Desenvolvimento de Aplicativos Nativos para o Sistema Operacional Android

com o uso do IDE Android Studio e Linguagem de Programação Java

ANDROID

E-Book de apoio ao curso

Desenvolvido pelo prof. Sérgio Luiz Banin

Ano 2026

Capítulo 1 – O Ecossistema Android

1.1. O que é o Android

O Android é um sistema operacional desenvolvido inicialmente pela empresa Android Inc. e posteriormente adquirido pelo Google. Ele é baseado no kernel do Linux e foi projetado principalmente para dispositivos móveis, como smartphones e tablets.

Uma das principais características do Android é ser uma plataforma aberta, permitindo que desenvolvedores criem aplicações para diferentes dispositivos.

Características do Android

O Android é o sistema operacional de código aberto voltado para dispositivos móveis. Desenvolvido pelo Google e baseado no núcleo do sistema operacional Linux, é atualmente o software mais popular do mundo em sua categoria.

Ele é o software básico de smartphones, tablets, TVs e wearables, responsável pelo gerenciamento de hardware, permitindo a instalação e execução de diversos aplicativos em total integração com serviços do Google.

Uma vez que é de código aberto, é altamente personalizável e amplamente utilizado por fabricantes de dispositivos de hardware como Samsung, Motorola, LG, entre outros.

Os aplicativos para Android são disponibilizados ao público por meio da Play Store do Google, uma plataforma que concentra milhares de aplicações criadas por uma ampla comunidade de desenvolvedores ao redor do mundo.

1.2. Tipos de Aplicações Android

Em linhas gerais é possível classificar as aplicações Android em três categorias:

- **Nativas**
  São desenvolvidas especificamente para o Android, utilizando linguagens como Java ou Kotlin e as APIs oficiais da plataforma.

  Oferecem melhor desempenho, maior integração com o sistema operacional e acesso completo aos recursos do dispositivo.
- **Híbridas**
  São desenvolvidas utilizando tecnologias web, como HTML, CSS e JavaScript, e executadas dentro de um contêiner nativo.

  Permitem reaproveitamento de código entre diferentes plataformas, mas podem apresentar limitações de desempenho e integração.
- **Web Apps**
  São aplicações acessadas por meio do navegador, sem necessidade de instalação.

  Dependem de conexão com a internet e possuem acesso restrito aos recursos do dispositivo.

Neste curso, o foco será exclusivamente no desenvolvimento de aplicações nativas, por serem as mais utilizadas no mercado e oferecerem maior controle sobre o comportamento e desempenho do aplicativo.

1.3. Aplicações Nativas Android

O que são as aplicações nativas Android?

Um aplicativo Android é um software desenvolvido para executar no sistema operacional Android, permitindo a interação com o usuário por meio de interfaces gráficas.

Além da interface, um aplicativo Android é composto por diferentes componentes, como Activities (telas), Services (processos em segundo plano), Broadcast Receivers (resposta a eventos do sistema) e Content Providers (compartilhamento de dados). Esses componentes trabalham de forma integrada e seguem o ciclo de vida definido pelo sistema operacional.

Os aplicativos são geralmente desenvolvidos utilizando linguagens como Java ou Kotlin, e utilizam recursos do próprio sistema, como acesso à internet, sensores do dispositivo (GPS, câmera, acelerômetro) e armazenamento de dados.

Após o desenvolvimento, os aplicativos podem ser distribuídos por meio de plataformas como a Play Store, onde ficam disponíveis para instalação pelos usuários.

O que é preciso para escrever uma aplicação Android?

Para um programador conseguir desenvolver uma aplicação Android é necessário reunir e dominar um conjunto requisitos, tais como:

- **Ambiente de desenvolvimento (IDE)**
  Android Studio instalado e configurado
  SDK do Android (bibliotecas e ferramentas da plataforma)
- **Linguagem de programação**
  Java (linguagem que vamos adotar) ou
  Kotlin
- **Conhecimentos fundamentais**
  Lógica de programação
  Programação orientada a objetos (POO)
  Estrutura de um projeto Android
- **Componentes da aplicação**
  Activities (telas)
  Services (execução em segundo plano)
  Broadcast Receivers (resposta a eventos)
  Content Providers (compartilhamento de dados)
- **Interface do usuário (UI)**
  Layouts (XML ou Jetpack Compose)
  Elementos visuais (botões, textos, listas, etc.)
  Princípios básicos de usabilidade
- **Recursos do sistema (Resources)**
  Imagens
  Strings (textos)
  Cores e estilos
  Arquivos de layout
- **Gerenciamento de dependências**
  Gradle (sistema de build)
  Bibliotecas externas
- **Acesso a funcionalidades do dispositivo**

Internet
Câmera
GPS/localização
Sensores (quando disponíveis)
Permissões do sistema

- **Testes**
  Emulador do Android
  Dispositivo físico
  Testes básicos de funcionamento
- **Geração e distribuição do aplicativo**
  Geração do APK ou AAB
  Publicação na Play Store (ou outras formas de distribuição)

Não se preocupe com esta lista neste momento (sim, ela é bem grande).

Neste curso vamos abordar um conjunto de assuntos suficientes para a produção de aplicativos bem legais.

Saiba inicialmente que o primeiro requisito é um ambiente de desenvolvimento adequado. A ferramenta oficial para isso é o Android Studio, que oferece suporte completo para criação de interfaces, escrita de código, testes e simulação de aplicativos em dispositivos virtuais ou reais.

Em relação à linguagem de programação, o desenvolvimento Android é feito principalmente com Java ou Kotlin. Atualmente, quando este texto está sendo escrito, Java segue sendo a linguagem mais usada, porém o Google recomenda fortemente a adoção do Kotlin por ser mais moderna, concisa e segura.

Em termos de mercado de trabalho para os estudantes, Java ainda apresenta maior quantidade de vagas, conforme pode ser constatado nas ofertas de sites de colocação profissional.

Também é fundamental compreender os conceitos básicos da plataforma Android, como a estrutura de um projeto, o funcionamento dos componentes (Layouts, ViewGroups, Views, Activities, Services, Broadcast Receivers e Content Providers) e o ciclo de vida das aplicações. Esses elementos definem a aparência e o comportamento do aplicativo, bem como ele interage com o sistema operacional.

Outro ponto importante é o uso de recursos (resources) do sistema, como layouts para construção de interfaces, arquivos de recursos (imagens, textos, estilos) e permissões para acesso a funcionalidades do dispositivo, como internet, câmera e localização.

Além disso, é necessário utilizar o sistema de build (Gradle), responsável por compilar o projeto, gerenciar dependências e gerar o arquivo final do aplicativo (APK ou AAB) para distribuição.

Por fim, testes e validação são etapas essenciais. O desenvolvedor pode utilizar emuladores ou dispositivos físicos para verificar o funcionamento do aplicativo, garantindo desempenho, compatibilidade e uma boa experiência para o usuário.

Com esses elementos, é possível iniciar o desenvolvimento de aplicações Android de forma estruturada e alinhada às práticas recomendadas da plataforma.

1.4.Software Development Kit (SDK) do Android

O SDK (Software Development Kit), ou Kit de Desenvolvimento de Software, é um conjunto de ferramentas, bibliotecas, documentações e amostras de código que permite aos desenvolvedores criarem aplicativos para uma plataforma específica — neste caso, o Android.

Neste contexto, os termos SDK e API (Application Program Interface) são usados como sinônimos.

Se compararmos o desenvolvimento de um app com a construção de uma casa, o SDK não é apenas o martelo; ele é a maleta completa do mestre de obras, contendo:

- As ferramentas (martelo, furadeira).

- Os manuais de instrução (como assentar o piso).
- As peças pré-fabricadas (portas e janelas prontas).

O SDK do Android se manifesta principalmente em quatro pilares:

1. **Bibliotecas de APIs**: São os "blocos" de código prontos. Quando o programador digita
```
import android.widget.Button;
...
    private Button btnTarefa;
...
```
ele está acessando uma funcionalidade fornecida pelo SDK, no caso a classe Button, que oferece recursos para criar e interagir com um botão na tela. Desse modo o programador não precisa programar um botão do zero.
2. **Ferramentas de Compilação** (Build Tools): Softwares que compilam o código Java e os arquivos XML, empacotando tudo no formato .apk ou .aab que o celular entende.
3. **Emuladores e Debuggers**: Ferramentas que permitem testar o aplicativo em um dispositivo virtual e encontrar erros no código em tempo real.
4. **Platform Tools**: Utilitários de baixo nível, como o ADB (Android Debug Bridge) que faz a comunicação entre o computador do aluno e o celular físico.

Níveis de SDK (API Level)

O Android está em constante evolução, recebendo atualizações frequentes que introduzem novos recursos, melhorias de desempenho e ajustes de segurança.

Cada nova versão do sistema é identificada de duas formas principais: por um número de versão (como Android 6.0 ou Android 13) e, nas versões mais antigas, por nomes de sobremesas (como KitKat e Marshmallow).

No entanto, para o desenvolvimento de aplicações, o identificador mais importante não é o nome da versão, mas sim o nível da API (API Level), também conhecido como nível do SDK.

O nível da API é um número inteiro que representa a versão técnica da plataforma Android. Cada nível de API corresponde a um conjunto específico de funcionalidades disponíveis para o desenvolvedor, incluindo classes, métodos e comportamentos do sistema.

É esse número que o Android Studio utiliza para definir compatibilidade, acesso a recursos e comportamento da aplicação, por meio de configurações como *minSdkVersion*, *targetSdkVersion* e *compileSdkVersion*.

Por exemplo, o Android 6.0 (Marshmallow) corresponde ao nível de API 23. Isso significa que, ao configurar um projeto com esse nível de API, o desenvolvedor tem acesso às funcionalidades introduzidas até essa versão do sistema.

Para que você compreenda melhor a importância de dominar o conceito de Nível de API (API Level) leia com atenção a próxima seção em que tratamos da evolução do Android.

Evolução do Android

O Android evoluiu significativamente ao longo dos anos, com diversas versões que trouxeram melhorias de desempenho, segurança e novos recursos para desenvolvedores e usuários.

A primeira versão comercial do sistema foi lançada em 2008 (Android 1.0), marcando o início da plataforma no mercado de dispositivos móveis. Nos anos seguintes, o Android passou por rápidas atualizações, com versões importantes como o Android 2.3 (Gingerbread), em 2010, que ampliou o suporte a diferentes tipos de dispositivos, e o Android 4.0 (Ice Cream Sandwich), em 2011, que unificou a interface entre smartphones e tablets.

Em 2014, o Android 5.0 (Lollipop) trouxe uma grande reformulação visual com a introdução do Material Design, estabelecendo um novo padrão de interface. Mais recentemente, versões como o Android 10 (2019) e o

Android 12 (2021) reforçaram aspectos como privacidade, segurança e personalização da experiência do usuário.

Inicialmente, as versões do Android eram nomeadas com nomes de sobremesas em ordem alfabética (como Cupcake, Donut, Lollipop, entre outros), prática que foi abandonada a partir do Android 10, quando o sistema passou a ser identificado principalmente por números de versão.

O quadro seguir, mostra de forma simplificada uma cronologia relacionando versões, nomes comerciais e níveis de API (SDK):

Versão	Nome Interno (não oficial)	API (SDK)	Ano de Lçto	Principais características	Relevância em 2026
1.0	—	1	2008	Lançamento inicial; suporte básico a apps, notificações e integração com serviços Google	Obsoleta
1.1	Petit Four*	2	2009	Correções e pequenas melhorias de API	Obsoleta
1.5	Cupcake	3	2009	Teclado virtual, widgets na tela inicial	Obsoleta
1.6	Donut	4	2009	Suporte a diferentes resoluções de tela, busca aprimorada	Obsoleta
2.0 / 2.1	Eclair	5–7	2009–2010	Navegação GPS, múltiplas contas, melhorias na câmera	Obsoleta
2.2	Froyo	8	2010	Melhor desempenho (JIT), tethering	Obsoleta
2.3	Gingerbread	9–10	2010	Melhor gerenciamento de energia, suporte a sensores	Obsoleta
3.x	Honeycomb	11–13	2011	Interface otimizada para tablets	Obsoleta
4.0	Ice Cream Sandwich	14–15	2011	Fragment API: Unificação de layouts para Celular e Tablet, interface Holo. ✅ Captura de tela nativa.	Obsoleta
4.1–4.3	Jelly Bean	16–18	2012–2013	Project Butter: maior fluidez com até 60fps na interface. Google Now e notificações ricas (com botões). ✅	Obsoleta
4.4	KitKat	19–20	2013	Otimização de recursos para dispositivos com pouca memória. WebView baseada em Chromium. Introdução do NFC Host Card Emulation – usado em pagamentos por aplicativo. ✅	Obsoleta
5.0–5.1	Lollipop	21–22	2014–2015	Material Design – elementos de interface modernizados. ✅ Android Runtime (ART) – compila o aplicativo para código de máquina na instalação. ✅	Muito Baixa
6.0	Marshmallow	23	2015	Permissões em tempo de execução. Modo Doze – recurso de economia de energia.	Baixa
7.0–7.1	Nougat	24–25	2016	Recurso de Tela dividida que permite dividir a tela entre dois apps. Compilação JIT/AOT combinada com o ART (modelo híbrido de copilação). Introdução das notificações interativas. ✅	Baixa
8.0–8.1	Oreo	26–27	2017	Canais de notificação que permitem ao usuário controlar tipos de avisos. Melhorias de segurança. Recurso de Autopreenchimento	Moderada

Versão	Nome Interno (não oficial)	API (SDK)	Ano de Lçto	Principais características	Relevância em 2026
9	Pie	28	2018	Digital Wellbeing (bem-estar digital) – recurso que permite controlar o tempo de tela por aplicativo. ✅ Restrições de uso de rede em background com objetivo de melhorar autonomia da bateria e economizar dados móveis. ✅ Navegação por gestos e bateria adaptativa.	Moderada
10	Quince Tart	29	2019	Modo Escuro (Dark Theme) nativo. Navegação por gestos. Fim das permissões de IMEI/Serial – foco na privacidade do usuário pois impede que aplicativos acessem o IMEI e o nº Serial do aparelho. ✅	Moderada
11	Red Velvet Cake	30	2020	Conversas organizadas (Conversations) – gerenciamento de mensagens em tempo real. ✅ Armazenamento com escopo (scoped storage) – modelo de armazenamento focado em privacidade e segurança. ✅ Permissões de localização "Só desta vez".	Alta
12 / 12L	Snow Cone	31–32	2021–2022	Material You (Material Design 3). ✅ API de Splash Screen obrigatória. ✅	Alta
13	Tiramisu	33	2022	Permissão de Notificação passou a ser opt-in. ✅ Seletor de fotos nativo (Photo Picker). ✅ Idioma por aplicativo. ✅	Alta
14	Upside Down Cake	34	2023	Melhorias gerais de segurança e desempenho. API de gênero gramatical. ✅ Restrição de fotos – acesso apenas a arquivos selecionados.	Muito Alta
15	Vanilla Ice Cream	35	2024	Arquivamento de Apps – libera espaço sem perder dados. ✅ Edição de tela parcial (compartilhamento).	Muito Alta
16	Baklava	36	2025	Modo Desktop - Janelas redimensionáveis nativas. ✅ IA Generativa integrada ao sistema. ✅	Essencial

Quadro 1.1 – Versões dA API do Android

As classificações na coluna de Relevância em 2026 devem ser interpretada da seguinte forma:

Relevância	Como interpretar
Obsoleta	São API's muito antigas e com recursos insuficientes ou limitados diante das demandas atuais. Não devem ser consideradas de modo nenhum em novos projetos. E projetos que ainda as usam devem ser atualizados para versões mais recentes.
Muito Baixa e Baixa	Ainda presentes em dispositivos em uso, porém em franco declínio e no final da vida útil. São aparelhos com muitos anos, com hardware antigo e quase obsoletos. Dar suporte a essas versões significa renunciar a importantes recursos.
Moderada	Presentes em dispositivos ainda em uso, que estão entre metade e o final de sua vida útil. O suporte a essas versões pode ainda ser relevante dependendo do público alvo da aplicação.
Alta e Muito Alta	São as versões mais utilizadas atualmente. Devem ser consideradas como principal foco de desenvolvimento.
Essencial	Deve ser sempre considerada como referência para testes e compatibilidade (targetSdkVersion e compileSdkVersion mais recente).

Uso dos Níveis de API em um projeto Android

Todo projeto Android exige, desde o seu início, a definição de três parâmetros essenciais: *minSdkVersion*, *targetSdkVersion* e *compileSdkVersion*.

Esses parâmetros determinam, respectivamente, a compatibilidade, o comportamento e a compilação do aplicativo. Neste momento, o mais importante é compreender o papel de cada um deles no projeto e mais adiante veremos como e onde eles são configurados no projeto.

minSdkVersion – palavra-chave Compatibilidade

Define o nível mínimo de API (ou versão do Android) em que o aplicativo pode ser executado.

Ou seja, dispositivos com versões inferiores não conseguirão instalar o aplicativo.

Importância

O Android utiliza esse valor para definir a versão mais antiga do sistema operacional compatível com o aplicativo.

Exemplo prático

Se o *minSdkVersion* for definido como API 21 (Android 5.0 – Lollipop), usuários com dispositivos na API 19 (Android 4.4 – KitKat) não poderão instalar o app — e ele nem será exibido na Play Store para esses usuários.

Impacto no desenvolvimento

Se você utilizar um recurso disponível apenas em APIs mais recentes (como canais de notificação, introduzidos na API 26), o Android Studio emitirá alertas indicando que aquele recurso não é compatível com versões mais antigas.

targetSdkVersion – palavra-chave Comportamento

Indica a versão do Android para a qual o aplicativo foi otimizado e testado.

Na prática, é uma forma de informar ao sistema: "*Este aplicativo está preparado para seguir as regras desta versão do Android*."

Importância

O Android utiliza esse valor para aplicar regras de comportamento e segurança.

Se o *targetSdkVersion* for antigo, o sistema pode ativar um modo de compatibilidade, mantendo comportamentos antigos para evitar falhas no aplicativo.

Exemplo prático

A partir do Android 6.0 (API 23), as permissões passaram a ser solicitadas em tempo de execução.

Se o *targetSdkVersion* for menor que 23 as permissões são concedidas automaticamente.

Se for 23 ou maior, o app deve solicitar permissões via código

Caso isso não seja implementado corretamente, o aplicativo pode apresentar falhas em tempo de execução.

Regra de ouro

Sempre utilizar uma versão recente. A Play Store exige que aplicativos publicados estejam alinhados com versões atuais do Android.

compileSdkVersion – palavra-chave Compilação

Define qual versão do SDK será utilizada pelo Android Studio para compilar o código-fonte.

Importância

O Android utiliza esse valor para realizar a compilação, o que determina quais APIs estão disponíveis durante o desenvolvimento.

Exemplo prático

Se você deseja utilizar uma funcionalidade introduzida na API 34 (Android 14), sua *compileSdkVersion* deve ser, no mínimo, 34.

Caso contrário, o compilador não reconhecerá as classes e métodos mais recentes.

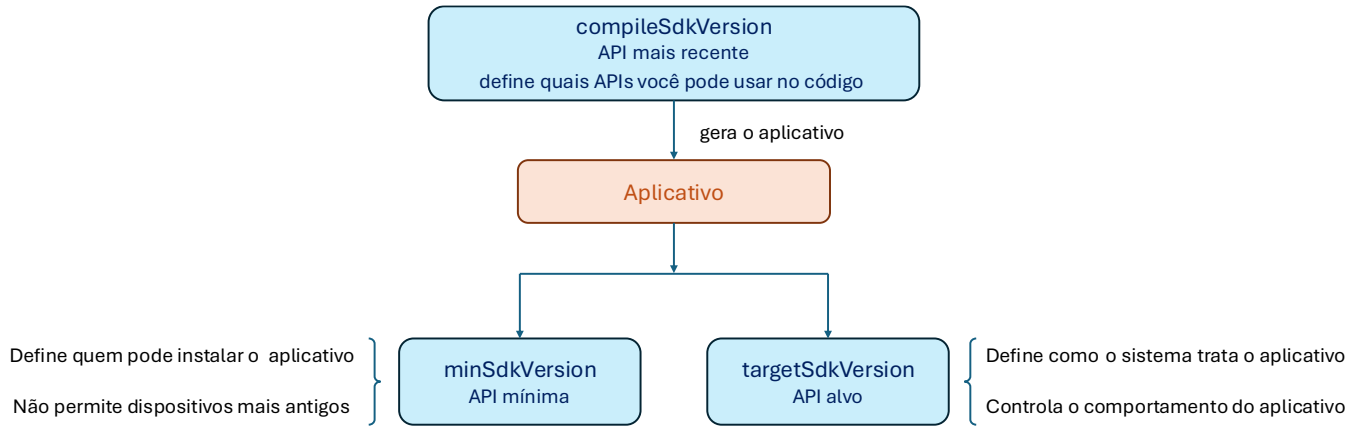
Importante

Esse parâmetro não afeta diretamente os usuários do aplicativo – ele é utilizado apenas no ambiente de desenvolvimento.

Regra de ouro

Utilize sempre a versão mais recente disponível para ter acesso a todos os recursos do Android e suas melhorias.

Resumo



A imagem acima e o quadro a seguir fornecem uma comparação direta entre esses três parâmetros e reforça o entendimento dos conceitos.

Parâmetro	Características	Impacto principal
`minSdkVersion`	• Define Compatibilidade mínima; • Representa o menor nível de API suportado; • Dispositivos com API menor não conseguem instalar o app; • Impacta diretamente o alcance de usuários;	Quem pode instalar o aplicativo
`targetSdkVersion`	• Define o Comportamento do aplicativo; • O Android pode **alterar regras de execução** com base nele; • Mesmo rodando em versões mais novas, o sistema pode aplicar modo de compatibilidade se o targetSdkVersion for antigo; • Podem existir restrições de upload na Play Store se o targetSdkVersion for muito antigo;	Como o aplicativo se comporta
`compileSdkVersion`	• Define quais APIs podem ser usadas no código; • Não afeta diretamente quais dispositivos podem instalar o aplicativo; • Deve ser sempre a versão mais recente possível;	O que você pode programar e quais API's você pode usar

Capítulo 2 – Interfaces em aplicativos Android

2.1. Introdução

Após os conceitos iniciais do Capítulo 1 agora é o momento de começar a implementar os primeiros aplicativos reais.

Neste capítulo você irá:

- Entender o que é uma Activity;
- Criar layouts (interface visual) utilizando XML;
- Entender os principais componentes visuais;
- Conectar a interface visual como com o código Java;
- Implementar os primeiros eventos.

2.2.Entendendo uma Activity

O que é Activity?

Em termos simples, uma Activity pode ser entendida como uma tela do aplicativo.

De forma mais técnica, trata-se de um dos principais componentes do Android, responsável por representar uma única interface com a qual o usuário pode interagir.

Cada Activity corresponde, normalmente, a uma tela específica do aplicativo, como uma tela de login, uma lista de produtos ou um formulário de cadastro.

Estrutura de uma Activity

Para a correta implementação de uma Activity é necessário que você conheça, compreenda e aplique três elementos fundamentais:

- **Interface gráfica (UI)**

  Define a aparência da tela.

  Pode ser construída utilizando uma de duas opções:
  - arquivos de layout em XML; ou
  - esquema de códigos do Jetpack Compose.
- **Lógica e regras funcionais**

  Define o que acontece na tela.

  É implementada por meio de uma classe em Java ou Kotlin, utilizando as APIs do Android para exibir texto e imagem; tratar eventos, controlar dados e responder às ações do usuário.
- **Ciclo de Vida da Activity**

  Define como a tela se comporta no aplicativo e em relação à outras telas que possam existir nele.

  Compreender esse ciclo é essencial para garantir o bom funcionamento do aplicativo, especialmente em situações como mudanças de tela, rotação do dispositivo ou uso em segundo plano.

  Cada Activity possui um ciclo de vida controlado pelo sistema operacional e definida pelos estados:
  - criação (`onCreate`)
  - exibição (`onStart`, `onResume`)
  - pausa (`onPause`)
  - finalização (`onDestroy`)

  Estes estados do ciclo de vida serão vistos com detalhes mais adiante neste curso.

Papel da Activity no aplicativo

As Activities também são responsáveis por estruturar a navegação do aplicativo.

Um aplicativo Android normalmente é composto por várias Activities, e a transição entre elas ocorre por meio de um conceito chamado *Intents*, permitindo que o usuário navegue entre diferentes telas.

Declaração de Activities no AndroidManifest.xml

Em um aplicativo Android, não basta apenas criar uma Activity no código.

Para que ela seja reconhecida e possa ser utilizada pelo sistema, **é obrigatório** declará-la no manifesto do aplicativo, representado pelo arquivo *AndroidManifest.xml*.

Ao declarar uma Activity no manifesto, você informa ao sistema que aquela tela faz parte do aplicativo e pode ser:

- iniciada pelo próprio app
- acessada por outras Activities
- chamada por outros aplicativos (dependendo da configuração)

Sem essa declaração, a Activity simplesmente não existe para o sistema, mesmo que esteja corretamente implementada no código.

Se uma Activity não estiver registrada no *AndroidManifest.xml*, o aplicativo apresentará erro ao tentar abri-la, geralmente resultando em falha de execução (crash).

Resumo sobre Activities

- Uma Activity é uma tela do aplicativo;
- Cada Activity necessita de um Layout, que é quem define seu aspecto visual;
- As funcionalidades da tela são criadas no código Java/Kotlin associado à Activity;
- Toda Activity tem um ciclo de vida, gerenciado pelo sistema operacional Android;

2.3. Conceitos sobre Layout de Activities

As interfaces no Android são definidas em arquivos XML ou através de códigos do pacote JetPack Compose.

Neste item vamos nos concentrar nos elementos de criação de Layouts na forma de arquivos XML.

O que é um Layout XML?

Layout XML no Android é um arquivo texto de marcação (markup language) formado por *tags* específicas usado para definir a estrutura visual e a interface do usuário (UI) de uma tela do aplicativo. Exemplo:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/buttonCenter"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botão Centralizado"/>

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

Código 2.1 – Exemplo de arquivo XML do Android

Sua importância e realizar a separação entre o design e o código lógico da tela (que é escrito em Java/Kotlin). Ele organiza os elementos visuais da tela, como textos, imagens, botões e outros, permitindo criar telas responsivas e adaptáveis a diferentes tamanhos de dispositivos.

Local de armazenamento

Os arquivos XML ficam dentro da pasta `res/layout` localizada na raiz do projeto. O caminho completo é este:

```
{nome do projeto}/app/src/main/res/layout
```

Organização do arquivo de Layout

Um layout de um projeto Android é um arquivo XML

XML, sigla para Extensible Markup Language (Linguagem de Marcação Extensível), é uma linguagem de marcação que define um conjunto de regras para codificar documentos em um formato legível tanto para humanos quanto para máquinas.

Pense em um sistema de organização flexível que permite estruturar informações de maneira padronizada. É exatamente isso que o XML faz! Ele não se limita a um conjunto pré-definido de tags como o HTML, usado para exibir conteúdo em páginas da web. Com o XML, você define suas próprias tags, criando uma estrutura personalizada para seus dados.

Essa flexibilidade é o que torna o XML tão poderoso e amplamente utilizado em sistemas computacionais.

A estrutura do XML é baseada nas tags, que são palavras-chave envolvidas por sinais de “menor que” (<) e “maior que” (>). As tags funcionam como contêineres, definindo o significado dos dados que envolvem.

Toda tag XML precisa, obrigatoriamente, ter início e fechamento. O fechamento pode ser de dois tipos:

- **Completo**: quando a tag de fechamento contém o nome completo da tag, precedido pela barra /

```
<produto  (atributos) >
    Elementos aninhados
</produto>
```

  Este tipo é usado quando a tag possui outros elementos dentro dela
- **Simples**: quando a tag de fechamento formada por apenas /> (barra e o sinalo maior que)

```
<item
    atributo1
    atributo1
    etc />
```

Um conjunto de tags forma um elemento, que representa uma unidade de informação. Os elementos podem conter atributos, que fornecem informações adicionais sobre o elemento.

Desde o início do Android, o Google escolheu esse formato de arquivo para definir os arquivos de Layout das Activities. No Código 1 (página anterior) você encontra um exemplo típico, onde dois elementos estão definidos:

- Um Layout:
  iniciando em `<androidx.constraintlayout.widget.ConstraintLayout ... >`
  terminando em `</androidx.constraintlayout.widget.ConstraintLayout>`
- Uma caixa de texto:
  iniciando em `<TextView ...`
  terminando em `/>`

No exemplo note que o fechamento da tag do ConstraintLayout é completa e do TextView é do tipo simples.

E os três pontos (. . .) mostrados representam atributos das tags. Tais atributos definem os aspectos visuais e comportamentais do componente.

2.4. Layout, View e ViewGroup

Já vimos que em Android, o termo layout refere-se à estrutura visual de uma tela, ou seja, à forma como os elementos da interface são organizados.

Precisamos dar o próximo passo e falar dos conceitos de **View** e **ViewGroup**., que são usados no layout para:

- Definir quais componentes aparecem na tela;
- Como eles estão organizados;
- Como se relacionam entre si.

Na criação do layout View e ViewGroup são as classes usadas para criar a interface de usuário (UI).

View

Uma View é um elemento da interface que o usuário vê e com os quais pode interagir. Exemplos comuns:

TextView	Button	ImageView	CheckBox
EditText	ImageButton	Switch	RadioButton

Eles ocupam uma área retangular na tela, possuem atributos que permitem controlar sua posição, tamanho, margens, conteúdo, aparência, efeitos e comportamento.

Além disso, podem ser associados a eventos como toque (clique), toque longo, arrasto, ganho de foco, entrada de teclado e vários outros.

ViewGroup

É um contêiner, a priori, invisível que contém Views. As Views contidas são chamadas de filhas.

Eles também ocupam uma área retangular na tela e possuem atributos para definição de sua posição na tela, tamanho, margens e aparência.

E talvez você pergunte: mas eles não são invisíveis? Em geral, sim, porém é possível configurar alguns elementos de aparência como uma cor ou imagem de fundo (atributo background).

A função desses contêineres é estruturar suas Views filhas de modo que o aplicativo tenha uma aparência rica e organizada.

Os ViewGroups podem ser classificados em algumas categorias, tais como:

Layouts

- ConstraintLayout – contêiner moderno, flexível e de alto desempenho que permite criar interfaces complexas, responsivas e planas (sem aninhamento), posicionando suas views através de vínculos (constraints) em relação ao pai ou a outros elementos;
- LinearLayout – contêiner que organiza suas views sequencialmente em uma única direção, que pode ser vertical ou horizontal;
- FrameLayout – contêiner simples para exibir uma ou mais views. Ele empilha os elementos filhos, um sobre o outro, ocupando o mesmo espaço, sendo que o primeiro elemento adicionado fica no fundo e o último elemento adicionado fica no topo;
- RelativeLayout – contêiner que organiza suas views com base em posições relativas entre si (irmãos) ou em relação ao contêiner pai. Embora ainda faça parte do SDK e seja totalmente funcional, seu uso foi amplamente substituído na prática pelo ConstraintLayout, que oferece maior flexibilidade e melhor desempenho em layouts mais complexos.

Scrollers (componentes de rolagem)

- ScrollView – contêiner para uma única View cujo conteúdo é mais alta que sua área retangular e que precisa rolar verticalmente;
- HorizontalScrollView – semelhante ao ScrollView, porém com conteúdo mais largo que sua área e com rolagem horizontal.
- NestedScrollerView – Uma versão mais moderna que suporta rolagem aninhada (um ScrollView dentro de outro, por exemplo);

AdapterViews (componentes baseados em adaptadores)

- RecyclerView – contêiner moderno e eficiente para listas. Ele reutiliza views para economizar memória e melhorar a performance
- ListView – uma versão mais antiga e menos eficiente para listas;
- GridView – exibe elementos em uma grade bidimensional;
- Spinner – lista suspensa de seleção;

Contêineres de funcionalidade específica

- CardView – contêiner especial que com bordas e sombreamento, semelhante a um cartão físico;
- ViewPager – contêiner que permite deslizar lateralmente entre diferentes telas, comum em tutoriais ou abas;
- AppBarLayout / ToolBar – contêineres de topo de tela que gerenciam menus, ações e títulos;
- FragmentContainerView

Hierarquia de Layout

Os elementos da interface são organizados em uma estrutura hierárquica, semelhante a uma árvore.

O elemento raiz geralmente é um ViewGroup e dentro dele existem outros elementos que podem ser Views ou outros ViewGroups.

Exemplo:

```
<LinearLayout>
    <TextView />
    <Button />
</LinearLayout>
```

Nesse caso temos um `LinearLayout` como elemento pai (no Android usa-se a palavra em inglês: *parent*).

E temos duas views como filhas: `TextView` e `Button`.

A hierarquia define uma relação de dependência entre os elementos. Duas regras se aplicam sempre:

- Uma View sempre depende de um ViewGroup para existir na tela.
- O posicionamento de um elemento depende do seu elemento pai.

E isso implica em duas consequências:

- A forma como um componente aparece está diretamente ligada ao layout onde ele está inserido;
- Alterar o layout pai pode impactar todos os elementos filhos;

Aninhamento de layouts

O aninhamento ocorre quando um ViewGroup contém outro ViewGroup, que por sua vez pode conter outros elementos, tanto Views como ViewGroups.

Veja o exemplo a seguir.

É um caso em que temos um LinearLayout pai, um LinearLayout filho e, dentro dele, outras Views.

```
<LinearLayout>
    <LinearLayout>
        <TextView />
        <Button />
    </LinearLayout>
</LinearLayout>
```

Embora seja comum organizar a interface utilizando múltiplos layouts, o uso excessivo de aninhamento pode impactar o desempenho da aplicação.

Isso acontece porque o sistema Android precisa:

- Medir cada elemento;
- Calcular o posicionamento de cada elemento;
- Desenhar o elemento na tela;

Quanto mais níveis na hierarquia, mais trabalho o sistema terá.

Um layout muito aninhado pode causar os seguintes impactos:

- Interface mais lenta para carregar
- Maior consumo de processamento
- Experiência menos fluida para o usuário

Importante: Em telas simples, isso pode não ser perceptível. Em interfaces mais complexas, o impacto pode ser bem relevante.

2.5. ConstraintLayout

Após compreender como os layouts funcionam e como a interface é organizada em forma de hierarquia, surge uma questão importante:

Qual layout devemos utilizar para construir interfaces de forma eficiente?

Atualmente, o `ConstraintLayout` é a principal recomendação para o desenvolvimento de interfaces no Android.

Um dos principais motivos para utilizar o `ConstraintLayout` é a capacidade de reduzir a quantidade de ViewGroups aninhados e, como exposto acima, esse aninhamento pode causar problemas de performance.

Em layouts como o `LinearLayout`, é comum criar múltiplos níveis apenas para alinhar elementos.

Exemplo de aninhamento

```
<LinearLayout orientation="vertical">
    <LinearLayout orientation="horizontal">
        <TextView />
        <TextView />
    </LinearLayout>

    <LinearLayout orientation="horizontal">
        <Button />
        <Button />
    </LinearLayout>
</LinearLayout>
```

Com o ConstraintLayout, esses mesmos elementos podem ser organizados em um único nível, sem a necessidade de múltiplos contêineres.

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

A redução da profundidade da hierarquia implica na melhora de desempenho.

Por outro lado, o ConstraintLayout introduz atributos que permitem posicionar os elementos filhos com base em vínculos (constraints):

- Em relação ao elemento pai; ou
- Em relação a outros elementos filhos; ou
- Em relação a ambos

Esses atributos estabelecem regras de vinculação como estas a seguir:

```
app:layout_constraintTop_toTopOf="parent"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintEnd_toEndOf="parent"
```

No lugar de `parent` pode ser usado o id de qualquer elemento presente no Layout.

2.6. Criação de Layouts – Unidades de Medida

Contexto – Conceito de Pixel (unidade *px*)

O hardware de qualquer tela, seja TV, computador, celular ou outros, é formado por pontos individuais que podem ser ligados e desligados. Cada um desses pontos pode exibir uma cor específica, e o conjunto deles forma a imagem visível no dispositivo.

Um ponto desses é comumente chamado de pixel (*picture element*). A imagem à direita mostra um mosaico de pixels, cada um com sua cor configurada.

Por outro lado, todo elemento visual, seja *View* ou *ViewGroup*, ocupa uma área retangular na tela. Essa área possui largura e altura, que podem ser medidas em pixels.

Assim, tudo que é exibido na tela ocupa uma determinada quantidade de pixels.

No entanto, nem tudo é tão simples assim. Vamos adiante.

O problema

Os dispositivos Android apresentam uma grande variedade de tamanhos de tela e densidades de pixels.

Em dispositivos diferentes, a mesma quantidade de pixels pode resultar em tamanhos visuais distintos, porque o tamanho do pixel físico – unidade px – varia de um aparelho para outro.

Em aparelhos com alta densidade, há mais pixels concentrados em um mesmo espaço físico. Em aparelhos com baixa densidade, essa concentração é menor.

Como resultado, uma mesma quantidade de pixels pode representar tamanhos visuais bastante diferentes. Assim, ao utilizar pixels (px) para definir dimensões, corre-se o risco de criar interfaces desproporcionais, com elementos que não mantêm um tamanho consistente entre diferentes dispositivos.

Assim, é necessário compreender as unidades utilizadas para definir o tamanho dos elementos na interface.

Unidade de Medida Density-Independent Pixel (unidade dp)

Para resolver o problema das variações de tamanho e densidade de tela, o Android utiliza uma unidade chamada dp (*density-independent pixel*).

O dp é uma unidade de medida que não depende diretamente da quantidade de pixels físicos da tela. Em vez disso, o sistema realiza uma conversão automática, ajustando o valor definido para que o elemento mantenha aproximadamente o tamanho físico em diferentes dispositivos.

Isso significa que, ao utilizar dp, as Views e ViewGroups tendem a ocupar proporções semelhantes, independentemente da densidade da tela.

Isto é feito através de uma conversão de unidades entre px e dp, usando para isso a densidade da tela.

Exemplo

O sistema operacional Android adota como padrão uma densidade de 160 dpi (dots per inch).

Este padrão implica que 2,54 cm (1 polegada) há 160 pixels físicos.

Em telas com essa densidade vale a equivalência: **1 dp = 1 px (para a densidade padrão de 160 dpi)**.

A fórmula de conversão é esta abaixo

$$px = \frac{dpi\ do\ dispositivo}{160} . dp$$

A partir desse padrão o valor de conversão entre px e dp pode ser calculado de modo proporcional.

Em um aparelho com densidade 320 dpi ocorrerá a equivalência: 1 px = 2 dp

O Quadro 2.1 mostra as relações mais comuns encontradas em dispositivos existentes no mercado.

Densidade	DPI	Fator de Escala
LDPI (baixa)	120	0.75x
MDPI (média)	160	1x (padrão)
HDPI (alta)	240	1.5x
XHDPI (extra)	320	2x
XXHDPI	480	3x
XXXHDPI	640	4x

Quadro 2.1 – Fator de escala para conversão de unidades px - dp

Uso do dp na prática

O dp deve ser utilizado para definir dimensões da interface. Tipicamente é usado nos atributos:

- largura (layout_width);
- altura (layout_height);
- margens (margin);
- espaçamentos internos (padding);

O uso do dp garante que a interface seja:

- mais consistente entre diferentes dispositivos;
- mais previsível durante o desenvolvimento;
- mais confortável para o usuário;

Unidade de Medida Scale-Independent Pixel (unidade sp)

Embora o dp seja ideal para dimensões gerais da interface, ele não é a melhor escolha para todos os casos.

Quando trabalhamos com textos, é importante considerar não apenas o tamanho físico dos elementos, mas também as **preferências de acessibilidade do usuário**.

Para isso, o Android disponibiliza a unidade **sp** (scale-independent pixel).

O **sp** funciona de forma semelhante ao **dp**, levando em conta a densidade da tela.

No entanto, ele possui uma diferença importante, além da densidade, o **sp** também considera o tamanho de fonte configurado pelo usuário nas configurações do dispositivo.

Acredito que você deve saber que o sistema Android permite que o usuário aumente ou diminua o tamanho dos textos para melhorar a leitura.

Assim, quando utilizamos **sp**:

- o texto acompanha essa configuração
- a interface se torna mais acessível

Por outro lado, quando utilizamos dp:

- o tamanho do texto permanece fixo
- podendo dificultar a leitura em alguns casos

Resumo sobre unidade de medida

- **px** – *pixel físico*: Corresponde a um pixel físico na tela.
  Se você desenhar algo com 100px, terá tamanhos físicos muito diferentes em um celular barato versus um topo de linha, pois a densidade de pixels varia.
- **dp** – *density-independent pixel*: É uma unidade virtual que se baseia na densidade física da tela.
  Ela normaliza a exibição, garantindo que um elemento tenha aproximadamente o mesmo tamanho físico em qualquer tela.
- **sp** – *scale-independent pixel*: É uma unidade virtual usada para tamanhos de caracteres de texto.
  Similar ao dp, mas também considera as preferências de tamanho de fonte do usuário nas configurações do sistema.

2.7. Criação de Layouts – Largura e Altura de elementos visuais

Contexto

Ao construir uma interface no Android, não basta apenas adicionar componentes à tela. É necessário também definir como esses elementos irão ocupar o espaço disponível.

Todo elemento visual, seja uma View ou um ViewGroup, possui uma área retangular definida por dois atributos principais, que **são configurados de forma independente e si**:

- layout_width (largura)
- layout_height (altura)

A forma como esses atributos são configurados influencia diretamente:

- o tamanho do componente
- sua posição na tela
- e o comportamento do layout como um todo

O Android oferece diferentes formas de definir essas dimensões, permitindo desde tamanhos fixos até comportamentos dinâmicos.

Dimensões fixas com dp

Quando se fala em altura e largura de componentes a primeira ideia que pode ocorrer ao desenvolvedor Android iniciante é especificá-las usando a unidade **dp**.

É possível fazer isso e neste caso o tamanho do componente será fixo. Trata-se de uma abordagem simples, que não requer maiores explicações e nesse caso teremos:

- o componente terá um tamanho previsível
- independente da densidade da tela

Isso será útil quando:

- o design exige dimensões específicas
- é necessário manter proporções bem definidas

Porém, existem outras forma de especificar largura e altura que contribuem para um layout mais dinâmico e responsivo.

Opção *wrap_content*

O valor *wrap_content* faz com que o componente ocupe apenas o espaço necessário para exibir seu conteúdo.

Como fica no XML:

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
```

Nesse caso ocorrerá:

- um TextView terá largura suficiente para exibir o texto;
- um ImageView terá tamanho suficiente para exibir a imagem;
- um Button se ajustará ao tamanho do seu rótulo.

e se um desses conteúdos for alterado na execução do aplicativo, o componente terá seu tamanho redimensionado dinamicamente para acomodar o novo conteúdo.

Esta opção é útil quando o tamanho do conteúdo é variável ou desconhecido.

Opção *match_parent*

O valor *match_parent* faz com que o componente ocupe todo o espaço disponível no elemento pai, na dimensão especificada.

Como fica no XML:

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

Como a configuração de largura e altura são independentes entre si, nesse caso ocorrerá:

- o componente se estende completamente na largura do container;
- a altura continua ajustada ao conteúdo.

Faz com que o componente ocupe todo o espaço disponível dentro do contêiner, na dimensão para a qual foi configurado.

Opção *match_constraint* (0dp – apenas para ConstraintLayout)

No ConstraintLayout, existe um comportamento específico equivalente ao preenchimento de espaço disponível, chamado de *match_constraint*.

Esse comportamento é ativado ao definir a dimensão como 0dp.

Como fica no XML:

```
android:layout_width="0dp"
```

Nesse caso:

- o tamanho do elemento será determinado pelas vínculos definidos para o componente;
- o componente se ajusta ao espaço disponível entre as restrições.

É importante ressaltar que isso só funciona corretamente quando as restrições de ambos os lados sejam especificadas:

- na largura é necessário especificar os vínculos start e end;
- na altura é necessário especificar os vínculos top e bottom.

2.8. Tarefas

Tarefa 1.

No Android Studio reproduza o aspecto visual do App 01 - Aplicativo Boas Vindas

O pdf deste aplicativo está disponível no site do professor.

Tarefa 2.

No Android Studio reproduza o aspecto visual do App 02 - Aplicativo Registro de Mensagens

O pdf deste aplicativo está disponível no site do professor.

Tarefa 3.

Usando o Figma - https://www.figma.com/pt-br - elabore um layout para o Aplicativo de Conversão de Unidades especificado no documento da Aula 06, disponível no SIGA.

Em seguida, no Android Studio inicie um novo projeto e crie o layout elaborado no item anterior.

Capítulo 3 – Funcionalidades Iniciais em aplicativos Android

3.1. Introdução

Até aqui, você aprendeu a estruturar interfaces utilizando arquivos XML e a organizar componentes dentro de layouts. No entanto, aplicativos Android não são estáticos — eles reagem às ações do usuário.

Neste capítulo, você dará o próximo passo: aprenderá **a interagir com os elementos da interface por meio de código**, tornando sua aplicação dinâmica e responsiva.

3.2. A Classe Activity e o Método onCreate()

A classe Activity

Toda tela de um aplicativo Android é representada por uma classe chamada `Activity`. É nela que o código responsável pela lógica da interface é executado.

O ponto de partida dessa lógica é o método `MainActivity`, conforme mostrado neste código:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

}
```

Código 3.1 – método onCreate()

Podemos verificar que a classe `MainActivity` herda a partir de `AppComatActivity`.

É importante conhecer essa hierarquia de classes, dada pela seguinte sequência:

1. `java.lang.Object`
2. **`android.content.Context`**
3. `android.content.ContextWrapper`
4. `android.view.ContextThemeWrapper`
5. **`android.app.Activity`** (A classe base nativa das Activities)
6. `androidx.core.app.ComponentActivity` (Gerencia o ciclo de vida e componentes)
7. `androidx.fragment.app.FragmentActivity` (Adiciona suporte a Fragmentos)
8. `androidx.appcompat.app.AppCompatActivity` (Adiciona suporte à ActionBar/Temas)

Ao longo dessa hierarquia as duas classes em negrito se destacam, por sua importância:

- `Context` é usada como argumento em inúmeros métodos que exigem a passagem de referências a activities;
- `Activity` é a espinha dorsal que define os recursos que as activities (telas) de um aplicativo possuem.

Será comum neste texto usarmos de forma geral a palavra **activity** quando nos referirmos à nossa classe específica `MainActivity`, ou qualquer outra classe que herde de `Activity`.

O Método onCreate()

Vamos agora abordar o método `onCreate()` que pode ser visto no código 3.1.

Este método é o ponto de inicialização de uma activity no Android. Ele é executado uma única vez quando a activity é criada.

De um modo geral é neste método que são realizadas as inicializações da tela, necessárias para o seu correto funcionamento. As tarefas típicas feitas no `onCreate()` são:

- definir o layout da interface com `setContentView()`;
- inicializar componentes da interface, como botões, textos, etc.;
- configurar eventos como cliques;
- construir objetos em geral, necessários ao funcionamento da activity;
- preparar quaisquer dados necessários.

Em síntese, o `onCreate()` pode ser comparado ao momento em que um palco é montado antes de um espetáculo: é nele que a interface é definida, os elementos são posicionados e tudo é preparado para que o usuário possa interagir com a tela.

3.3. Referenciando Componentes com findViewById()

Para interagir com um componente (como um botão ou campo de texto), primeiro precisamos obter uma referência a ele no código.

Pense na situação:

1. você tem um botão definido no arquivo XML, portanto ele é visível na tela;
2. você quer fazer com que um toque no botão (clique) provoque uma ação a ser executada no código Java do aplicativo.

O item 1 é resolvido no XML que conterá um componente `Button`, desta forma:

```
<Button
    android:id="@+id/btnEnviar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Enviar" />
```

Note o atributo `android:id`. Ele é fundamental neste uso, pois é usado para identificar de forma única uma View dentro do layout.

Ele funciona como um “nome” ou “etiqueta” que permite que o programador encontre e manipule aquele componente no código Java ou Kotlin.

Agora podemos abordar o item 2. Veja o trecho de código Java:

```
Button btnEnviar = findViewById(R.id.btnEnviar);
```

Neste código está sendo feita a declaração e inicialização do objeto `btnEnviar`.

O método `findViewById()` é o responsável pela construção (uso do comando new) e inicialização do objeto. Nessa inicialização este objeto é vinculado à `View` que possui o id `R.id.btnEnviar`.

A partir disso, o objeto `btnEnviar` poderá ser usado pelo programador ao longo de todos os métodos da classe `MainActivity`.

O que se pode fazer depois?

Uma vez que temos a referência ao objeto, podemos interagir com as propriedades do componente de diversas formas, como mostrado neste exemplo:

```
btnEnviar.setText("Clique aqui");          // altera o texto do botão
btnEnviar.setEnabled(false);               // desabilita o botão
btnEnviar.setVisibility(View.INVISIBLE);   // torna o botão invisível
```

3.4. A Classe R no Android

Talvez você esteja curioso para compreender o que é o identificador `R.id.btnEnviar` usado na seção anterior.

Pois bem, trata-se da identificação (id) do botão. Mas e o que é `R` ?

Durante o desenvolvimento de aplicações Android, é comum acessar layouts, imagens, textos e outros recursos diretamente pelo código. Isso é possível graças à classe `R`.

A classe `R` é uma classe especial gerada automaticamente pelo sistema de compilação (*build*) do Android (Gradle). Ela funciona como um índice central de todos os recursos definidos no projeto, permitindo que eles sejam acessados de forma segura e organizada no código Java ou Kotlin.

Em outras palavras, sempre que você cria um recurso – como um layout XML, uma string, uma cor, uma imagem, os id's das Views nos XML, etc – o Android Studio automaticamente registra esse recurso dentro da classe `R`, atribuindo a ele um identificador único.

Considere o seguinte exemplo:

```
<Button
    android:id="@+id/btnEnviar"
    android:text="Enviar"/>
```

Ao compilar o projeto, o Android gera automaticamente uma referência equivalente na classe R:

```
R.id.btnEnviar
```

Essa referência é o que permite acessar o componente no código:

```
Button btn = findViewById(R.id.btnEnviar);
```

Estrutura da Classe R

A classe R é organizada em subclasses estáticas, que agrupam os recursos por tipo. Cada uma dessas categorias corresponde a uma pasta dentro do diretório res.

A seguir, as principais categorias:

- R.layout - Contém referências para arquivos de layout XML.

  Exemplo:

  ```
  R.layout.activity_main
  ```

  Pasta de Origem: res/layout/
- R.id - Armazena os identificadores de componentes de interface.

  Exemplo:

  ```
  R.id.btnEnviar
  ```

  Pasta de Origem: atributos android:id nos XMLs
- R.string - Contém textos definidos como recursos.

  Exemplo:

  ```
  R.string.app_name
  ```

  Pasta de Origem: res/values/strings.xml
- R.drawable - Referência imagens e elementos gráficos.

  Exemplo:

```
R.drawable.logo
```

Pasta de Origem: res/drawable/

- R.color - Define cores utilizadas no aplicativo.

  Exemplo:

```
R.color.primaryColor
```

  Pasta de Origem: res/values/colors.xml

A classe `R` é o elo entre os arquivos de recursos e o código da aplicação. Compreender seu funcionamento é essencial para desenvolver aplicações Android de forma organizada, segura e eficiente.

Boas Práticas

- Nomear recursos de forma clara e padronizada (btnEnviar, txtNome, imgFoto, etc);
- Centralizar textos em strings.xml;
- Centralizar cores em colors.xml;
- Evitar duplicação de recursos;
- Não modificar a classe R manualmente (nas versões antigas do Android Studio isso era possível, nas atuais não é possível).

3.5. Eventos de Clique – implementação de OnClickListener

Uma tarefa muito comum em aplicativos é reagir ao clique de um botão ou de qualquer outro tipo de `View`.

O `OnClickListener` é uma interface utilizada no Android para responder a eventos de clique em elementos da interface, como botões, imagens e outros componentes interativos, que herdam da classe `View`.

Em outras palavras, ele permite definir o que deve acontecer quando o usuário toca em uma `View`.

Como funciona?

No Android, eventos de interação são tratados por meio de listeners (ouvintes).

O `OnClickListener` é um desses ouvintes, responsável por detectar cliques.

Quando um usuário clica em uma `View`:

1. O sistema identifica o evento;
2. Verifica se há um `OnClickListener` associado;
3. Executa o método `onClick()` implementado dentro do listener.

Exemplo completo em Java

```
public class MainActivity extends AppCompatActivity {

    private Button botaoMensagem;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Ligação com o componente do XML
        botaoMensagem = findViewById(R.id.botaoMensagem);

        // Definição do evento de clique
        botaoMensagem.setOnClickListener(new View.OnClickListener() {
```

```
            @Override
            public void onClick(View view) {
                // Ação ao clicar no botão
                Toast.makeText(MainActivity.this, "Botão clicado!",
                               Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Explicação passo a passo

1. setContentView(...) – Define qual layout será exibido na tela.
2. findViewById(...) – Faz a ligação entre o componente do XML e o código Java.
3. setOnClickListener(...) – Associa um listener ao botão, ou seja, define quem "ouve" o clique.
4. Método onClick(View view) – Contém a lógica executada quando o usuário clica no botão.
5. Toast.makeText(...) – Exibe uma mensagem rápida na tela (feedback ao usuário).

Resumo até aqui

O fluxo básico para implementar o `OnClickListener` é:

1. Criar o botão no XML com um id;
2. Criar um objeto para o botão (ou outra View) e referenciá-lo com `findViewById`;
3. Definir o comportamento com `setOnClickListener`;

3.6. Aprofundamento sobre OnClickListener

Imagine uma situação em que é necessário implementar `OnClickListener` em múltiplos botões.

É claro que sempre será possível fazer a implementação um a um, porém existe uma outra forma de fazer que pode ser bem interessante do ponto de vista da organização e legibilidade do código.

Essa opção é implementar a interface `OnClickListener` na própria classe `MainActivity`.

Isso permite a centralização do tratamento dos cliques em um único lugar.

Em síntese:

em vez de criar um listener separado para cada botão, a própria activity implementa a interface `View.OnClickListener`.

Com isso:

- todos os botões compartilham o mesmo método `onClick()`;
- você diferencia qual botão foi clicado usando o id;

Exemplo prático:

Considere que o arquivo XML contém três botões – btnSomar, btnSubtrair, btnMultiplicar.

Disso, podemos escrever o código Java da seguinte forma:

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    Button btnSomar, btnSubtrair, btnMultiplicar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        // Associação com os componentes do XML
        btnSomar = findViewById(R.id.btnSomar);
        btnSubtrair = findViewById(R.id.btnSubtrair);
        btnMultiplicar = findViewById(R.id.btnMultiplicar);

        // Definindo o listener (a própria Activity)
        btnSomar.setOnClickListener(this);
        btnSubtrair.setOnClickListener(this);
        btnMultiplicar.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {
        if (view.getId() == R.id.btnSomar)
            Toast.makeText(this, "Você clicou em Somar", Toast.LENGTH_SHORT).show();
        else if (view.getId() == R.id. btnSubtrair)
            Toast.makeText(this, "Você clicou em Subtrair", Toast.LENGTH_SHORT).show();
        else if (view.getId() == R.id. btnMultiplicar)
            Toast.makeText(this, "Você clicou em Multiplicar", Toast.LENGTH_SHORT).show();
    }
}
```

E agora vamos descrever este código, passo a passo:

1. A classe implementa a interface

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
```

Isto significa que a classe `MainActivity` se compromete a implementar todos os métodos declarados nessa interface, e a partir daí ela pode ser usada como uma instância daquele comportamento.

2. Associação dos botões no onCreate()

```
Button btnSomar = findViewById(R.id.btnSomar);
Button btnSubtrair = findViewById(R.id.btnSubtrair);
Button btnMultiplicar = findViewById(R.id.btnMultiplicar);
```

A esta altura já vimos para que serve o `findViewById()`.

3. Definição do listener para todos

```
btnSomar.setOnClickListener(this);
btnSubtrair.setOnClickListener(this);
btnMultiplicar.setOnClickListener(this);
```

E aqui está o conceito chave. Ao usar `this` como argumento para `setOnClickListener(...)`, onde é esperado um listener, estamos admitindo que a própria classe `MainActivity` tem comportamento de um listener.

4. Implementação do método onClick()

```
    @Override
    public void onClick(View view) {
        ...
        if (view.getId() == R.id.btnSomar)
            Toast.makeText(this, "Você clicou em Somar", Toast.LENGTH_SHORT).show();
        else if (view.getId() == R.id. btnSubtrair)
            Toast.makeText(this, "Você clicou em Subtrair", Toast.LENGTH_SHORT).show();
        else if (view.getId() == R.id. btnMultiplicar)
            Toast.makeText(this, "Você clicou em Multiplicar", Toast.LENGTH_SHORT).show();
    }
```

Este método centraliza o tratamento dos cliques dos botões da interface.

O parâmetro `View view` representa a `View` que disparou o evento, ou seja, o componente que foi clicado pelo usuário.

Através do método `getId()`, é possível obter o identificador desse componente e, com base nele, distinguir qual botão foi acionado, permitindo executar ações específicas para cada caso dentro de um único método.

Resumo deste aprofundamento

A implementação do `OnClickListener` diretamente na activity consiste em fazer com que a própria classe se torne responsável por tratar os eventos de clique dos componentes da interface.

Para isso, a activity implementa a interface `View.OnClickListener`, assumindo o compromisso de definir o método `onClick(View view)`, que será chamado automaticamente sempre que uma View associada receber um clique.

No método `onCreate()`, os componentes da interface (como botões) são obtidos por meio do `findViewById()` e configurados para utilizar a própria activity como listener, através da chamada `setOnClickListener(this)`. Dessa forma, todos os eventos de clique são direcionados para um único ponto de controle.

Já no método `onClick(View view)`, o parâmetro recebido representa a `View` que disparou o evento. A partir do método `getId()`, é possível identificar qual componente foi acionado e, com isso, definir comportamentos específicos para cada caso. Essa abordagem permite centralizar o tratamento dos eventos, reduzindo a repetição de código e facilitando a organização em aplicações com múltiplos elementos interativos.

3.7.Tarefas

Tarefa 1.

No Android Studio implemente o código do App 01 - Aplicativo Boas Vindas

O pdf deste aplicativo está disponível no site do professor.

Tarefa 2.

No Android Studio implemente o código do App 02 - Aplicativo Registro de Mensagens

O pdf deste aplicativo está disponível no site do professor.

Tarefa 3.

No Android Studio implemente o código do aplicativo de Conversão de Unidades especificado no documento da Aula 06, disponível no SIGA.