }
    Spacer(modifier = tamBotao)
    Button(
        onClick = { game.direcaoAtual = Pair(1, 0) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Icon(
            painter = painterResource(id = R.drawable.ic_seta_direita),
            contentDescription = "direita"
        )
    }
}
Button(
```

```
        onClick = { game.direcaoAtual=Pair(0,1) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
    Icon(
        painter = painterResource(id = R.drawable.ic_seta_baixo),
        contentDescription = "baixo"
    )
}
```

7.2.4 Criar obstáculos

Daremos forma ao corpo da nossa cobra. Até agora tínhamos apenas um ponto se movendo pela tela. Para criar o corpo da cobra, utilizaremos a coleção mutableListOf, que permite adicionar, remover ou modificar elementos durante a execução do programa, ao contrário do listOf, que é imutável. Isso é essencial para que a cobra possa crescer conforme o jogo avança. Criaremos inicialmente uma cobra com três pontos.

```
var cobra by mutableStateOf(
    mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))
)
```

O código declara uma variável cobra que representa a posição inicial da cobra no jogo. Ela é inicialmente representada por três segmentos, cada um com sua própria posição (x, y) armazenada na lista. O uso de mutableStateOf permite que qualquer parte do código que dependa da posição da cobra seja automaticamente atualizada quando a cobra se move ou cresce.

Como as variáveis posicaoX e posicaoY representam a cabeça da cobra, trocaremos o valor inicial dessas variáveis para as coordenadas do primeiro elemento, o elemento com índice zero.

```
// variáveis
var direcaoAtual by mutableStateOf(Pair(1,0))
var cobra by mutableStateOf(
    mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))
)
var posicaoX by mutableStateOf(cobra[0].first)
var posicaoY by mutableStateOf(cobra[0].second)
```

Observe que agora o ponto mudou para uma posição central do display, mas o corpo não aparece. Como o corpo da cobra, neste momento, possui três elementos, precisamos mostrar cada um deles na tela. No código do JogoCobrinha da tela, vamos então criar uma variável que receberá o corpo da cobra.

```
// tela
BoxWithConstraints() {
    val dimensaoPonto = maxWidth/16
    val tamBotao = Modifier.size(64.dp)
    var corpoCobra = game.cobra
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Para visualizarmos cada segmento do corpo da cobra na tela, utilizaremos o laço `forEach`. Esse laço percorre cada elemento de uma lista. Nesse caso, a lista que contém as posições de cada parte do corpo da cobra. Para cada posição encontrada, posicionaremos um Box que já montamos, compondo assim o corpo completo da cobra.

```
corpoCobra.forEach { corpo ->
    Box(
        modifier = Modifier
            .offset(x = dimensaoPonto * corpo.first, y = dimensaoPonto * corpo.second)
        .size(dimensaoPonto)
        .background(
            Color.DarkGray,
            Shapes().small
        )
    )
}

BoxWithConstraints(){
    Box(
        Modifier.(...))
    {
        corpoCobra.forEach {
            corpo->Box(
                modifier = Modifier
                    .offset(
                        x = dimensaoPonto * corpo.first,
                        y = dimensaoPonto * corpo.second
                    )
                    .size(30.dp)
                    .background(
                        Color.DarkGray,
                        Shapes().small
                    )
            )
        }
    }
}
//botões de direção
```

Figura 388

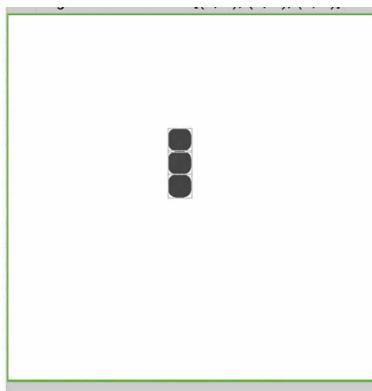


Figura 389



Lembrete

Para rever os conceitos de laços, releia 2.1.4 Repetição.

Apenas para acompanharmos o funcionamento do jogo, alteraremos o texto do cabeçalho para:

```
Column(  
    modifier = Modifier.background(Color.LightGray),  
    verticalArrangement = Arrangement.Top,  
    horizontalAlignment = Alignment.CenterHorizontally  
) {  
    texto = "Jogo da Cobrinha ${corpoCobra.size} ${corpoCobra}"  
    Text(text = texto)  
    //Quadro do jogo
```

A fim de organizar melhor o nosso código e facilitar a compreensão das operações, vamos criar uma variável auxiliar do tipo Pair. Essa variável será utilizada para armazenar as coordenadas (x, y) da nova posição da cabeça da cobra, formada pelo par posicaoX e posicaoY. Na classe Game, crie a variável novaCabeca.

```
var novaCabeca by mutableStateOf(Pair(posicaoX, posicaoY))
```

Dentro do laço da função roda após o cálculo das novas posições X e Y, coloque a atualização da variável novaCabeça:

```
suspend fun roda() {  
    while(true) {  
        posicaoX=posicaoX + direcaoAtual.first  
        posicaoY=posicaoY + direcaoAtual.second  
        posicaoX=(posicaoX + 16) % 16  
        posicaoY=(posicaoY + 16) % 16  
        novaCabeca =Pair(posicaoX, posicaoY)
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Nosso código ainda não faz com que a cobra se move de forma correta. Atualmente, apenas as coordenadas da cabeça da cobra estão sendo alteradas. Para que o corpo da cobra acompanhe o movimento da cabeça, precisamos adicionar a nova posição da cabeça no começo da lista que representa o corpo. O método add adiciona elementos ao final da lista, portanto, precisaremos utilizar um método diferente para inserir elementos no início, fazendo com que todos os outros elementos sejam deslocados para uma posição posterior na lista.

```
// ação
suspend fun roda() {
    while(true) {
        posicaoX=posicaoX + direcaoAtual.first
        posicaoY=posicaoY + direcaoAtual.second
        posicaoX=(posicaoX + 16) % 16
        posicaoY=(posicaoY + 16) % 16
        novaCabeca =Pair(posicaoX,posicaoY)
        cobra.add(0,novaCabeca)
```

Ao adicionar um novo elemento no início da lista, o tamanho da lista que forma o corpo da cobra aumentará a cada rodada. Para manter o tamanho da cobra constante, precisamos reconstruir a lista de forma a preservar o tamanho original.

```
// ação
suspend fun roda() {
    while(true) {
        posicaoX=posicaoX + direcaoAtual.first
        posicaoY=posicaoY + direcaoAtual.second
        posicaoX=(posicaoX + 16) % 16
        posicaoY=(posicaoY + 16) % 16
        novaCabeca =Pair(posicaoX,posicaoY)
        cobra.add(0,novaCabeca)
        cobra=cobra.take(3).toMutableList()
        delay(500)
    }
}
```

O método take(3) retorna uma nova lista contendo os primeiros três elementos da lista original. O método toMutableList() converte a lista resultante em uma lista mutável, permitindo que novos elementos sejam adicionados ou removidos.

Tratamento da comida

Neste jogo, por enquanto, não temos obstáculos, mas o desafio é comer as maçãs. A maçã, além de ser a recompensa do jogador, representa um desafio crescente. À medida que a cobra consome maçãs, ela aumenta de tamanho, dificultando os movimentos e aumentando a probabilidade de colisão com ela mesma.

Para criar a primeira maçã, ela ficará em uma posição fixa, apenas para facilitar a compreensão. Em versões futuras, ela poderá ser colocada em uma posição aleatória.

Na classe game, crie uma nova variável que armazenará as coordenadas da comida.

```
class Game() {
    // variáveis
    var cobra by mutableStateOf(
        mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))
    )
    var direcaoAtual by mutableStateOf(Pair(0,1))
    var posicaoX by mutableStateOf(cobra[0].first)
    var posicaoY by mutableStateOf(cobra[0].second)
    var novaCabeça by mutableStateOf(Pair(posicaoX,posicaoY))
    var comida by mutableStateOf(Pair(5,5))
```

Na função JogoCobrinha, criaremos a variável para receber a atual posição da comida gerada na classe Game.

```
// tela
BoxWithConstraints() {
    val dimensaoPonto = maxWidth/16
    val tamBotao = Modifier.size(64.dp)
    var corpoCobra = game.cobra
    var comidaAtual=game.comida
    Column(
```

Colocaremos no BoxWithConstraints do quadro um Box vermelho arredondado na tela do jogo para representar a comida. A posição do quadrado é definida pelas coordenadas contidas na variável comidaAtual.

```
//Comida
Box(
    modifier = Modifier
        .offset(
            x = dimensaoPonto*comidaAtual.first,
            y = dimensaoPonto*comidaAtual.second)
        .size(dimensaoPonto)
        .background(Color.Red, Shapes().small))
{ }
```

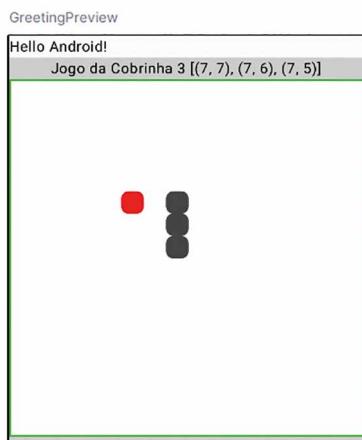


Figura 390

Uma vez criada a comida, precisamos tratar a colisão da cobra com a comida. Quando a cabeça da cobra entra em contato com a comida, duas ações são disparadas:

- A cobra aumenta de tamanho.
- Uma nova comida é gerada em uma posição aleatória na tela.

7.2.5 Colisões do objeto com obstáculos

Quando a cabeça da cobra alcança a posição da comida, o jogo deve aumentar o tamanho da cobra. No entanto, fixar o tamanho em 4 elementos não é flexível, pois a cobra pode comer a comida diversas vezes durante o jogo. Para resolver esse problema, vamos utilizar uma variável para armazenar o tamanho atual da cobra. A cada vez que a comida for consumida, essa variável será incrementada, permitindo que a cobra cresça de forma ilimitada.

```
class Game() {
    // variáveis
    var cobra by mutableStateOf(
        mutableListOf(Pair(7, 7), Pair(7, 6), Pair(7, 5))
    )
    var direcaoAtual by mutableStateOf(Pair(0, 1))
    var posicaoX by mutableStateOf(cobra[0].first)
    var posicaoY by mutableStateOf(cobra[0].second)
    var novaCabeca by mutableStateOf(Pair(posicaoX, posicaoY))
    var comida by mutableStateOf(Pair(5, 5))
    var tamanho by mutableStateOf(3)
```

A colisão é detectada quando as coordenadas da cabeça da cobra coincidem com as coordenadas da comida.

```
if(novaCabeca == comida) {  
}
```

Quando esta condição for verdadeira, então o tamanho deve aumentar.

```
if(novaCabeca==comida) {
    tamanho++
}  
}
```

Para sortear a nova posição, o Kotlin fornece método random que seleciona um número pertencente a um intervalo gerado entre dois valores.

```
(n1..n2).random()
```

Então, vamos criar um para compor a nova localização da comida:

```
if(novaCabeca==comida) {  
    comida=Pair((1..16).random(),(1..16).random())  
    tamanho++  
}
```

Assim, precisamos também atualizar o tamanho do corpo da cobra:

```
if(novaCabeca==comida) {  
    comida=Pair((1..16).random(),(1..16).random())  
    tamanho++  
}  
cobra=cobra.take(tamanho).toMutableList()
```

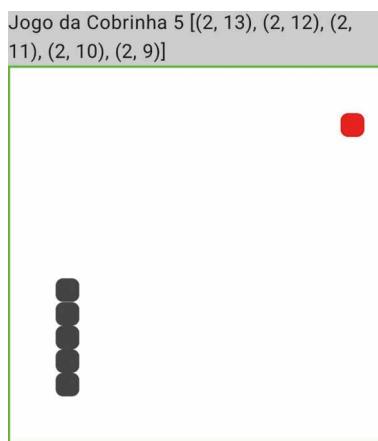


Figura 391

Tratamento da colisão com o próprio corpo

A colisão mais importante é com o próprio corpo, o que deve resultar em um game over. Vamos criar uma variável auxiliar indicando que o jogo terminou, ela será importante mais tarde:

```
// variáveis  
var cobra by mutableListOf(  
    mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))  
)  
var direcaoAtual by mutableStateOf(Pair(0,1))  
var posicaoX by mutableStateOf(cobra[0].first)  
var posicaoY by mutableStateOf(cobra[0].second)  
var novaCabeca by mutableStateOf(Pair(posicaoX,posicaoY))  
var comida by mutableStateOf(Pair(5,5))  
var tamanho by mutableStateOf(3)  
var gameOver by mutableStateOf(false)
```

A detecção de colisão é realizada verificando se a posição atual da cabeça da cobra está contida no conjunto de posições já ocupadas pelo corpo, representado por uma lista mutável. O método contains retorna true se houver uma intersecção entre esses conjuntos. Como o jogo deverá continuar se o game over não acontecer, ou seja, devemos colocar a operação de negação na condição do if.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
    novaCabeca = Pair(posicaoX, posicaoY)
    if(!cobra.contains(novaCabeca)){
        cobra.add(0, novaCabeca)
        if (novaCabeca == comida) {
            comida = Pair((1..16).random(), (1..16).random())
            tamanho++
        }
        cobra = cobra.take(tamanho).toMutableList()
    }
}
```

Para reiniciar o jogo, criaremos uma função chamada `roda` que restaura todas as variáveis aos seus valores iniciais. Ao chamar essa função, também informaremos ao jogador que o jogo foi pausado.

```
    suspend fun roda(){...}
    //reset
    fun reset(){
        tamanho = 3
        cobra = mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))
        direcaoAtual = Pair(0,1)
        novaCabeca = Pair(7,7)
        posicaoX = (novaCabeca.first)
        posicaoY = (novaCabeca.second)
    }
```

Figura 392

A condição final para o encerramento do jogo fica:

```
if(!cobra.contains(novaCabeca)){
    cobra.add(0, novaCabeca)
    if (novaCabeca == comida) {
        comida = Pair((1..16).random(), (1..16).random())
        tamanho++
    }
    cobra = cobra.take(tamanho).toMutableList()
} else{
    gameOver=true
    reset()
}
```

É importante entender este trecho, pois é ele que analisa se a cobra pode se mover para a nova posição sem colidir consigo mesma, o que acontece se encontra a comida ou se o jogo é encerrado. Vamos analisar detalhadamente o algoritmo:

```
if(!cobra.contains(novaCabeca)) {
```

Se a cobra não contém a nova cabeça, verifica-se se a nova posição da cabeça da cobra não coincide com nenhuma outra parte do seu corpo. Isso serve para dizer que a cobra permanece viva.

```
    cobra.add(0, novaCabeca)
```

Então, adicione a nova cabeça no início da lista da cobra. A nova posição da cabeça é adicionada no começo da lista, representando o crescimento da cobra.

```
if (novaCabeça == comida) {  
    comida = Pair((1..16).random(), (1..16).random())  
    tamanho++  
}  
  
}
```

Se a nova cabeça estiver na posição da comida, então uma nova posição aleatória é gerada para a comida dentro do limite entre 1 e 16 e o tamanho da cobra é incrementado, indicando que ela cresceu ao comer a comida.

```
cobra = cobra.take(tamanho).toMutableList()
```

Recorte a lista da cobra para o tamanho atual e a torne mutável novamente. Considerando que a cobra está viva, a lista que forma o seu corpo é cortada para corresponder ao seu tamanho atual, removendo qualquer parte extra que possa ter sido adicionada anteriormente. Esta lista é convertida para uma lista mutável de novo, permitindo que novas posições sejam adicionadas.

```
}else{  
    gameOver=true  
    reset()  
}  
  
}
```

Caso contrário (se a cobra colidiu consigo mesma), então a variável gameOver é definida como true, indicando que o jogo acabou e a função reset() é chamada para reiniciar o jogo.

Precisamos indicar na nossa Activity que o jogo terminou. Vamos utilizar o espaço do texto superior onde estamos monitorando a lista com as coordenadas da cobra para mostrar que o jogo acabou:

```
if (game.gameOver) {  
    texto = "Game Over"  
    jogoRodando = false  
} else {  
    texto = "Jogo da Cobrinha ${corpoCobra.size} ${corpoCobra}"  
}  
Text(text = texto)
```

```
// tela  
BoxWithConstraints(){  
    val dimensãoPonto = maxWidth/16  
    val tamBotao = Modifier.size(64.dp)  
    var corpoCobra = game.cobra  
    var comidaAtual=game.comida  
    Column(  
        modifier = Modifier.background(Color.LightGray),  
        verticalArrangement = Arrangement.Top,  
        horizontalAlignment = Alignment.CenterHorizontally  
    ) {  
        texto = "Jogo da Cobrinha ${corpoCobra.size} ${corpoCobra}"  
        Text(text = texto)  
        //Quadro do Jogo
```

Figura 393

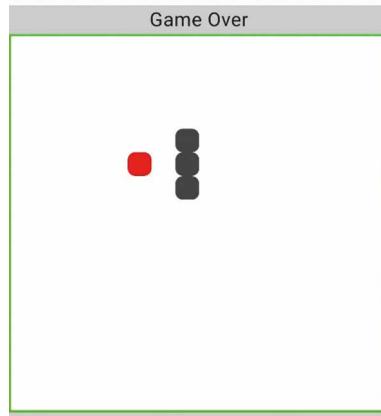


Figura 394

Atualmente, ao perder o jogo, clicar em Start não reinicia a partida. Precisamos ajustar o botão para que ele reinicie a partida após o Game Over. Além disso, alteraremos o rótulo do botão para Iniciar/Reiniciar, permitindo que o jogador reinicie a qualquer momento.

```
//Liga e desliga o jogo
Button(
    onClick = {
        game.gameOver=false
        game.reset()
        jogoRodando = !jogoRodando},
    modifier = Modifier.fillMaxWidth(),
) {
    Text(if (jogoRodando) "Reset" else "Start")
}
```

O nosso jogo já está funcional, faltando apenas acertar os detalhes de acabamento da tela.

7.2.6 Adicionar placar

O nosso jogo está quase pronto, vamos ao acabamento final. Tiraremos o Hello Android!! e no seu lugar utilizaremos um espaçador para evitar que o jogo fique sobre a barra de Status.

```
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column(modifier = Modifier.padding(16.dp)) {
        Spacer(modifier = Modifier.height(30.dp))
        JogoCobrinha()
    }
}
```

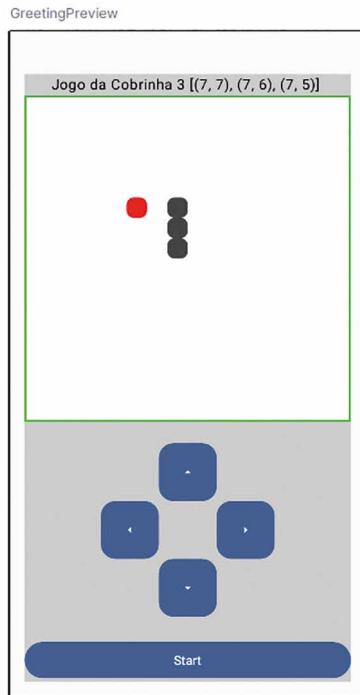


Figura 395

Criaremos o nosso placar, que será uma interface de usuário dinâmica para um jogo. O layout muda dependendo do estado do jogo, exibindo uma mensagem de Game Over ou o placar atual.

Teremos as seguintes variáveis:

- **texto**: se o jogo terminou, o texto a ser exibido é definido como Game Over. Caso contrário, será a palavra PLACAR, seguida do valor da variável game.tamanho, que representa a pontuação do jogador.
- **corTexto**: a cor do texto é definida como branco.
- **corFundo**: a cor de fundo é definida como vermelho para indicar o estado de derrota, e como azul, indicando que o jogo está em progresso.

O texto estará em um Box ocupando toda a largura disponível na tela, a cor de fundo definida com base na variável corFundo, com uma borda verde de 2dp de espessura ao redor da caixa. Esta caixa terá um espaço interno de 20dp entre o conteúdo e as bordas da caixa e os elementos internos estarão centralizados.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Declararemos as variáveis faltantes corTexto e corFundo.

```
// tela
BoxWithConstraints() {
    val dimensaoPonto = maxWidth/16
    val tamBotao = Modifier.size(64.dp)
    var corpoCobra = game.cobra
    var comidaAtual=game.comida
    var corTexto = Color.White
    var corFundo = Color.Blue
```

Alteraremos então o algoritmo que define o texto, definindo não somente o texto, como a cor de fundo, além de pausar o jogo em caso de Game Over, desligando a variável de controle geral do jogo, o jogo rodando:

```
if (game.gameOver) {
    texto = "Game Over"
    corTexto = Color.White
    corFundo = Color.Red //fundo vermelho em caso de game over
    jogoRodando = false //jogo pausado após game over
}

} else {
    texto = "PLACAR ${game.tamanho}"
    corTexto = Color.White
    var corFundo = Color.Blue //fundo azul em caso de jogo rodando
}
```

Definimos o Box, com texto interno centralizado de tamanho 30 sp em negrito e itálico:

```
Box(
    Modifier
        .fillMaxWidth() // Preenche a largura máxima disponível
        .background(corFundo) // Define a cor de fundo
        .border(2.dp, Color.Green) // Borda verde de 2dp
        .padding(20.dp), // Margem de 20dp em todas as direções
        contentAlignment = Alignment.Center // Centraliza o conteúdo dentro da caixa
) {
    Text(text = texto,
        color = corTexto,
        textAlign = TextAlign.Center,
        fontSize = 30.sp,
        fontWeight = FontWeight.Bold,
        fontStyle = FontStyle.Italic)
}
```

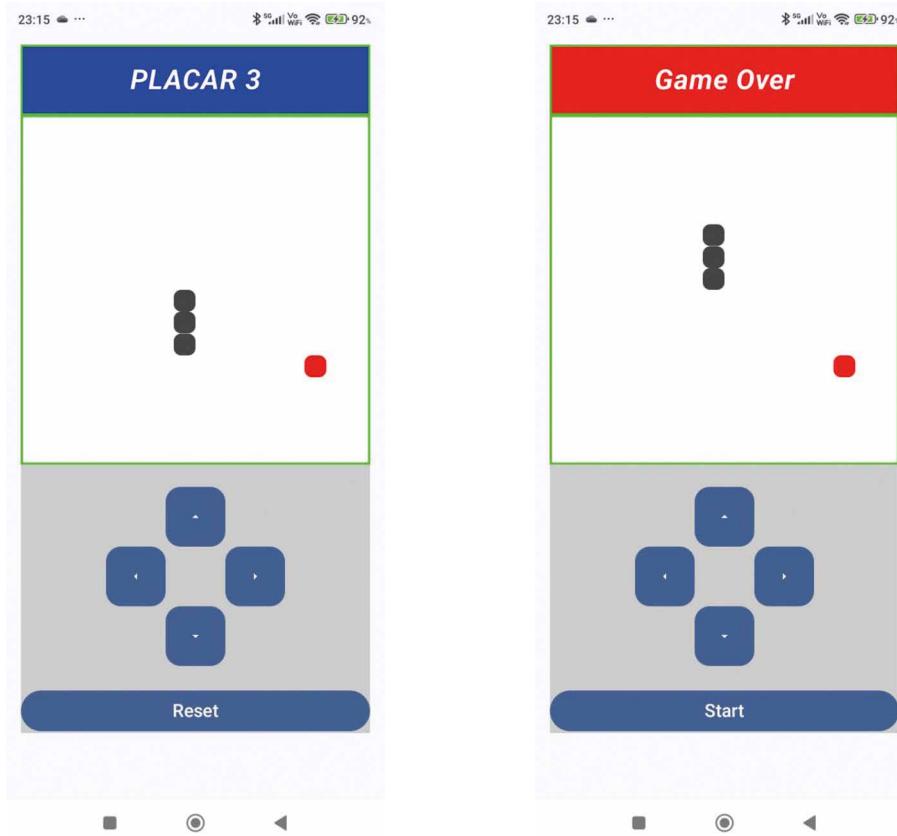


Figura 396

Com isso, nosso jogo básico está concluído, mas ainda há muitas possibilidades de aprimoramento. Podemos adicionar inimigos, representados por caixas na área de jogo, implementar um sistema de vidas para o jogador e aumentar a dificuldade do jogo acelerando a cobra à medida que ela se alimenta, reduzindo o tempo entre cada movimento.

Consta a seguir a listagem final completa do nosso jogo:

MainActivity.kt

```
package com.example.cobrinhadesevolvimento

import android.annotation.SuppressLint
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.BoxWithConstraints
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.offset
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Button
import androidx.compose.material3.Icon
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Shapes
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.text.font.FontStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.cobrinhadesarvolvimento.ui.theme.CobrinhaDesenvolvimentoTheme

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

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column(modifier = Modifier.padding(16.dp)) {
        Spacer(modifier = Modifier.height(30.dp))
        JogoCobrinha()
    }
}
```

Unidade III

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

@Suppress("UnusedBoxWithConstraintsScope")
@Composable
fun JogoCobrinha() {
    // variáveis
    var jogoRodando by remember { mutableStateOf(false) }
    val game = remember{Game()}
    var texto = ""

    // controle
    if (jogoRodando) {
        LaunchedEffect(game) {
            game.roda()
        }
    }
    // tela
    BoxWithConstraints() {
        val dimensaoPonto = maxWidth/16
        val tamBotao = Modifier.size(64.dp)
        var corpoCobra = game.cobra
        var comidaAtual=game.comida
        var corTexto = Color.White
        var corFundo = Color.Blue
        Column(
            modifier = Modifier.background(Color.LightGray),
            verticalArrangement = Arrangement.Top,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {

            if (game.gameOver) {
                texto = "Game Over"
                corTexto = Color.White
                corFundo = Color.Red //fundo vermelho em caso de game over
                jogoRodando = false //jogo pausado após game over
            } else {
                texto = "PLACAR ${game.tamanho}"
                corTexto = Color.White
                var corFundo = Color.Blue //fundo azul em caso de jogo rodando
            }

            Box(
                Modifier
                    .fillMaxWidth() // Preenche a largura máxima disponível
                    .background(corFundo) // Define a cor de fundo de acordo com
a variável
                    .border(2.dp, Color.Green) // Adiciona uma borda verde de 2dp
                    .padding(20.dp), // Adiciona um padding de 20dp em todas as
direções
            )
        }
    }
}
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
        contentAlignment = Alignment.Center // Centraliza o conteúdo
dentro da caixa
    ) {
        Text(text = texto,
            color = corTexto,
            textAlign = TextAlign.Center,
            fontSize = 30.sp,
            fontWeight = FontWeight.Bold,
            fontStyle = FontStyle.Italic)
    }

//Quadro do jogo
BoxWithConstraints() {

    //Campo de jogo
    Box(
        Modifier
            .size(maxWidth)
            .background(Color.White)
            .border(2.dp, Color.Green)
    ) {}

//Comida
Box(
    modifier = Modifier
        .offset(
            x = dimensaoPonto*comidaAtual.first,
            y = dimensaoPonto*comidaAtual.second)
        .size(dimensaoPonto)
        .background(Color.Red, Shapes().small))
}

//Cobra
corpoCobra.forEach {
    corpo ->
    Box(
        modifier = Modifier
            .offset(x = dimensaoPonto * corpo.first, y =
dimensaoPonto * corpo.second)
            .size(dimensaoPonto)
            .background(
                Color.DarkGray,
                Shapes().small
            )
    )
}

//botões de direção
Column(horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier.padding(24.dp)) {
    Button(
        onClick = { game.direcaoAtual = Pair(0, -1) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    )
}
```

```

        Icon(
            painter = painterResource(id = R.drawable.ic_seta_cima),
            contentDescription = "cima"
        )
    }
Row{
    Button(
        onClick = { game.direcaoAtual = Pair(-1, 0) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Icon(
            painter = painterResource(id = R.drawable.ic_seta_esquerda),
            contentDescription = "esquerda")
    }
    Spacer(modifier = tamBotao)
    Button(
        onClick = { game.direcaoAtual = Pair(1, 0) },
        modifier = tamBotao,
        shape = RoundedCornerShape(16.dp)
    ) {
        Icon(
            painter = painterResource(id = R.drawable.ic_seta_direita),
            contentDescription = "direita"
        )
    }
}
Button(
    onClick = { game.direcaoAtual=Pair(0,1) },
    modifier = tamBotao,
    shape = RoundedCornerShape(16.dp)
) {
    Icon(
        painter = painterResource(id = R.drawable.ic_seta_baixo),
        contentDescription = "baixo"
    )
}
}

}

//Liga e desliga o jogo
Button(
    onClick = {
        game.gameOver=false
        game.reset()
        jogoRodando = !jogoRodando},
    modifier = Modifier.fillMaxWidth(),
) {
    Text(if (jogoRodando) "Reset" else "Start")
}

```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
    }
}
```

Logica.kt

```
package com.example.cobrinhadedesenvolvimento

import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import kotlinx.coroutines.delay

class Game() {
    // variáveis
    var cobra by mutableStateOf(
        mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))
    )
    var direcaoAtual by mutableStateOf(Pair(0,1))
    var posicaoX by mutableStateOf(cobra[0].first)
    var posicaoY by mutableStateOf(cobra[0].second)
    var novaCabeca by mutableStateOf(Pair(posicaoX,posicaoY))
    var comida by mutableStateOf(Pair(5,5))
    var tamanho by mutableStateOf(3)
    var gameOver by mutableStateOf(false)
    // ação
    suspend fun roda(){
        while(true){
            posicaoX=posicaoX + direcaoAtual.first
            posicaoY=posicaoY + direcaoAtual.second
            posicaoX=(posicaoX + 16) % 16
            posicaoY=(posicaoY + 16) % 16
            novaCabeca =Pair(posicaoX,posicaoY)
            if(!(cobra.contains(novaCabeca))){
                cobra.add(0, novaCabeca)
                if (novaCabeca == comida) {
                    comida = Pair((1..16).random(), (1..16).random())
                    tamanho++
                }
                cobra = cobra.take(tamanho).toMutableList()
            }else{
                gameOver=true
                reset()
            }
            delay(500)
        }
    }
    //reset
    fun reset(){
        tamanho = 3
        cobra = mutableListOf(Pair(7,7),Pair(7,6),Pair(7,5))
        direcaoAtual = Pair(0,1)
    }
}
```

```
    novaCabeca = Pair(7, 7)
    posicaoX = (novaCabeca.first)
    posicaoY = (novaCabeca.second)
}
}
```

8 PUBLICAÇÃO DE APPS E JOGOS

A publicação de aplicativos na Google Play Store é um processo que exige o cumprimento de normas e o pagamento de taxas, com o objetivo de garantir a qualidade e a segurança dos itens disponíveis para os usuários. Embora possa parecer burocrático, esse processo garante que os aplicativos sejam livres de malwares e vírus, proporcionando uma experiência segura para os usuários. Além disso, a presença na loja confere maior credibilidade aos aplicativos, facilitando a descoberta e a instalação por parte dos usuários.

8.1 Gerando o app instalável

Após a finalização do aplicativo, podemos gerar o pacote instalável (APK), que será distribuído para seus usuários. A maneira tradicional é por meio de APKs (Android Package Kit) e elas podem ser de dois tipos, o APK simples e o Signed APK (APK assinado).

A principal diferença entre um APK e um signed APK está na assinatura digital. Enquanto o primeiro contém todo o código e recursos necessários para instalar e executar um aplicativo Android, o segundo é um APK que foi assinado digitalmente com uma chave privada.

A assinatura digital serve para garantir a autenticidade e a integridade do aplicativo. Quando um APK é assinado, ele inclui um diretório META-INF contendo arquivos que verificam a assinatura e a integridade do conteúdo do APK. Isso permite que o sistema operacional Android valide se o APK foi alterado desde sua assinatura e se ela corresponde à chave registrada para o aplicativo.

Um APK não assinado pode ser usado para testes durante o desenvolvimento, mas não ser distribuído oficialmente na Google Play Store ou em outras plataformas de distribuição de aplicativos. Para publicar um aplicativo, é necessário gerar um signed APK, garantindo que o aplicativo é seguro e confiável para os usuários.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Ambas as construções do pacote de instalação são feitas a partir do menu Build.

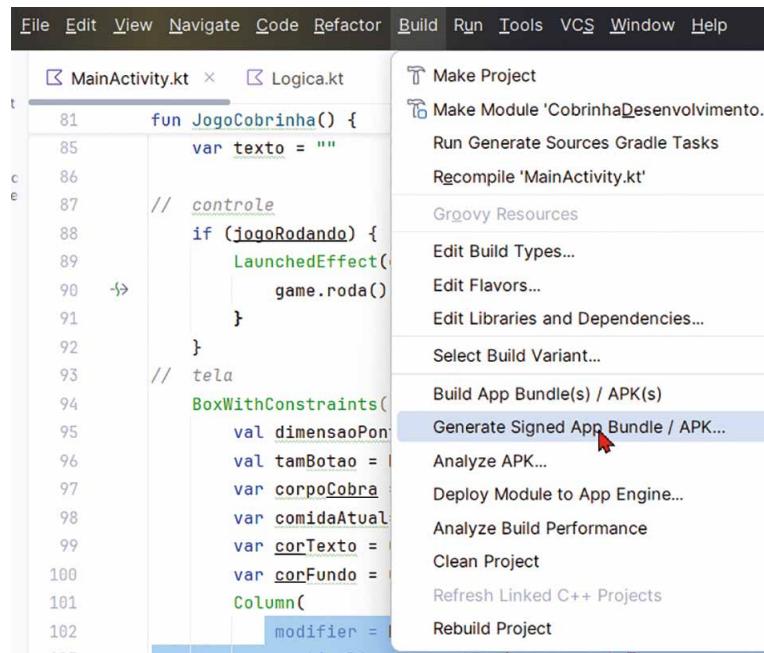


Figura 397

Para criar um APK simples, basta clicar em Build e depois em Build App Bundle(s) / APK(s), em seguida, selecione Create APK. O Android Studio começará a compilar seu projeto e salvará o arquivo de instalação em um local e nome padrão. O processo de criação de APKs pode ser demorado. Após a conclusão da construção, uma janela será exibida. Clique em locate para abrir o explorer de arquivos diretamente na pasta onde o APK foi salvo.

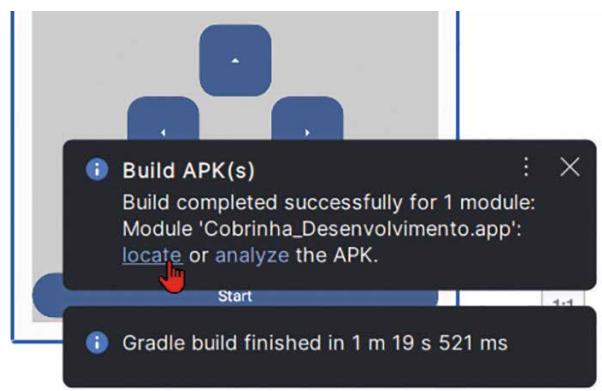


Figura 398

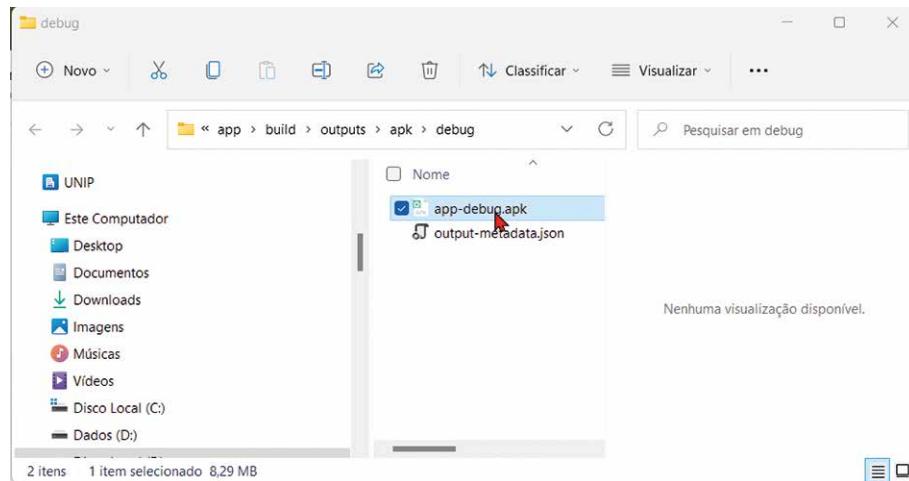


Figura 399

Para acessar a geração do signed APK, no mesmo menu Build, clique em Generate Signed App Bundle / APK...

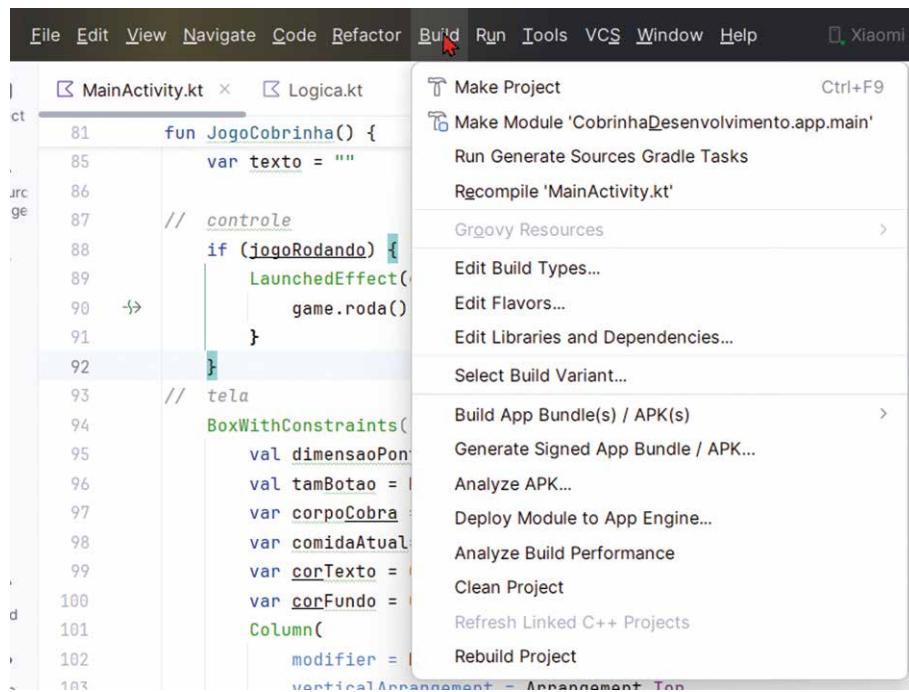


Figura 400

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Uma janela abrirá para escolher entre APK e AAB (Android App Bundle).

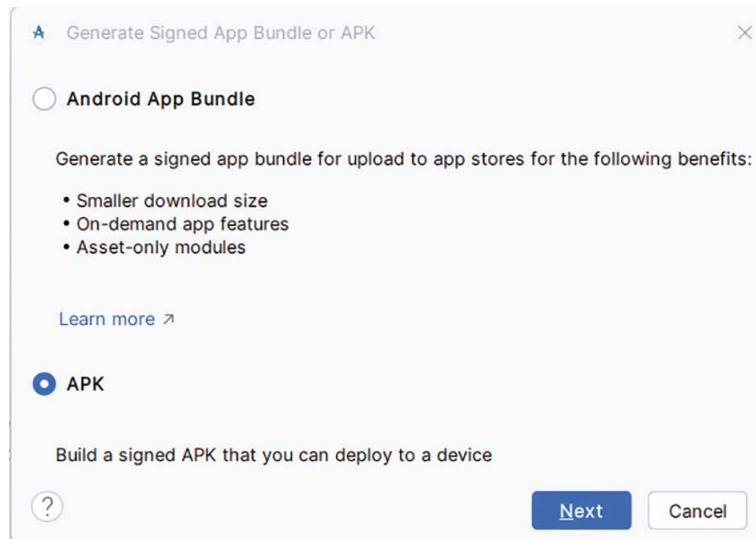


Figura 401

O AAB é um pacote que ocupa menos espaço, pois os usuários não precisam baixar recursos ou códigos desnecessários, economizando espaço de armazenamento em seus dispositivos. Esses pacotes são adaptados às configurações do dispositivo, o que melhora o desempenho e a eficiência do aplicativo. Além disso, quando o aplicativo é atualizado, apenas as partes afetadas são baixadas, em vez de todo o aplicativo, tornando as atualizações mais rápidas e econômicas em termos de dados. No entanto, o AAB não pode ser instalado diretamente em dispositivos, sendo somente um formato de publicação. Ele requer ferramentas adicionais para converter AABs em APKs antes da instalação. Portanto, vamos escolher APK e clicar em Next.

O passo seguinte é anexar a chave. Neste ponto é importantíssimo saber que as informações que serão preenchidas devem estar guardadas, pois se uma delas estiver errada, você não conseguirá mais publicar o seu aplicativo.

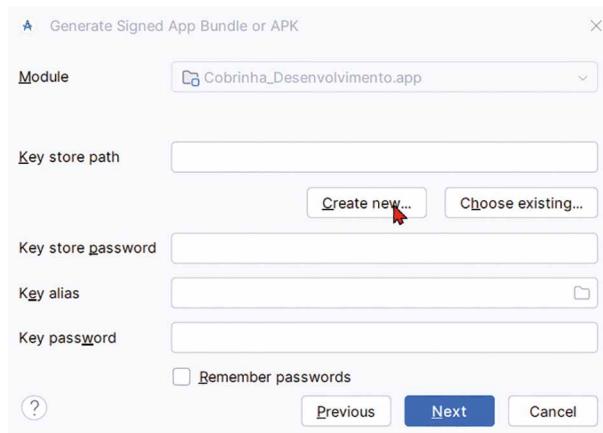


Figura 402

Caso ainda não tenha uma chave, crie uma, clicando em Create New.

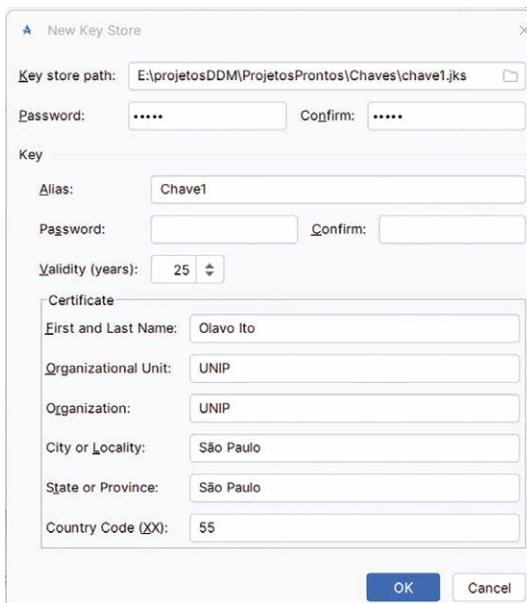


Figura 403

Na tela de criação da chave, complete e guarde todas as informações e tire um Backup. Esta chave servirá para este e outros projetos criados por você. Clique em OK e continue até o novo APK ser criado.

8.2 Informações e especificações técnicas dos detalhes do app

É fundamental que todos os arquivos estejam em conformidade com as diretrizes da plataforma, garantindo a qualidade e compatibilidade do aplicativo. Organizar os arquivos e informações antes de iniciar o processo de registro garante a agilidade na conclusão do cadastro e na redução de erros. Ao ter todos os materiais à mão, o desenvolvedor pode dedicar-se integralmente ao preenchimento das informações e à finalização do processo. As principais informações para o registro do aplicativo no Play são:

- Idioma padrão (pt-br).
- Título.
- Uma descrição curta sobre as funções do aplicativo.
- Uma descrição completa sobre as funções do aplicativo.
- As telas do seu App.
- Um ícone de alta qualidade, que deverá ter exatamente 512px por 512px.
- Uma imagem que ficará como sua propaganda na loja, deverá ter 1024px de largura por 500px de altura.
- Saber classificação do aplicativo, se ele é um jogo ou um aplicativo.

8.3 Versões do app

Durante o processo de construção de um APK (arquivo de instalação do aplicativo Android), uma chave privada digital é associada ao aplicativo. Ela serve como uma assinatura única que identifica o desenvolvedor e garante a autenticidade do aplicativo. Ao realizar uma atualização, o desenvolvedor utiliza a mesma chave privada para assinar a nova versão do aplicativo. Dessa forma, o dispositivo do usuário consegue verificar se a atualização é legítima, comparando a assinatura da nova versão com a assinatura da versão já instalada.

8.4 Classificação do app

Existe um órgão internacional da Coalizão Internacional de Classificação Etária (IARC) que oferece as classificações de conteúdo para diversos países. Assim, se o aplicativo for de distribuição mundial, ele deve ser classificado conforme o país de destino. Esta classificação é obrigatória para a publicação na Play Store.

8.5 Preços e distribuição

A definição do modelo de monetização de um aplicativo é uma decisão estratégica que varia de acordo com o objetivo do desenvolvedor e as características do mercado. Não existem regras rígidas ou tabelas predefinidas para determinar se um aplicativo será gratuito, pago ou conterá anúncios. Essa escolha depende de diversos fatores, como o público-alvo, a concorrência, o valor percebido pelo usuário e os custos de desenvolvimento. O desenvolvedor possui a liberdade de explorar diferentes modelos de negócio e ajustar sua estratégia conforme a evolução do mercado, ou mesmo oferecer gratuitamente visando se promover, inclusive podendo ganhar dinheiro através de anúncios ou compras dentro do aplicativo ou assinaturas.

8.6 Conta de desenvolvedor

Para publicar um aplicativo na Google Play Store, é necessário ter uma conta de desenvolvedor. A fim de criar uma conta de desenvolvedor, é preciso acessar o site do Google Play Console:

<https://play.google.com/apps/publish/>

Ao acessar o site, será solicitado que você faça login com sua conta oficial do Google. Na sequência, será necessário pagar uma taxa de US\$ 25 para criar a conta de desenvolvedor. Essa taxa será debitada da sua carteira virtual.

Após o pagamento, você receberá um e-mail de confirmação. O e-mail conterá um link para concluir o cadastro da sua conta de desenvolvedor. Depois de concluir o cadastro, você poderá começar a publicar seus aplicativos na Google Play Store.



Resumo

Nesta unidade, discutimos o desenvolvimento de um aplicativo Android que utiliza a API Gemini para fornecer sugestões de roteiros turísticos personalizados. A interface do usuário foi desenvolvida em Kotlin, utilizando a técnica de Views do Android Studio. Destacamos a utilização de inteligência artificial em aplicativos como uma tendência crescente, permitindo experiências altamente personalizadas e a automação de tarefas complexas.

Além disso, desenvolvemos um protótipo do clássico jogo da cobrinha utilizando o Jetpack Compose. O jogo foi implementado de forma declarativa, facilitando a criação e manutenção da interface do usuário. O tabuleiro do jogo é composto por uma grade de 16x16 quadrados, no qual a cobra se movimenta e os alimentos aparecem. O layout do jogo inclui um display de informações na parte superior, o campo de jogo no meio e os controles na parte inferior. Através deste projeto aprofundamos nossos conhecimentos em desenvolvimento de jogos, aprendendo a lidar com game loops, colisões e criação de elementos interativos.

Abordamos os passos essenciais para a publicação de aplicativos na Google Play Store, destacando a importância de seguir normas específicas e pagar taxas para garantir a qualidade e segurança dos aplicativos. Exploramos a necessidade de organizar todas as informações necessárias, incluindo a associação de uma chave privada digital ao APK e a classificação do aplicativo conforme as diretrizes da Coalizão Internacional de Classificação Etária (IARC). Além disso, discutimos a definição de modelos de monetização e o processo de criação de uma conta de desenvolvedor, proporcionando uma visão abrangente sobre como garantir que os aplicativos sejam seguros e confiáveis para os usuários.



Exercícios

Questão 1. O GitHub é uma plataforma interessante para desenvolvedores, pois permite o armazenamento, gerenciamento e colaboração em projetos de software. No contexto do desenvolvimento Android com o Android Studio, a integração com o GitHub facilita o versionamento de código, a colaboração em equipe e o gerenciamento de alterações no projeto.

Considerando as funcionalidades e os processos do GitHub, avalie as afirmativas a seguir.

I – O GitHub é uma plataforma que utiliza o sistema de controle de versão Git, permitindo o rastreamento de alterações no código e a colaboração entre desenvolvedores.

II – No Android Studio, é possível clonar repositórios do GitHub, criar repositórios e realizar operações como commit, push e pull diretamente na interface do ambiente de desenvolvimento.

III – Para conectar um projeto Android Studio ao GitHub, é necessário primeiro criar um repositório no GitHub e, depois, usar a opção Share Project on GitHub no menu VCS do Android Studio.

É correto o que se afirma em:

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

Resposta correta: alternativa E.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: o GitHub é uma plataforma baseada no sistema de controle de versão Git. Ele permite que desenvolvedores armazenem e gerenciem código-fonte, rastreiem alterações (por meio de commits) e colaborem em projetos. O Git é a tecnologia que gerencia o histórico de alterações, enquanto o GitHub fornece uma interface amigável e recursos adicionais aos desenvolvedores.

II – Afirmativa correta.

Justificativa: o Android Studio tem integração nativa com o Git e o GitHub. Por meio do menu Version Control System (VCS), os desenvolvedores podem clonar repositórios existentes, criar repositórios

e realizar operações comuns do Git, como commit, push e pull. Essa integração facilita o gerenciamento de versões diretamente na IDE, sem a necessidade de usar ferramentas externas.

III – Afirmativa correta.

Justificativa: para conectar um projeto do Android Studio ao GitHub, o desenvolvedor deve primeiro criar um repositório no GitHub (pela interface web ou por outras ferramentas). Em seguida, no Android Studio, ele pode usar a opção VCS > Import into Version Control > Share Project on GitHub para vincular o projeto local ao repositório remoto. Esse processo configura automaticamente o Git no projeto e envia os arquivos iniciais para o GitHub.

Questão 2. A publicação de aplicativos na Google Play Store envolve uma série de etapas e de requisitos técnicos para garantir a qualidade, segurança e conformidade com as diretrizes da plataforma. Desde a geração do pacote (APK ou AAB) até o preenchimento das informações necessárias para o registro do aplicativo, cada etapa é crucial para o sucesso da publicação.

Nesse contexto, avalie as afirmativas a seguir.

I – O Android App Bundle (AAB) é um formato de pacote que otimiza o tamanho do aplicativo, permitindo que os usuários baixem apenas os recursos necessários para seu dispositivo, mas ele não pode ser instalado diretamente, exigindo conversão para APK antes da instalação.

II – Durante o registro do aplicativo na Google Play Store, é necessário fornecer informações como o título do aplicativo, descrições curta e completa, telas do aplicativo, um ícone de alta qualidade e uma imagem de propaganda.

III – A classificação etária do aplicativo, fornecida pela Coalizão Internacional de Classificação Etária (IARC), é opcional e somente necessária caso o aplicativo seja destinado a um público específico ou contenha conteúdo sensível.

É correto o que se afirma em:

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

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: o AAB é um formato de pacote que permite a criação de aplicativos otimizados para diferentes dispositivos. Ele reduz o tamanho do aplicativo, pois os usuários baixam apenas os recursos necessários para seu dispositivo. No entanto, ele não pode ser instalado diretamente em dispositivos Android. A Google Play Store é responsável por converter o AAB em APKs específicos para cada dispositivo durante o processo de distribuição. Essa conversão é feita automaticamente pela plataforma.

II – Afirmativa correta.

Justificativa: para publicar um aplicativo na Google Play Store, o desenvolvedor deve fornecer diversas informações, incluindo: título do aplicativo, descrição curta (até 80 caracteres), descrição completa (até 4.000 caracteres), capturas de telas do aplicativo, ícone de alta qualidade (de 512x512 pixels) e imagem de propaganda para promoção na loja (de 1024x500 pixels). Essas informações são essenciais para o registro e para a aprovação do aplicativo.

III – Afirmativa incorreta.

Justificativa: a classificação etária do aplicativo fornecida pela IARC é obrigatória para todos os aplicativos publicados na Google Play Store, independentemente do público-alvo ou do conteúdo. Ela ajuda a informar os usuários sobre a adequação do aplicativo para diferentes faixas etárias e é um requisito para a publicação.

REFERÊNCIAS

- ARAÚJO, S. *Lógica de programação e algoritmos*. São Paulo: Contentus, 2020.
- BENEDETTI, L. Snake: conheça a história do jogo que revolucionou o mundo mobile. *Universo Retrô*, 23 mar. 2021. Disponível em: <https://shre.ink/MRXp>. Acesso em: 6 mar. 2025.
- BRANDÃO, A. *APP para iniciantes: faça seu primeiro aplicativo low code*. Jundiaí: Paço e Literatura, 2022.
- CARDOSO, L. C. *Design de aplicativos*. Curitiba: Intersaberes, 2015.
- DEITEL, H.; DEITEL, P.; DEITEL, A. *Android para programadores*. Porto Alegre: Grupo A.
- DEVELOPERS. *Android Studio*. [s.d.]a. Disponível em: <https://shre.ink/M6h0>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Ciclo de vida da atividade*. [s.d.]b. Disponível em: <https://shre.ink/M6At>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Configuração de testes avançados*. [s.d.]c. Disponível em: <https://shre.ink/MNnp>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Desenvolver uma interface com visualizações*. [s.d.]d. Disponível em: <https://shre.ink/M6YG>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Executar apps em um dispositivo de hardware*. [s.d.]e. Disponível em: <https://shre.ink/M6ZJ>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Fundamentos de aplicativos*. [s.d.]f. Disponível em: <https://shre.ink/M6o9>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *ListView*. [s.d.]g. Disponível em: <https://shre.ink/M6PM>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Scaffold*. [s.d.]h. Disponível em: <https://shre.ink/M0zr>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Temas*. [s.d.]i. Disponível em: <https://shre.ink/M63T>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *View*. [s.d.]j. Disponível em: <https://shre.ink/M6EU>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Visão geral do build do Gradle*. [s.d.]k. Disponível em: <https://shre.ink/M6FS>. Acesso em: 6 mar. 2025.
- DEVELOPERS. *Visão geral dos avisos*. [s.d.]l. Disponível em: <https://shre.ink/M0Ua>. Acesso em: 6 mar. 2025.

DEVELOPERS. *Introdução a corrotinas no Android Studio*. 21 maio 2024. Disponível em: <https://shre.ink/MRBq>. Acesso em: 6 mar. 2025.

FORBELLONE, A. V.; EBERSPÄCHER, H. F. *Lógica de programação: a construção de algoritmos e estruturas de dados*. São Paulo: Pearson, 2005.

GOOGLE AI FOR DEVELOPERS. *Tutorial: primeiros passos com a API Gemini*. [s.d.]. Disponível em: <https://shre.ink/MVFH>. Acesso em: 6 mar 2025.

GUERREIRO, M. *Os efeitos do Game Design no processo de criação de Jogos Digitais utilizados no Ensino de Química e Ciências: o que devemos considerar?* 2015. Dissertação (Mestrado em Educação para a Ciência) – Universidade Estadual Paulista – Júlio de Mesquita Filho, Bauru, São Paulo, 2015.

GLAUBER, N. *Dominando o Android com Kotlin*. São Paulo: Novatec, 2019.

KOTLIN. *Coroutines*. 18 out. 2022. Disponível em: <https://shre.ink/MRBq>. Acesso em: 6 mar. 2025.

M3. *Material design*. [s.d.]. Disponível em: <https://m3.material.io/>. Acesso em: 6 mar. 2025.

MOMA. *Art and artists*. [s.d.]. Disponível em: <https://shre.ink/MRiL>. Acesso em: 6 mar. 2025.

RESENDE, K. *Kotlin com android: crie aplicativos de maneira fácil e divertida*. São Paulo: Casa do Código, 2018.

SHIMOHARA, C.; SOBREIRA, E. R.; ITO, O. Potencializando a programação de jogos digitais de matemática através do Scratch e da avaliação Game Flow. In: Workshop de Informática na Escola (WIE), 22, 2016, Uberlândia. *Anais [...]*. Uberlândia: Congresso Brasileiro de Informática na Educação (CBIE), 2016.

SINTES, T. *Aprenda Orientação a Objetos em 21 dias*. São Paulo: Pearson, 2002.



UNIVERSIDADE PAULISTA

Informações:
www.sepi.unip.br ou 0800 010 9000