

UNIP

UNIVERSIDADE PAULISTA

Desenvolvimento para Dispositivos Móveis

Autor: Prof. Olavo Tomohisa Ito

Colaboradoras: Profa. Vanessa Santos Lessa

Profa. Larissa Rodrigues Damiani

Professor conteudista: Olavo Tomohisa Ito

Mestre em Engenharia de Produção pela UNIP, bacharel em Física pelo Instituto de Física da USP, licenciado em Ciências pela Faculdade de Educação da USP e bacharel em Língua e Literatura Japonesa da FFLCH-USP. Possui experiência com linguagem Fortran e teve contato com um dos dois únicos computadores IBM 360 remanescentes no mundo, pertencente ao laboratório do acelerador de partículas Pelletron. Também teve a oportunidade de trabalhar com um Itautec I-7000, que tinha tanto poder de processamento quanto o gigantesco IBM360 e permitia o uso de outras linguagens de programação além de Fortran: Basic, xBase, C, Pascal, Algol, Cobol e muitas outras. Professor dos cursos de graduação de Sistemas de Informação e Ciência da Computação, bem como do curso superior tecnológico de Análise e Desenvolvimento de Sistemas. Das disciplinas que ministra, *Programação de Dispositivos Móveis* é especial, pois durante muitos anos atuou no ensino de programação para crianças e jovens desde a pré-escola com Scratch Jr. até o Ensino Médio com o próprio App Inventor, entre outros. Em 2015, recebeu a certificação Mobile Computing with App Inventor – CS Principles pelo Trinity College – Connecticut.

Dados Internacionais de Catalogação na Publicação (CIP)

I89d Ito, Olavo Tomohisa.

Desenvolvimento para Dispositivos Móveis / Olavo Tomohisa Ito.
– São Paulo: Editora Sol, 2025.

336 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. App. 2. Android. 3. Projetos. I. Título.

CDU 681.3.06

U521.42 – 25

Prof. João Carlos Di Genio
Fundador

Profa. Sandra Rejane Gomes Miessa
Reitora

Profa. Dra. Marília Ancona Lopez
Vice-Reitora de Graduação

Profa. Dra. Marina Ancona Lopez Soligo
Vice-Reitora de Pós-Graduação e Pesquisa

Profa. Dra. Claudia Meucci Andreatini
Vice-Reitora de Administração e Finanças

Profa. M. Marisa Regina Paixão
Vice-Reitora de Extensão

Prof. Fábio Romeu de Carvalho
Vice-Reitor de Planejamento

Prof. Marcus Vinícius Mathias
Vice-Reitor das Unidades Universitárias

Profa. Silvia Renata Gomes Miessa
Vice-Reitora de Recursos Humanos e de Pessoal

Profa. Laura Ancona Lee
Vice-Reitora de Relações Internacionais

Profa. Melânia Dalla Torre
Vice-Reitora de Assuntos da Comunidade Universitária

UNIP EaD

Profa. Elisabete Brihy
Profa. M. Isabel Cristina Satie Yoshida Tonetto

Material Didático

Comissão editorial:

Profa. Dra. Christiane Mazur Doi
Profa. Dra. Ronilda Ribeiro

Apoio:

Profa. Cláudia Regina Baptista
Profa. M. Deise Alcantara Carreiro
Profa. Ana Paula Tôrres de Novaes Menezes

Projeto gráfico:

Prof. Alexandre Ponzetto

Revisão:

Kleber Souza
Maria Cecília França

Sumário

Desenvolvimento para Dispositivos Móveis

APRESENTAÇÃO	9
INTRODUÇÃO	10

Unidade I

1 INTRODUÇÃO AOS APPS MÓVEIS	13
1.1 Conceitos	13
1.2 Histórico	13
1.2.1 As bases da comunicação móvel	13
1.2.2 Era dos celulares	14
1.2.3 Smartphones	14
1.2.4 Sistemas operacionais e tecnologias de banda móvel	14
1.3 Aplicação	15
1.3.1 Distribuição de apps móveis	16
1.3.2 Ferramentas para o desenvolvimento de apps móveis	16
1.3.3 Características	18
2 CONCEITOS DE ORIENTAÇÃO A OBJETO	19
2.1 Programação estrutural	20
2.1.1 Variáveis	20
2.1.2 Tipos	21
2.1.3 Operações	23
2.1.4 Repetição	30
2.2 Objeto	39
2.3 Classe	40
2.4 Atributo	42
2.5 Métodos	43
2.6 Herança	44
2.7 Polimorfismo	45
3 INTRODUÇÃO AO ANDROID	45
3.1 Infraestrutura Android	48
3.2 Plataforma de desenvolvimento	52
3.2.1 Ambiente de Desenvolvimento Integrado (IDE)	52
3.2.2 Linguagens de programação	53
3.2.3 Bibliotecas e frameworks	53
4 ACTIVITY	54
4.1 Intents e intent filters	55
4.2 Componentes	57

4.3 Ciclo de vida	58
4.4 Layout	60
4.5 Widgets.....	61
4.6 Temas.....	62

Unidade II

5 ANDROID STUDIO.....	68
5.1 Instalação	69
5.2 Iniciando um projeto	73
5.3 Interface.....	79
5.3.1 Estrutura do projeto	82
5.4 Emulador.....	83
5.4.1 Espelhar um dispositivo físico	83
5.4.2 Espelhar via AVD	86
5.4.3 Executar via AVD.....	88
6 INTERFACES GRÁFICAS	90
6.1 Iniciando o projeto com views.....	90
6.1.1 Gradle	92
6.2 Interface.....	92
6.2.1 MainActivity.kt.....	93
6.2.2 activity_main.xml	97
6.3 Views	108
6.3.1 ListView	116
6.3.2 TextView (visualização de texto).....	120
6.3.3 Menu	123
6.3.4 AlertDialog.....	131
6.3.5 LinearLayout	133
6.3.6 RelativeLayout	135
6.3.7 TableLayout	137
6.4 Jetpack Compose.....	140
6.4.1 Introdução ao Jetpack Compose	146
6.4.2 Material Design	147
6.4.3 Widgets.....	149
6.4.4 Fundamentos dos layouts	152
6.4.5 Composables.....	152
6.4.6 Principais layouts em Compose	153
6.4.7 Modificadores de layout	177
6.4.8 Gerenciamento de espaço e alinhamento	194

Unidade III

7 PROJETOS	204
7.1 Projeto de um app	204
7.1.1 Criar o projeto na plataforma Android	208
7.1.2 Registrar o projeto	211

7.1.3 Implementar as activities	219
7.2 Protótipo de um jogo	267
7.2.1 Criar o cenário do jogo.....	270
7.2.2 Desenhar o objeto principal.....	284
7.2.3 Mover o objeto através do comando do usuário	295
7.2.4 Criar obstáculos.....	306
7.2.5 Colisões do objeto com obstáculos.....	311
7.2.6 Adicionar placar	315
8 PUBLICAÇÃO DE APPS E JOGOS.....	324
8.1 Gerando o app instalável	324
8.2 Informações e especificações técnicas dos detalhes do app	328
8.3 Versões do app.....	329
8.4 Classificação do app.....	329
8.5 Preços e distribuição	329
8.6 Conta de desenvolvedor	329

APRESENTAÇÃO

Caro aluno,

Desenvolvimento para Dispositivos Móveis é a disciplina de boas-vindas. Ela é acessível no mundo da programação e introduz a programação dos dispositivos móveis.

A computação móvel hoje está mais presente no cotidiano do que a computação pessoal, estando praticamente o dia todo na vida das pessoas. Desta forma, nada melhor do que uma disciplina que reúna a praticidade do dispositivo versátil, com a aprendizagem de conceitos de tecnologias computacionais.

O curso se baseia no Android Studio, uma ferramenta completa fornecida pelo Google para criar aplicativos. Ela foca somente o necessário em lógica de programação, paradigmas de linguagens e em aspectos práticos na montagem de aplicativos simples para compreender o processo de desenvolvimento.

A linguagem de desenvolvimento inicialmente no Android Studio era o Java, mas o Google, desenvolvedor do sistema operacional Android, está adotando o Kotlin e o Compose como tecnologias principais para o desenvolvimento Android. Isso moderniza e simplifica o processo de criação de aplicativos, oferecendo aos desenvolvedores ferramentas eficientes e intuitivas, facilitando a sua aprendizagem.

INTRODUÇÃO

A integração dos smartphones em nosso dia a dia é tão natural que mal percebemos a revolução tecnológica ocorrida. Tarefas que antes exigiam a utilização de computadores de mesa complexos, como a comunicação por mensagens instantâneas, agora são realizadas de forma intuitiva em dispositivos móveis. A popularização dos smartphones impulsionou a demanda por profissionais capazes de desenvolver aplicativos móveis.

Este livro-texto contempla aspectos da arquitetura antes utilizados pelo Java e o novo kit de ferramentas Jetpack Compose, que utiliza uma abordagem declarativa, simplificando a construção de Interfaces de Usuário (UIs) complexas.

Na primeira unidade, você conhecerá o universo do desenvolvimento de aplicativos móveis. Aprenderá a configurar os ambientes de desenvolvimento, compreendendo sua evolução histórica e os conceitos fundamentais que regem sua utilização. Para uma compreensão aprofundada do desenvolvimento de aplicativos móveis, é fundamental dominar os conceitos da programação orientada a objetos.

Serão explorados os pilares da programação orientada a objetos (POO), como classes, objetos e seus relacionamentos, e como esses conceitos são aplicados na construção de aplicativos. Aprofundaremos os conhecimentos, aplicando-os ao desenvolvimento de aplicativos Android com introdução ao sistema operacional Android, explorando sua arquitetura e a ferramenta para desenvolvimento de aplicativos.

A unidade será concluída apresentando o componente fundamental dos aplicativos Android: a activity. Serão explorados os conceitos de intents (mensagens que permitem a comunicação entre componentes), intent filters (mecanismos para definir quais intents uma activity pode responder) e ciclo de vida de uma activity (estados que uma activity pode assumir durante seu tempo de execução). Além disso, serão abordados os conceitos de layout (organização visual dos elementos na tela), widgets (elementos visuais que compõem a interface do usuário) e temas (estilização da aparência do aplicativo).

Na segunda unidade estudaremos as técnicas de desenvolvimento de interfaces gráficas para aplicativos móveis, com ênfase em duas abordagens principais: a abordagem tradicional com XML e a abordagem inovadora e declarativa do Jetpack Compose. A construção de interfaces utilizando XML será explorada por meio de componentes como ListView, TextView, Menu, AlertDialog, LinearLayout, RelativeLayout e TableLayout. Isso proporciona uma base sólida para a criação de layouts flexíveis e responsivos. Na sequência, o foco será no Jetpack Compose, uma ferramenta moderna que simplifica o processo de construção de interfaces, um toolkit moderno para a construção de interfaces declarativas. Serão abordados componentes como column, row, box, surface, scaffold, além de restrições e modificadores, que permitem a criação de interfaces mais intuitivas e personalizadas. Ao final desta unidade, você estará apto a desenvolver interfaces de usuário eficientes e modernas, utilizando as melhores ferramentas e técnicas disponíveis no Android Studio.

Na terceira unidade, os conhecimentos adquiridos nas unidades anteriores serão aplicados na construção de projetos práticos. Serão desenvolvidos um aplicativo e um jogo, explorando diferentes aspectos do desenvolvimento para Android. O aplicativo servirá como base para a aplicação dos conceitos de interfaces, atividades e intents, enquanto o jogo permitirá a exploração de conceitos mais avançados como gráficos, animações e física. A unidade culminará com a publicação do aplicativo na Google Play Store, preparando-o para compartilhar seus projetos com o mundo.

Unidade I

A evolução dos dispositivos móveis transformou a forma como nos comunicamos e interagimos. Nesta unidade, traçaremos um breve histórico dessa evolução, desde os primeiros celulares até os smartphones modernos.

1 INTRODUÇÃO AOS APPS MÓVEIS

1.1 Conceitos

Dispositivos móveis são aparelhos eletrônicos portáteis que permitem a comunicação, o acesso à internet e a execução de diversas aplicações. Exemplos incluem smartphones, tablets e smartwatches. Eles são caracterizados pela mobilidade, conectividade sem fio e capacidade de executar uma ampla gama de funções, desde chamadas telefônicas até navegação na web e uso de aplicativos.

Tipos de dispositivos móveis:

- **Smartphones:** são os mais comuns, com funcionalidades avançadas de comunicação, navegação na web, câmera e um vasto ecossistema de aplicativos.
- **Tablets:** dispositivos móveis com telas maiores que os smartphones, voltados para consumo de mídia, leitura e produtividade.
- **Smartwatches:** relógios inteligentes que, além de mostrar as horas, oferecem funcionalidades como monitoramento de saúde, notificações e conexão com smartphones.
- **E-readers:** dispositivos projetados para a leitura de e-books, como o Kindle.

1.2 Histórico

A história da comunicação móvel é uma jornada incrível que mudou completamente a forma como nos conectamos. Desde os primeiros rádios portáteis até os poderosos smartphones atuais, essa evolução tecnológica impactou nossas vidas.

1.2.1 As bases da comunicação móvel

Tudo começou com os rádios portáteis, aqueles aparelhos grandes e simples que permitiam a comunicação em curtas distâncias. Imagine um grupo de exploradores em uma expedição: com esses rádios, eles podiam se comunicar sem precisar gritar. A polícia e os bombeiros logo perceberam o potencial desses dispositivos e os adotaram em suas operações.

1.2.2 Era dos celulares

Os primeiros dispositivos móveis surgiram na década de 1970. Logo avançaram e os celulares surgiram, revolucionando a comunicação móvel. Os primeiros modelos eram grandes e pesados, mas com o objetivo de fazer ligações a qualquer hora e em qualquer lugar. Em 1973, Martin Cooper, engenheiro da Motorola, fez a primeira chamada de um telefone celular, marcando o início da comunicação móvel.

Nos anos 1980 e 1990, os telefones celulares começaram a se popularizar. As redes de telefonia móvel se expandiram, cobrindo mais áreas. Inicialmente, os aparelhos eram analógicos, grandes e com funções limitadas, como fazer chamadas e enviar mensagens de texto. Com o tempo, os celulares digitais surgiram, trazendo recursos como agendas eletrônicas e mensagens de texto mais avançadas.

1.2.3 Smartphones

A chegada dos smartphones marcou a história da comunicação móvel no início dos anos 2000. Empresas como BlackBerry e Nokia lançaram dispositivos que faziam ligações, enviavam mensagens, acessavam a internet, tiravam fotos e vídeos de alta qualidade, com games, aplicativos de navegação e muito mais.

Foi o lançamento do iPhone pela Apple em 2007 que revolucionou o mercado. O iPhone introduziu uma interface de usuário inovadora com tela sensível ao toque e uma vasta gama de aplicativos transformando os smartphones em computadores de bolso.

1.2.4 Sistemas operacionais e tecnologias de banda móvel

A história da comunicação móvel está intrinsecamente ligada à evolução dos sistemas operacionais e das tecnologias de banda. Cada avanço nesses dois pilares impulsionou a capacidade dos dispositivos móveis e a forma como nos conectamos.

Os sistemas operacionais são os softwares que gerenciam o hardware do dispositivo e fornecem a interface com o usuário. Nos dispositivos móveis, alguns sistemas operacionais se destacaram:

- Sistemas operacionais proprietários:
 - **Palm OS:** um dos primeiros sistemas operacionais para PDAs, popular nos anos 1990.
 - **Symbian:** um dos primeiros sistemas operacionais móveis amplamente utilizados, especialmente em dispositivos da Nokia. Ele oferecia suporte para câmeras, MMS, Bluetooth e outras funções multimídia. Apesar de sua popularidade inicial, foi gradualmente substituído pelo Android e pelo iOS.
- Sistemas operacionais abertos:
 - **Android:** desenvolvido pelo Google e lançado em 2008, rapidamente se tornou o sistema operacional móvel mais utilizado no mundo. Sua natureza aberta e capacidade de personalização

o tornaram popular entre fabricantes e desenvolvedores. O Android suporta uma ampla variedade de dispositivos e oferece uma vasta gama de aplicativos.

- **Windows Mobile e Windows Phone:** a Microsoft entrou no mercado com o Windows Mobile, que mais tarde evoluiu para o Windows Phone. Esses sistemas eram conhecidos por sua integração com outros produtos da Microsoft, como o Office e o Outlook. No entanto, enfrentaram dificuldades para competir com Android e iOS.
- **iOS:** lançado pela Apple em 2007, revolucionou o mercado com a introdução do iPhone. Ele trouxe uma interface de usuário intuitiva e vasta gama de aplicativos. O iOS é conhecido por sua segurança e desempenho consistente, sendo exclusivo para dispositivos Apple.

Neste histórico é necessário incorporar as tecnologias de banda fundamentais para a comunicação móvel, pois determinam a velocidade e a capacidade de transmissão de dados, influenciando no potencial dos aplicativos criados. As tecnologias são batizadas com o número seguido da letra G.

- **1G:** a primeira geração de redes móveis era analógica e limitada a chamadas de voz.
- **2G:** introduziu a comunicação digital e permitiu o envio de mensagens de texto (SMS).
- **3G:** marcou um grande avanço, com a transmissão de dados em velocidades mais altas, permitindo o acesso à internet móvel, download de arquivos de videoconferências e streaming de áudio e vídeo (mesmo com alguma latência ou interrupções).
- **4G:** trouxe velocidades ainda maiores e menor latência, permitindo uma experiência de internet móvel comparável à banda larga fixa. Aplicativos como streaming de vídeo em alta definição, jogos online e videoconferências se tornaram comuns.
- **5G:** a quinta geração promete velocidades ainda mais rápidas, menor latência e maior capacidade. Isso não somente melhora a experiência do usuário em dispositivos móveis, mas também abre caminho para inovações como carros autônomos, cidades inteligentes, Internet das Coisas (IoT) e realidade virtual.

1.3 Aplicação

Dispositivos móveis servem para uma variedade de funções, facilitando a comunicação, o acesso à informação e o entretenimento.

Aqui estão algumas das suas principais aplicações:

- **Comunicação:** dispositivos móveis facilitam a comunicação instantânea por meio de chamadas, mensagens de texto e aplicativos de mensagens. Exemplos: WhatsApp, Telegram, Signal.
- **Navegação e geolocalização:** aplicativos que utilizam GPS para fornecer direções e informações sobre a localização. Exemplos: Google Maps, Waze.

- **E-commerce:** plataformas de compra e venda que permitem transações financeiras via dispositivos móveis. Exemplos: Amazon, eBay, Shopify.
- **Entretenimento e mídia:** aplicativos que fornecem acesso a jogos, filmes, músicas e redes sociais. Exemplos: Netflix, Spotify, TikTok.
- **Saúde e bem-estar:** aplicativos que monitoram a saúde, oferecem dicas fitness e conectam pacientes a profissionais de saúde. Exemplos: MyFitnessPal, Fitbit, telemedicina.
- **Educação e aprendizado:** plataformas educacionais que oferecem cursos e recursos de aprendizado via dispositivos móveis. Exemplos: Duolingo, Coursera, Khan Academy.
- **Pagamentos móveis:** soluções que permitem a realização de pagamentos e transferências financeiras por meio de dispositivos móveis. Exemplos: Apple Pay, Google Pay, Venmo.
- **Automação residencial:** serve de controle de dispositivos domésticos por meio de aplicativos móveis, promovendo conveniência e eficiência. Exemplos: Google Home, Amazon Alexa.
- **Redes sociais:** aplicativos que permitem a interação social, o compartilhamento de conteúdo e o networking. Exemplos: Facebook, Instagram, X.
- **Realidade aumentada (AR) e realidade virtual (VR):** aplicações que oferecem experiências imersivas, combinando elementos virtuais com o mundo real. Exemplos: Pokémon GO, Google Cardboard.

1.3.1 Distribuição de apps móveis

Os sistemas operacionais móveis possuem lojas de aplicativos digitais, que funcionam como mercados virtuais em que desenvolvedores disponibilizam seus aplicativos. Nesses ambientes, os usuários podem navegar, pesquisar e instalar aplicativos de acordo com suas necessidades.

A App Store (2008) é a loja de aplicativos da Apple, que revolucionou a distribuição de software móvel. O Google Play (2012) é a loja de aplicativos do Android, proporcionando variedade de aplicativos para usuários.

1.3.2 Ferramentas para o desenvolvimento de apps móveis

Existem diversas ferramentas de desenvolvimento de aplicativos móveis que atendem a diferentes necessidades e níveis de habilidade. A ferramenta tradicional de desenvolvimento é um software ou ambiente de desenvolvimento integrado (IDE) que permite ao desenvolvedor criar aplicativos nativos ou multiplataforma usando linguagens de programação específicas. Essas ferramentas oferecem uma ampla gama de funcionalidades, incluindo editores de código, depuradores, emuladores e ferramentas de design de interface. Aqui estão alguns exemplos:

- **Android Studio:** a principal ferramenta para desenvolvimento de aplicativos Android. Oferece um ambiente de desenvolvimento integrado (IDE) completo com suporte para Kotlin e Java, além de ferramentas de depuração e emulação.
- **Xcode:** utilizado para desenvolvimento de aplicativos iOS. Esse IDE da Apple suporta Swift e Objective-C, incluindo um simulador de dispositivos, ferramentas de depuração e um editor de interface gráfica.
- **Flutter:** framework de código aberto do Google para desenvolvimento de aplicativos multiplataforma. Utiliza a linguagem Dart e permite criar aplicativos nativos para Android e iOS com uma única base de código.
- **React Native:** desenvolvido pelo Facebook, esse framework permite criar aplicativos móveis usando JavaScript e React. É popular por sua capacidade de compartilhar código entre as plataformas Android e iOS.

As ferramentas No-Code são plataformas que permitem a criação de aplicativos sem a necessidade de escrever código. Elas utilizam interfaces visuais, como arrastar e soltar, para facilitar o processo de desenvolvimento, tornando-o acessível a pessoas sem experiência em programação. Aqui estão alguns exemplos dessas ferramentas:

- **Appgyver:** plataforma No-Code que permite criar aplicativos móveis e web sem a necessidade de escrever código. Ideal para usuários que não têm experiência em programação.
- **Adalo:** ferramenta No-Code que facilita a criação de aplicativos móveis por meio de uma interface de arrastar e soltar. Permite a integração com diversas APIs (interfaces de programação de aplicativos) e serviços externos.
- **Bubble:** embora mais conhecido por desenvolvimento web, também pode ser usado para criar aplicativos móveis. Oferece uma interface visual para construir aplicativos complexos sem codificação.
- **MIT App Inventor:** ferramenta de desenvolvimento visual criada pelo MIT que permite a criação de aplicativos Android por meio de uma interface de arrastar e soltar. É popular em ambientes educacionais por sua simplicidade e facilidade de uso, permitindo que iniciantes aprendam os conceitos básicos de programação e desenvolvimento de aplicativos.

1.3.3 Características

Os dispositivos móveis são computadores portáteis que se tornaram parte essencial do nosso dia a dia. Eles apresentam uma série de características que os diferenciam dos computadores tradicionais e os tornam tão populares. As suas características são:

- **Portabilidade:** projetados para serem leves e compactos, permitindo que sejam transportados facilmente em bolsos, bolsas ou mochilas.
- **Autonomia:** possuem baterias internas que permitem o uso por várias horas sem a necessidade de serem conectados a uma fonte de energia.
- **Conectividade:** conectam-se a redes sem fio (Wi-Fi, Bluetooth) e redes móveis (3G, 4G, 5G), permitindo acesso à internet e comunicação em qualquer lugar.
- **Tela touchscreen:** a maioria dos dispositivos móveis utiliza telas sensíveis ao toque, facilitando a interação com o usuário.
- **Miniaturização:** a tecnologia permite que componentes eletrônicos cada vez menores sejam integrados aos dispositivos móveis, aumentando sua capacidade e diminuindo seu tamanho.
- **Multifuncionalidade:** além de realizar chamadas telefônicas e enviar mensagens, os dispositivos móveis permitem acessar a internet, tirar fotos e vídeos, jogar games, utilizar aplicativos de navegação e muito mais.
- **Personalização:** os sistemas operacionais dos dispositivos móveis permitem que os usuários personalizem seus aparelhos, escolhendo wallpapers, widgets e aplicativos de acordo com suas preferências.

Outras características importantes:

- **Processador:** executa as tarefas do dispositivo, similar ao processador de um computador.
- **Memória RAM:** armazena temporariamente os dados que estão sendo utilizados pelo dispositivo, permitindo um funcionamento mais rápido.
- **Armazenamento interno:** armazena os dados do usuário, como aplicativos, fotos, vídeos e músicas.
- **Câmera:** captura fotos e vídeos com qualidade que varia de acordo com o modelo do dispositivo.
- **Sensores:** os dispositivos móveis possuem diversos sensores, como acelerômetro, giroscópio e sensor de luz, que detectam movimentos, orientação e condições de iluminação.

2 CONCEITOS DE ORIENTAÇÃO A OBJETO

Quando trabalhamos no Android Studio, a programação em Java e Kotlin utiliza o paradigma da orientação a objetos na técnica de programação utilizando views. Veremos uma breve introdução a essa maneira de programar, focando no uso dos projetos aqui presentes.



Saiba mais

A linguagem de programação Kotlin será a base para o desenvolvimento de aplicativos móveis neste curso. Para facilitar o aprendizado dos conceitos iniciais, utilize plataformas online de programação em Kotlin, como o Kotlin Playground, o JDoodle ou o Tutorials Point que estão nos seguintes links:

Kotlin Playground: <https://tinyurl.com/yc4z4788>. Acesso em: 6 mar. 2025.

JDoodle: <https://tinyurl.com/3ja2m6u9>. Acesso em: 6 mar. 2025.

Tutorials Point: <https://tinyurl.com/ymn29xkt>. Acesso em: 6 mar. 2025.

Podemos definir programação orientada a objetos (POO) como o modelo baseado em objetos. Ela se baseia em conceitos como classes, objetos, herança, polimorfismo, abstração, entre outros, que ajudam a criar software modular e reutilizável. No Android Studio, esses objetos são representados por componentes visuais, como botões, caixas de texto e imagens. Esses componentes possuem características e funcionalidades que podem ser personalizadas para atender as necessidades do aplicativo. Além dos componentes básicos, o Android Studio permite a importação de bibliotecas externas, que oferecem funcionalidades adicionais, como mapas, gráficos e redes sociais.

Assim, um aplicativo é composto por um conjunto de objetos que interagem entre si para garantir seu funcionamento. Os conceitos fundamentais para começar a programar são:

- **Objetos:** no Android Studio, cada elemento da tela do seu celular, como um botão, uma caixa de texto ou uma imagem, é um objeto. Cada objeto tem suas próprias propriedades (cor, tamanho e texto) e comportamentos (tocado ou clicado).
- **Classes:** uma classe é como um molde para criar objetos. Por exemplo, a classe Botão define características e comportamentos que todos os botões do seu aplicativo terão.
- **Instanciação:** para usar um objeto, precisamos criar uma cópia dele, chamada de instância. É como tirar uma cópia de um molde de Lego para criar um tijolo específico. No Android Studio, quando se insere um botão para a tela, cria-se uma instância da classe Botão.

2.1 Programação estrutural

Além de conhecer a orientação a objeto, é necessário saber pelo menos o fundamento do paradigma da programação estrutural, pois é essencial para qualquer tipo de desenvolvimento, incluindo o desenvolvimento de aplicativos Android.

A programação estruturada é um paradigma de programação que organiza o código de forma clara e lógica, facilitando escrita, leitura e manutenção de programas. Ela se baseia em três estruturas de controle fundamentais.

- **Sequência:** as instruções são executadas uma após a outra, na ordem em que aparecem no código.
- **Seleção:** permite tomar decisões com base em condições, utilizando estruturas como se, então e senão.
- **Repetição:** executa um bloco de código várias vezes, utilizando estruturas como repita e enquanto.

2.1.1 Variáveis

As variáveis são como recipientes que armazenam dados em um programa. Elas são declaradas usando as palavras-chave `var` ou `val`.

- **var:** declara uma variável mutável, ou seja, seu valor pode ser alterado após a declaração.
- **val:** declara uma variável imutável, cujo valor não pode ser modificado após a atribuição inicial.

```
1 fun main() {  
2     // Declarando variáveis  
3     var nome = "Bartolomeu" // Variável mutável  
4     val idade = 25 // Variável imutável  
5     var altura: Double = 1.65 // Variável com tipo explícito  
6  
7     // Utilizando as variáveis  
8     println("Olá, $nome! Você tem $idade anos e sua altura é $altura metros.")  
9  
10    // Alterando o valor de uma variável mutável  
11    nome = "Bonifácio"  
12    println("Agora seu nome é $nome.")  
13  
14    // Tentando alterar o valor de uma variável imutável (irá gerar um erro de compilação)  
15    // idade = 30 // Isso causará um erro!  
16 }
```

Olá?, Bartolomeu! Você tem 25 anos e sua altura é 1.65 metros.
Agora seu nome é Bonifácio.

Figura 1 – Criação e manipulação de variáveis

Na linha 3 da figura 1, é criada a variável chamada nome e ela armazenará a informação Bartolomeu. Observe que está entre aspas, isso significa que é um conjunto de caracteres comumente chamado de string. Na linha 4 é criada uma variável chamada idade e recebe a informação 25, um número inteiro. Ela foi declarada com a palavra-chave val, portanto, é imutável, este valor não poderá ser alterado.

Na linha 5 é criada a variável altura, tem dois pontos depois do nome e é escrito Double. Isso significa que essa variável sempre armazenará um valor com decimais de grande precisão. No momento da criação não é necessário saber o tipo de dado que cada variável irá receber, mas é importante conhecer o tipo de dado correto. Isso é crucial para garantir a eficiência e a segurança do seu código. Ao entender os diferentes tipos de dados e suas características, você estará preparado para desenvolver aplicativos Android robustos e escaláveis.

Continuando a análise da figura 1, na linha 11, pode-se mostrar o conteúdo das variáveis. Na linha 9, o conteúdo da variável nome é alterado, atribuindo-se um novo valor a ele. Na linha seguinte é exibida a nova informação armazenada na variável nome.

Note que, neste primeiro exemplo, várias linhas contêm o símbolo //. Isso indica um comentário, que é ignorado pelo programa até o final da linha. Comentários são úteis para adicionar observações que ajudam a entender o código. Ao remover o símbolo do comentário da linha 15, o programa acusará erro, pois a variável idade é imutável.

2.1.2 Tipos

Kotlin, a linguagem oficial para o desenvolvimento de aplicativos Android, oferece uma variedade de tipos de dados para representar diferentes tipos de informações. A escolha do tipo de dado correto é crucial para garantir eficiência e segurança do seu código.

- **Tipos de dados primitivos:** representam valores básicos, como números inteiros, números reais, caracteres e booleanos, que ocupam um espaço fixo na memória:
 - **Int:** representa números inteiros (figura 2, linha 2).
 - **Long:** representa números inteiros de 64 bits, para valores muito grandes.
 - **Double:** representa números de ponto flutuante de precisão dupla (figura 2, linha 3).
 - **Float:** representa números de ponto flutuante de precisão simples.
 - **Boolean:** representa valores lógicos, verdadeiro (true) ou falso (false) (figura 2, linha 4).
 - **Char:** representa um único caractere Unicode (figura 2, linha 5).
 - **Byte:** representa um número inteiro de 8 bits.
 - **Short:** representa um número inteiro de 16 bits.

- **Tipos de dados de referência:** representam objetos com tamanho de memória variável, acessados por meio de referências a endereços de memória.
 - **String:** representa uma sequência imutável de caracteres.
 - **Array:** representa uma coleção ordenada de elementos do mesmo tipo.
 - **List:** representa uma coleção ordenada de elementos, podendo conter duplicados (figura 2, linha 6).
 - **Set:** representa uma coleção não ordenada de elementos únicos.
 - **Map:** representa um conjunto de pares chave-valor (figura 2, linha 7).
 - **Pair:** representa uma coleção ordenada de dois elementos do mesmo tipo.
- **Tipos de dados especiais:** representam caracteres genéricos que contêm conceitos abstratos e desempenham papéis específicos na linguagem.
 - **Any:** é o tipo base de todas as classes e as interfaces em Kotlin.
 - **Unit:** representa a ausência de valor, similar ao void em outras linguagens.
 - **Nothing:** representa a ausência de valor e não retorna de nenhuma função.

Uma das características mais interessantes do Kotlin é a inferência de tipos. O compilador pode inferir o tipo de uma variável com base no valor inicial atribuído.

```
1 fun main() {  
2     val idade: Int = 25  
3     val altura: Double = 1.75  
4     val estaChovendo = true  
5     val primeiraLetraDoNome: Char = 'A'  
6     val listaDeFrutas = listOf("Laranja", "Banana", "Uva")  
7     val mapaDeIdades = mapOf("Bartolomeu" to 25, "Bernardo" to 30)  
8  
9     println("Idade: $idade")  
10    println("Altura: $altura")  
11    println("Esta chovendo? $estaChovendo")  
12    println("Primeira letra do nome: $primeiraLetraDoNome")  
13    println("Lista de frutas: $listaDeFrutas")  
14    println("Mapa de idades: $mapaDeIdades")  
15 }
```

Idade: 25
Altura: 1.75
Esta chovendo? true
Primeira letra do nome: A
Lista de frutas: [Laranja, Banana, Uva]
Mapa de idades: {Bartolomeu=25, Bernardo=30}

Figura 2 – Tipos de dados

2.1.3 Operações

O Kotlin oferece variadas operações para manipular dados e realizar cálculos. Exploraremos as principais categorias de operações.

Operadores aritméticos

Utilizados para realizar cálculos matemáticos:

- **Adição:** + (figura 3, linha 4).
- **Subtração:** -.
- **Multiplicação:** *.
- **Divisão:** /.
- **Módulo:** % (resto da divisão) (figura 3, linha 5).
- **Incremento:** ++.
- **Decremento:** --.

```
1 fun main() {  
2     val x = 10  
3     val y = 3  
4     val soma = x + y  
5     val resto = x % y  
6     println("Soma: $soma, Resto: $resto")  
7 }
```

Soma: 13, Resto: 1

Figura 3 – Operações aritméticas

Operadores de comparação

Utilizados para comparar valores:

- **Igual:** == (figura 4, linha 4).
- **Diferente:** !=.
- **Maior que:** > (figura 4, linha 5).

- Menor que: <.
- Maior ou igual: >=.
- Menor ou igual: <=.

```
1 fun main() {  
2     val a = 5  
3     val b = 10  
4     val igual = a == b  
5     val maior = a > b  
6     println("Sao iguais? $igual")  
7     println("A e maior que B? $maior")  
8 }
```

Sao iguais? false
A e maior que B? false

Figura 4 – Operadores de comparação

Operadores lógicos

Utilizados para combinar expressões booleanas.



Lembrete

Operações booleanas são operações lógicas que trabalham com valores binários, ou seja, aqueles que podem ser apenas verdadeiros (true) ou falsos (false). Elas são fundamentais em áreas como matemática, lógica e computação.

E lógico: &&

Quadro 1 – Tabela verdade da operação E

A (Entrada 1)	B (Entrada 2)	A && B (Saída)
false	False	false
false	True	false
true	False	false
true	True	true

O resultado é True somente quando os dois argumentos forem True (quadro 1).

Ou lógico: ||

Quadro 2 – Tabela verdade da operação OU

A (Entrada 1)	B (Entrada 2)	A B (Saída)
false	False	false
false	True	true
true	False	true
true	True	true

O resultado é True quando um dos argumentos for True (quadro 2).

Negação: !

Quadro 3 – Tabela verdade da operação NÃO

A (Entrada)	NOT A (Saída)
false	True
true	False

O resultado é invertido se é True vira False, se é False vira True (quadro 3). Observe que, na figura 5, a decisão é True apenas quando os dois argumentos, temSol e estaChovendo, forem verdadeiros.

```
1 fun main() {  
2     val temSol = true  
3     val estaChovendo = false  
4     val possoSair = temSol && estaChovendo  
5     println("Posso sair? $possoSair")  
6 }
```

Posso sair? false

Figura 5 – Operação lógica E

```
1 fun main() {  
2     val temSol = true  
3     val estaChovendo = true  
4     val possoSair = temSol && estaChovendo  
5     println("Posso sair? $possoSair")  
6 }
```

Posso sair? true

Figura 6

Operadores de nulidade

Utilizados para verificar se uma referência é nula.



Observação

Uma referência nula é como uma variável que aponta para nada. É como se estivesse procurando um endereço em um mapa, mas ele não existisse. Um dos maiores problemas em muitas linguagens de programação é o chamado `NullPointerException`. Isso acontece quando se tenta acessar um valor de uma variável que é nula, ou seja, ela não aponta para nenhum objeto ou valor específico na memória, procura uma variável e ela não está na memória.

Operador Elvis: ?

O operador Elvis retorna o valor à esquerda se ele não for nulo; caso contrário, retorna o valor à direita. Ele é especialmente útil em situações em que se precisa fornecer um valor padrão caso uma expressão seja nula. Normalmente ele é utilizado quando:

- **Queremos evitar `NullPointerException`:** ao invés de verificar explicitamente se um valor é nulo antes de usá-lo, o operador Elvis fornece um valor padrão caso o valor original seja nulo.
- **Queremos simplificar o código:** o operador Elvis torna o código mais conciso e legível, eliminando a necessidade de escrever condicionais longas.
- **Queremos fornecer um valor padrão:** quando uma expressão pode retornar null, o operador Elvis nos permite fornecer valores padrão quando uma variável pode ser nula.

```
1 fun main() {  
2     val nome: String? = null  
3     val nomeSeguro = nome ?: "Anonimo"  
4     println(nomeSeguro)  
5 }  
Anonimo  
1 fun main() {  
2     val nome: String? = "Basilio"  
3     val nomeSeguro = nome ?: "Anonimo"  
4     println(nomeSeguro)  
5 }  
Basilio
```

Figura 7 – Operador de nulidade Elvis

Operador de segurança: ?.

O operador de segurança ?. verifica se o objeto à esquerda é nulo antes de tentar acessar a propriedade ou método à direita. Se o objeto for nulo, a expressão inteira retorna null em vez de lançar uma exceção.

```
1 fun main() {
2     val nome: String? = "Basilio"
3     val tamanhoNome = nome ?.length
4     println("O nome tem $tamanhoNome letras")
5 }
O nome tem 7 letras

fun main() {
    val nome: String? = null
    val tamanhoNome = nome ?.length
    println("O nome tem $tamanhoNome letras")
}

1 //sem a segurança ?.
2 fun main() {
3     val nome: String? = null
4     val tamanhoNome = nome.length
5     println("O nome tem $tamanhoNome letras")
6 }
main.kt:4:27: error: only safe (?.) or non-null asserted (!!)
calls are allowed on a nullable receiver of type String?
val tamanhoNome = nome.length
                    ^
```

Figura 8 – Uso do operador de segurança

Operador de chamada segura: ?.let

O objeto à esquerda do operador ?. é passado como argumento para a função let apenas se não for nulo. Dentro do bloco let, acessa-se o objeto usando a palavra-chave it.

```
1 fun main() {
2     val nome: String? = null
3     nome?.let {
4         println("O comprimento do nome é ${it.length}")
5     }
6 }
//nada é mostrado

1 fun main() {
2     val nome: String? = "Benevides"
3     nome?.let {
4         println("O comprimento do nome é ${it.length}")
5     }
6 }
O comprimento do nome ? 9
```

Figura 9 – Operador de nulidade ?.let

Seleção

As estruturas de seleção, ou as condicionais em Kotlin, permitem que seu programa tome decisões com base em determinadas condições.

Condicional if/else

A condicional if/else é usada para executar diferentes blocos de código com base em uma condição booleana. A condicional if pode ser usada tanto como uma expressão quanto como uma instrução. Caso a condição booleana resulte verdadeira, o bloco logo após a condição é executado; caso resulte falsa, o bloco após o else é executado.

A estrutura da sintaxe é:

```
if (condicao) {  
    // Bloco de código executado se a condição for verdadeira  
} else {  
    // Bloco de código executado se a condição for falsa  
}
```

```
1 fun main() {  
2     val numero = 10  
3     if (numero > 0) {  
4         println("O numero e positivo")  
5     } else {  
6         println("O numero e negativo ou zero")  
7     }  
8 }
```

O numero e positivo

```
1 fun main() {  
2     val numero = -7  
3     if (numero > 0) {  
4         println("O numero e positivo")  
5     } else {  
6         println("O numero e negativo ou zero")  
7     }  
8 }
```

O numero e negativo ou zero

Figura 10 – O uso da condicional if/else

when

A estrutura condicional when permite avaliar uma variável ou expressão contra múltiplos valores ou condições, tornando o código mais limpo e legível.

A sua estrutura é:

```
when (expressao) {  
    valor1 -> {  
        // Bloco de código para valor1  
    }  
    valor2 -> {  
        // Bloco de código para valor2  
    }  
    else -> {  
        // Bloco de código para qualquer outro valor  
    }  
}
```

Ela funciona direcionando a sequência de instruções para o item que é igual ao conteúdo da expressão.

Exemplo:

```
1 fun main() {  
2     val dia = 4  
3  
4     val diaDaSemana = when (dia) {  
5         1 -> "Domingo"  
6         2 -> "Segunda-feira"  
7         3 -> "Terça-feira"  
8         4 -> "Quarta-feira"  
9         5 -> "Quinta-feira"  
10        6 -> "Sexta-feira"  
11        7 -> "Sábado"  
12        else -> "Dia inválido"  
13    }  
14    println(diaDaSemana)  
15 }
```

Quarta-feira

Figura 11 – A escolha do dia da semana baseada em um número

Na figura 11, a variável dia é comparada com vários valores. Dependendo do valor de dia, a variável diaDaSemana recebe um valor correspondente. Se dia não corresponder a nenhum dos valores especificados, o bloco else é executado.

```
1 fun main() {  
2     val numero = 15  
3  
4     when {  
5         numero < 0 -> println("Número negativo")  
6         numero in 1..10 -> println("Número entre 1 e 10")  
7         numero is Int -> println("Numero e um inteiro")  
8         else -> println("Outro caso")  
9     }  
10 }
```

Numero e um inteiro

```
1 fun main() {  
2     val numero = 5  
3  
4     when {  
5         numero < 0 -> println("Numero negativo")  
6         numero in 1..10 -> println("Numero entre 1 e 10")  
7         numero is Int -> println("Numero e um inteiro")  
8         else -> println("Outro caso")  
9     }  
10 }
```

Numero entre 1 e 10

Figura 12 – A estrutura de escolha baseada em instruções

Na figura 12, `when` não possui a expressão para efetuar a escolha. A variável em si já funciona como condição de execução.

2.1.4 Repetição

As estruturas de repetição em Kotlin, também conhecidas como loops, são mecanismos que permitem executar um bloco de código repetidamente enquanto uma determinada condição for verdadeira. Elas são essenciais para automatizar tarefas repetitivas e percorrer coleções de dados. Kotlin oferece três principais estruturas de repetição: `for`, `while` e `do-while`.

for

A estrutura `for` é usada para iterar, percorrer uma coleção de itens, como uma lista ou um intervalo de números, ou um array e outros tipos de coleções.

A sua estrutura é:

```
for (item in colecao) {  
    // Bloco de código a ser executado para cada item  
}
```

Quando se deseja mostrar cada elemento de uma lista, utiliza-se o laço for. No laço for, a variável numero recebe sequencialmente um elemento da lista números a cada iteração.

```
1 fun main() {  
2     val numeros = listOf(1, 2, 3, 4, 5)  
3     for (numero in numeros) {  
4         println(numero)  
5     }  
6 }  
  
1  
2  
3  
4  
5
```

Figura 13 – Laço percorrendo uma lista com número

while

A estrutura while executa um bloco de código enquanto uma condição especificada for verdadeira. A estrutura do while é:

```
while (condicao) {  
    // Bloco de código a ser executado enquanto a condição for verdadeira  
}
```

Na figura 14, a variável contador é subtraída de 1 até que a condição contador > 0 do while se mantenha positiva.

```
1 fun main() {  
2     var contador = 5  
3     while (contador > 0) {  
4         println("resta $contador")  
5         contador--  
6     }  
7 }  
  
resta 5  
resta 4  
resta 3  
resta 2  
resta 1
```

Figura 14 – Efetuando uma contagem regressiva com a variável maior que zero

Há a possibilidade de o comando da linha 4 nunca ser executado, por exemplo se o contador for iniciado por um número negativo. Caso seja necessário pelo menos uma iteração, utiliza-se a estrutura do-while.

Do-while

Esta estrutura garante que o bloco de código seja executado pelo menos uma vez, pois a condição é verificada após a execução do bloco. A sua estrutura é:

```
do {  
    // Bloco de código a ser executado pelo menos uma vez  
} while (condicao)
```

Na figura 15, a condição é verificada apenas no final dos comandos do corpo do laço. Assim, esse corpo é sempre executado pelo menos uma vez.

```
1 fun main() {  
2     var contador = 5  
3     do {  
4         println("resta $contador")  
5         contador--  
6     } while (contador > 0)  
7 }
```

resta 5
resta 4
resta 3
resta 2
resta 1
|

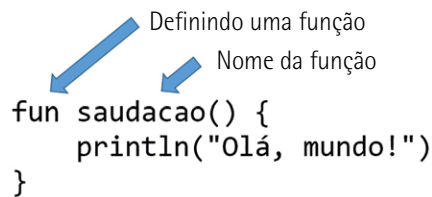
Figura 15 – No laço do-while, a comparação é feita no final do laço

Funções

Em programação, uma função é um bloco de código que pode ser chamado várias vezes de diferentes partes do programa para realizar uma tarefa específica. A função pode receber dados como entrada (parâmetros) e fornecer um resultado (retorno).

Declaração de funções

Para declarar uma função em Kotlin, usa-se a palavra-chave fun, seguida pelo nome da função, parênteses e um bloco de código entre chaves (figura 16).

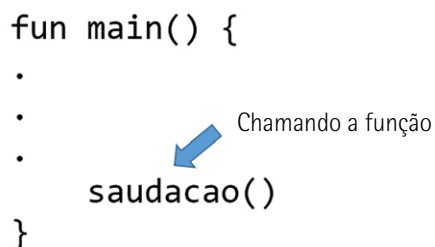


```
fun saudacao() {  
    println("Olá, mundo!")  
}
```

Figura 16

Chamando funções

Para chamar uma função, basta usar seu nome seguido de parênteses.

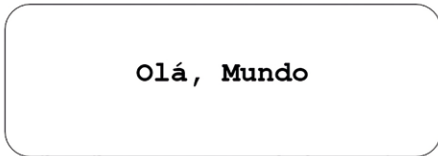


```
fun main() {  
    .  
    .  
    .  
    saudacao()  
}
```

Figura 17

O Kotlin executa a função `main()`. Ao encontrar a chamada para `saudacao()`, a execução é temporariamente transferida para a função `saudacao()`, retornando depois de executá-lo.

Saída:



Olá, Mundo

Figura 18

Funções com parâmetros

As funções podem aceitar parâmetros, que são valores passados para a função quando ela é chamada (figura 19).

```
fun saudacao(nome: String) {  
    println("Olá, $nome!")  
}  
  
fun main() {  
    saudacao("Maria")  
}
```

Annotations in the diagram:

- Parâmetro (points to `nome`)
- Tipo do parâmetro (points to `String`)
- Envio do parâmetro (points to `"Maria"`)

Figura 19

Ao invocar a função `saudacao` com o argumento `Maria`, fornece-se um valor inicial para a variável `nome` dentro da função. Essa variável será utilizada pela função para personalizar a saudação.

Saída:

Olá, Maria!

Figura 20

Funções com valor de retorno

As funções podem retornar um valor usando a palavra-chave `return`. O tipo de retorno é especificado após os parênteses da função (figura 21).

```
fun soma(a: Int, b: Int): Int {  
    return a + b  
}  
  
fun main() {  
    val resultado = soma(3, 4)  
    println("Resultado: $resultado")  
}
```

Annotations in the diagram:

- Tipo do valor retornado (points to `Int` in the function signature)
- Retornando o valor (points to `return a + b`)
- A função retorna um valor (points to the assignment `val resultado = soma(3, 4)`)

Figura 21

Ao invocar a função soma com os argumentos 3 e 4, fornecem-se valores iniciais para as variáveis a e b dentro da função. A função calcula a soma desses valores e devolve o resultado para o local onde foi chamada.

Saída:

Resultado: 7

Figura 22

Funções de expressão única

Se a função consiste em uma única expressão, usa-se uma sintaxe mais concisa.

```
fun quadrado(x: Int) = x * x
fun main() {
    println(quadrado(5))
}
```

A função retorna o resultado da expressão

Figura 23

Saída:

25

Figura 24

Funções de extensão

Kotlin permite adicionar novas funções a classes existentes sem modificá-las, usando funções de extensão.

Para declarar uma função de extensão, é necessário prefixar o nome da função com o tipo do receptor, que é o tipo que está estendendo. A função e a classe (veremos a seguir) recebem uma nova funcionalidade além daquela que ele já possui.

```
fun String.saudar() {  
    println("Olá, $this!")  
}  
  
fun main() {  
    "Kotlin".saudar()  
}
```

Variável que recebe um valor do tipo

Variável recebido na chamada

Figura 25

As strings em Kotlin possuem uma série de métodos embutidos para realizar diversas operações, como calcular o comprimento com `length` ou converter para maiúsculas com `toUpperCase`. No entanto, a criação de saudações não é uma funcionalidade padrão. Para resolver isso, define-se uma função de extensão chamada `saudar()` que, quando aplicada a uma string, retorna uma nova string com a palavra `Olá` concatenada à string original.

Saída:

Olá, Kotlin!

Figura 26

Funções anônimas e lambdas

Em Kotlin, lambdas são funções anônimas que podem ser atribuídas a variáveis, passadas como argumentos para outras funções ou retornadas como resultado de uma função. Elas fornecem uma sintaxe concisa e flexível para expressar blocos de código de forma funcional. Ele é caracterizado pela estrutura:

```
{ parâmetros -> corpo }
```

```
fun main() {  
    val soma: (Int, Int) -> Int = { a, b -> a + b }  
    print(soma(2, 3))  
}
```

Variável que faz o papel de uma função

Corpo da função

Figura 27

Saída:

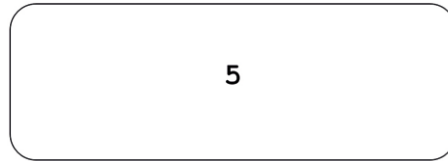


Figura 28

Funções de ordem superior

São funções que aceitam outras funções como parâmetros ou retornam funções.

Passagem de função como parâmetro

```
val numeros = listOf(1, 2, 3, 4, 5)
val numerosPares = numeros.filter { it % 2 == 0 }
println(numerosPares)
```

Figura 29

Para compreender cada passo:

- **Linha 1:**
 - **listOf(1, 2, 3, 4, 5):** essa função cria uma lista imutável contendo os números 1, 2, 3, 4 e 5 armazenada na variável imutável `numeros`.
- **Linha 2:**
 - **`numeros.filter { it % 2 == 0 }`:** a função `filter` é uma função de ordem superior que recebe uma lambda como argumento. Esta lambda é aplicada a cada elemento da lista `numeros`. `it % 2 == 0`.
 - **A lambda verifica se o número (it) é par:** o operador `%` calcula o resto da divisão do número por 2. Se o resto for 0, o número é par.
 - **`val numerosPares`:** a variável `numerosPares` armazena a nova lista contendo apenas os números que passaram no filtro, ou seja, os números pares.
- **Linha 3:**
 - **`println(numerosPares)`:** essa linha imprime a lista `numerosPares` no console. Se a lista original `numeros` contiver os números [1, 2, 3, 4, 5], a saída será [2, 4], pois esses são os números pares da lista original.

Saída:

[2, 4]

Figura 30



Observação

O uso das funções é muito importante, pois o Android Studio Compose, uma ferramenta moderna para criar interfaces de usuário no Android, introduziu uma nova forma de pensar sobre a construção de UIs, utilizando uma abordagem declarativa e funcional.



Lembrete

Ao invés de classes tradicionais, o Compose utiliza funções composáveis para definir a estrutura da UI (User Interface).

Exemplo: analisaremos a função Greeting, padrão do projeto inicial Empty Activity do Compose no Android Studio:

```
1 @Composable
2 fun Greeting(name: String, modifier: Modifier = Modifier) {
3     Text(
4         text = "Olá $name!",
5         modifier = modifier
6     )
7 }
```

Figura 31

- Anotação (linha 1):
 - A anotação `@Composable` indica que a função `Greeting` é uma função composável. Isso significa que ela pode ser usada para construir a UI.
- Declaração e parâmetros (linha 2):
 - O comando `fun` inicia a função batizada de `Greeting` (linha 2).

- Parâmetros:
 - **name**: string que representa o nome a ser exibido na saudação.
 - **modifier**: modificador opcional que permite personalizar a aparência e o comportamento do texto. Se não for fornecido, o modificador padrão será usado.
- Corpo da função (linhas 3 a 5):
 - **Text**: chama uma função composável predefinida que exibe texto na tela.
 - **text**: "Olá \$name!": define o texto a ser exibido, que inclui a saudação "Olá" seguida pelo nome passado como argumento.
 - **modifier**: o modificador aplicado ao texto, que pode ser usado para definir margens, preenchimento, cor de fundo etc.



Observação

Frequentemente, os parâmetros podem ser colocados em linhas separadas para uma visualização mais concisa. Sintaticamente, o código permanece idêntico, pois, após uma vírgula, o compilador ignora a quebra de linha e aguarda o próximo parâmetro. A função Greeting fica assim:

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

2.2 Objeto

Objetos podem ser considerados uma imitação do comportamento intrínseco de entidades reais. Tal como em sistemas reais, em uma POO não é viável abrir um objeto e olhar em seu interior e tampouco alterar seu estado. Nesse paradigma, a única forma de fazer evoluir um programa é permitir que objetos compartilhem dados entre si a partir de trocas explícitas de mensagens.

Para se criar um objeto em Kotlin devemos criar uma classe. Um objeto é uma instância de uma classe (Sintes, 2002). Assim é a concretização de uma classe.

2.3 Classe

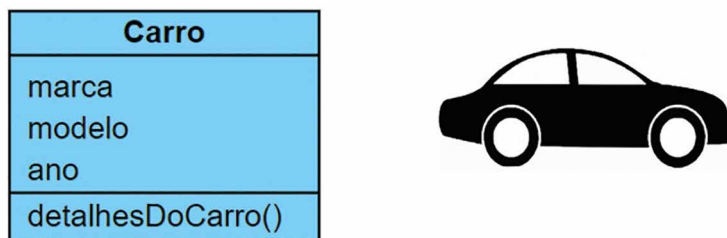


Figura 32

Pensando nos carros, há uma classe que serve de modelo a todos os objetos criados a partir dele. Essa classe possui atributos (características) e métodos (comportamentos) próprios da classe Carro. Esses métodos e atributos se materializam em carros mais específicos, como aqueles que saem da fábrica. Podemos dizer, portanto, que os objetos carro1, carro2 e carro3 são instâncias.

No Kotlin, a classe Carro, seguindo o exemplo, é escrita da seguinte forma:

```
class Carro(val marca: String, val modelo: String, val ano: Int) {  
    fun detalhesDoCarro() {  
        println("Marca: $marca, Modelo: $modelo, Ano: $ano")  
    }  
}
```

A classe Carro define uma classe com três propriedades (marca, modelo e ano) e um método (detalhesDoCarro) que imprime os detalhes do carro.

É importante saber a respeito desse código que a partir dele o programa reconhece uma classe pela declaração `class` e ele tem um nome, no caso Carro. Logo após o nome, entre parênteses há o chamado parâmetros. Depois dos parênteses há uma chave abrindo. Após isso chama-se de corpo da classe onde vão as instruções. A classe é encerrada fechando-se a chave.

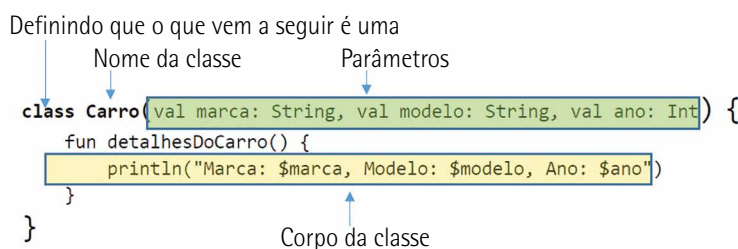


Figura 33

Uma classe define os atributos e comportamentos comuns compartilhados por um tipo de objeto. Os objetos de certo tipo ou classificação compartilham os mesmos comportamentos e atributos. As classes atuam de forma muito parecida com um cortador de molde ou biscoito, no sentido de que você usa uma classe para criar ou instanciar objetos [...] (Sintes, 2002, p. 8).

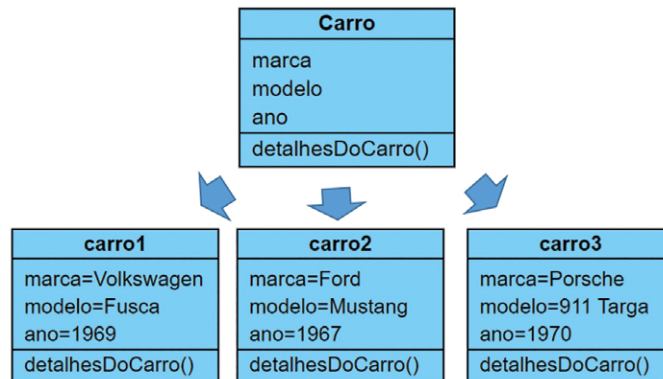


Figura 34

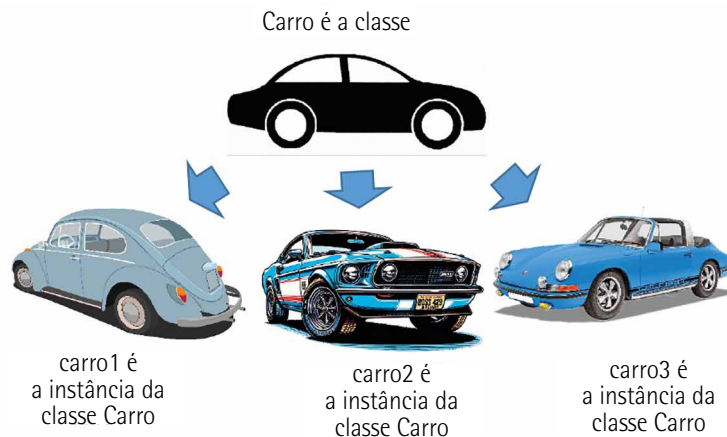


Figura 35

Codificando o processo de criar uma instância, são criados três objetos (`carro1`, `carro2`, `carro3`) a partir da classe `Carro`, cada um com valores diferentes para as propriedades.

```
val carro1 = Carro("Volkswagen", "Fusca", 1969)
```

```
val carro2 = Carro("Ford", "Mustang", 1967)
```

```
val carro3 = Carro("Porsche", "911", 2012)
```

Cada objeto chama o método `detalhesDoCarro` imprimindo os detalhes de cada carro.

```
carro1.detalhesDoCarro()
carro2.detalhesDoCarro()
carro3.detalhesDoCarro()
```

No Android Studio, você irá trabalhar com diversas classes para construir a interface do usuário, gerenciar dados, lidar com eventos e muito mais.

- **Classes fundamentais:**

- **Activity:** representa uma única tela interativa em um aplicativo. É a classe base para a maioria das telas do seu app.
- **Fragment:** um componente modular de uma activity, permitindo criar interfaces mais complexas e reutilizáveis.
- **View:** a classe base para todos os elementos visuais na tela, como botões, text views etc.
- **Intent:** um objeto que descreve uma ação a ser realizada, como iniciar uma nova activity, abrir uma página da web ou enviar um e-mail.

- **Classes para Interface do Usuário (UI):**

- **TextView:** exibe texto na tela.
- **Button:** um botão que o usuário pode clicar.
- **EditText:** campo de texto para entrada do usuário.
- **ImageView:** exibe imagens.
- **LinearLayout:** layout que organiza os elementos da interface em uma linha horizontal ou vertical.
- **RelativeLayout:** layout que posiciona os elementos em relação a outros elementos ou aos pais.
- **ConstraintLayout:** layout flexível que permite criar layouts complexos com facilidade.

Exemplo: no Android Studio criando uma instância da classe Button referenciado por botaoOk.

```
val botaoOk = Button(this)
```

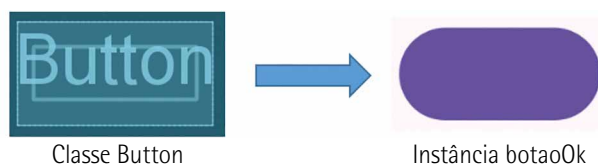


Figura 36 – Ilustração do processo de criação do objeto botaoOk a partir da classe Button

2.4 Atributo

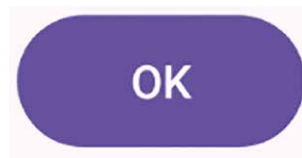
"Atributos são as características de uma classe visíveis externamente. A cor dos olhos e a cor dos cabelos são exemplos de atributos" (Sintes, 2002, p. 8).

Os atributos são variáveis que pertencem a uma classe e representam características ou propriedades dos objetos criados a partir dessa classe. Eles são definidos dentro da classe e podem ser acessados e modificados por instâncias dessa classe.

No caso da classe Carro, os atributos caracterizam cada um dos carros criados. Assim, o carro1 tem como características ser da marca Volkswagen, modelo Fusca e o ano de fabricação 1969.

Exemplo: manipulação da instância botaoOk, trocando o texto e o tamanho do texto. No caso, o atributo text é alterado.

```
botaoOk.text = "OK"
```



Atributo
text = "OK"

Figura 37 – Atributo text do objeto botaoOk alterado

2.5 Métodos

Métodos são funções associadas a uma classe e definem o comportamento de um objeto dessa classe. Eles permitem a manipulação dos dados (atributos) do objeto e realizam tarefas específicas.

No caso da classe Carro, o método, no caso a função detalhesDoCarro(), quando é acionado executa o print mostrando na tela as características (atributos) do objeto criado.

Exemplo: o método setOnClickListener capta um clique no botaoOk e o texto é alterado:

```
botaoOk.text = "OK"  
botaoOk.setOnClickListener {  
    // Alterando o texto do botão quando clicado  
    botaoOk.text = "Botão clicado!"  
}
```

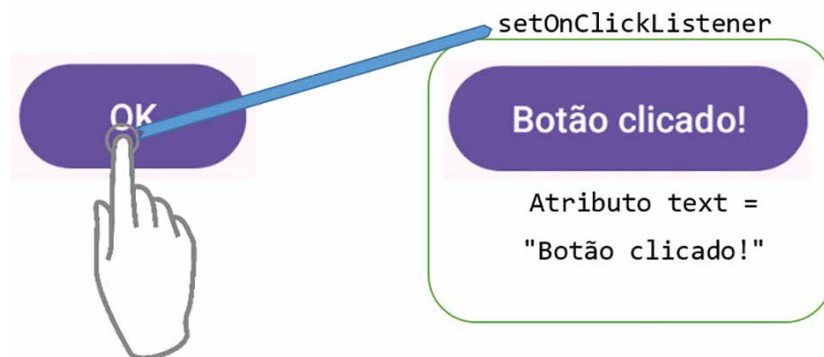


Figura 38 – Ao clicar, setOnClickListener captura o evento e o atributo text é alterado

2.6 Herança

A herança permite que você baseie a definição de uma nova classe em uma classe previamente existente. Quando você baseia uma classe em outra, a definição da nova classe herda automaticamente todos os atributos, comportamentos e implementações presentes na classe previamente existente (Sintes, 2002, p. 72).

Em programação orientada a objetos, a herança é um mecanismo que permite criar classes (subclasses ou classes-filhas) a partir de uma classe existente (superclasse ou classe-pai). A nova classe herda todos atributos e métodos da classe-pai, podendo adicionar novos atributos e métodos ou sobrescrever os existentes.

Para que uma classe possa ser herdada, ela deve ser marcada com a palavra-chave `open`. Por padrão, todas as classes em Kotlin são `final`, o que significa que não podem ser herdadas a menos que sejam explicitamente marcadas como `open`.

```
open class Veiculo {
    open fun acelerar() {
        println("O veiculo esta acelerando.")
    }
}
class Carro(val marca:String, val modelo:String, val ano:Int):Veiculo(){
    fun detalhesDoCarro(){
        print("Marca: $marca, Modelo: $modelo, Ano: $ano ")
    }
}
fun main(){
    val carro1 = Carro("Volkswagen", "Fusca", 1969)
    carro1.detalhesDoCarro()
    carro1.acelerar()
}
```

Figura 39

A classe `Carro` é instanciada. Embora ela não tenha o método `acelerar`, ele pode ser utilizado devido à herança da classe `Veículo`, que possui esse método.

Saída:

```
Marca: Volkswagen, Modelo: Fusca,
Ano: 1969 O veiculo esta
acelerando.
```

Figura 40

Exemplo: a classe `BotaoContador` é herdeiro da classe `Button`. Desta forma, os atributos e os métodos da classe `Button` estarão presentes na instância da classe `BotaoContador`.

```
class BotaoContador : Button {
    private var contaClick = 0
    constructor(context: Context) : super(context) {
        // Código de inicialização
    }
    constructor(context: Context, attrs: AttributeSet) : super(context, attrs)
{
    // Código de inicialização
}
fun incrementaContaClick() {
    contaClick++
    text = "Clicado $contaClick vezes"
}
}

val botaoContador = BotaoContador(this)
botaoContador.setOnClickListener {
    botaoContador.incrementaContaClick()
}
```

2.7 Polimorfismo

Polimorfismo, em programação orientada a objetos, significa muitas formas. É a capacidade de um objeto assumir diferentes formas dependendo do contexto. Em outras palavras, um mesmo método pode se comportar de maneiras distintas em classes diferentes, desde que elas estejam relacionadas por herança.

```
open class Veiculo () {
    open fun acelerar() {
        println("O veiculo esta acelerando.")
    }
}
class Carro(val marca: String, val modelo: String, val ano: Int):Veiculo()
{
    override fun acelerar(){
        print("Marca: $marca, Modelo: $modelo, Ano: $ano esta acelerando\n ")
    }
}
fun main(){
    val moto1 = Veiculo()
    val carro1 = Carro("Volkswagen", "Fusca", 1969)
    carro1.acelerar()
    moto1.acelerar()
}
```



Instâncias de classes diferentes com método de mesmo nome

Figura 41

3 INTRODUÇÃO AO ANDROID

Atualmente, o Android domina o mercado de dispositivos móveis. É o sistema operacional mais popular em smartphones, tablets e outros dispositivos como automóveis e wearables. Com mais de 2 bilhões de dispositivos ativos, o Android é a plataforma de desenvolvimento mais utilizada no mundo (Leal, 2019).

O sistema operacional Android foi desenvolvido originalmente pela Android Inc. e posteriormente adquirido pelo Google em 2005. O Android é resultado de um esforço colaborativo da Open Handset Alliance. Essa aliança, composta por 84 empresas, foi fundada em 2007 para desenvolver, manter e aprimorar a plataforma Android. Por meio dessa parceria, a indústria inova tecnologicamente no setor móvel, otimiza a experiência do usuário e torna os dispositivos mais acessíveis (Deitel; Deitel; Deitel, 2015).

O Android vai além de um simples sistema operacional. É uma plataforma abrangente que inclui uma base de software composta de três camadas principais: o kernel Linux, responsável pela gestão de recursos do dispositivo; o middleware, que atua como intermediário entre os aplicativos e o sistema operacional, facilitando a comunicação e a troca de dados; e os aplicativos básicos, como discador, navegador e contatos, que oferecem funcionalidades básicas aos usuários.

O Android, um sistema operacional de código aberto licenciado sob Apache, uma das licenças de software livre mais populares e permissivas. Ela permite liberdade para os desenvolvedores utilizarem e modificarem o código-fonte, dando flexibilidade para os fabricantes criarem versões personalizadas.



Saiba mais

O código-fonte está disponível no link a seguir, para modificações e adaptações. No entanto, a licença não impõe o compartilhamento dessas alterações. Para utilizar os serviços do Google, os dispositivos devem cumprir as especificações do CDD (Compatibility Definition Document) e passar pelo CTS (Compatibility Test Suite), assegurando a compatibilidade com a plataforma Android.

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

Desde o seu lançamento em 2008, o Android passou por diversas atualizações, cada uma com um nome de doce (Cupcake, Donut, Eclair etc.) e um número de API associado. Esse número é crucial para os desenvolvedores, pois indica quais ferramentas e funcionalidades estão disponíveis para criar aplicativos nessa versão específica.

Quadro 4 – Histórico das versões do Android até 2024

Codinome	Versão	Nível da API/versão do NDK	Lançamento
(sem codinome)	1.0	API de nível 1	23 de setembro de 2008
(sem codinome)	1.1	API de nível 2	9 de fevereiro de 2009
Cupcake	1.5	API de nível 3, NDK 1	27 de abril de 2009
Donut	1.6	API de nível 4, NDK 2	15 de setembro de 2009
Eclair	2.0	API de nível 5	27 de outubro de 2009
Eclair	2.0.1	API de nível 6	3 de dezembro de 2009

Codiname	Versão	Nível da API/versão do NDK	Lançamento
Eclair	2.1	API de nível 7, NDK 3	11 de janeiro de 2010
Froyo	2.2.x	API de nível 8, NDK 4	20 de maio de 2010
Gingerbread	2.3 - 2.3.2	API de nível 9, NDK 5	6 de dezembro de 2010
Gingerbread	2.3.3 - 2.3.7	API de nível 10	9 de fevereiro de 2011
Honeycomb	3.0	API de nível 11	22 de fevereiro de 2011
Honeycomb	3.1	API de nível 12, NDK 6	10 de maio de 2011
Honeycomb	3.2.x	API de nível 13	15 de julho de 2011
Ice Cream Sandwich	4.0.1 - 4.0.2	API de nível 14, NDK 7	18 de outubro de 2011
Ice Cream Sandwich	4.0.3 - 4.0.4	API de nível 15, NDK 8	16 de dezembro de 2011
Jelly Bean	4.1.x	API de nível 16	9 de julho de 2012
Jelly Bean	4.2.x	API de nível 17	13 de novembro de 2012
Jelly Bean	4.3.x	API de nível 18	24 de julho de 2013
KitKat	4.4 - 4.4.4	API de nível 19	31 de outubro de 2013
KitKat	4.4W - 4.4W.2	API de nível 19	25 de junho de 2014
Lollipop	5.0	API de nível 21	4 de novembro de 2014
Lollipop	5.1	API de nível 22	2 de março de 2015
Marshmallow	6.0	API de nível 23	2 de outubro de 2015
Nougat	7.0	API de nível 24	22 de agosto de 2016
Nougat	7.1	API de nível 25	4 de outubro de 2016
Oreo	8.0.0	API de nível 26	21 de agosto de 2017
Oreo	8.1.0	API de nível 27	5 de dezembro de 2017
Pie	9	API de nível 28	6 de agosto de 2018
Android10	10	API de nível 29	3 de setembro de 2019
Android11	11	API de nível 30	8 de setembro de 2020
Android12	12	API de nível 31	4 de outubro de 2021
Android12L	12,1	API de nível 32	7 de março de 2022
Android13	13	API de nível 33	15 de agosto de 2022
Android14	14	API de nível 34	4 de outubro de 2023
Android15	15		

As principais características do Android são:

- **Código aberto:** a natureza open source do Android permite que desenvolvedores personalizem e adaptem o sistema para diversas necessidades, inovando e criando uma grande variedade de dispositivos.
- **Flexibilidade:** o Android oferece uma plataforma flexível que permite a criação de aplicativos personalizados e a integração com diversos serviços online.
- **Personalização:** uma das maiores vantagens do Android é a sua capacidade de personalização. Os usuários podem alterar a aparência e o comportamento de seus dispositivos de acordo com suas preferências, utilizando temas, widgets e lançadores personalizados.

- **Comunidade ativa:** possui uma vasta comunidade de desenvolvedores que contribuem para a criação de aplicativos, ferramentas e atualizações para o sistema.
- **Google Play Store:** a loja de aplicativos do Google oferece milhões de aplicativos, jogos e outros conteúdos para download.
- **Integração com serviços Google:** o Android se integra perfeitamente com os serviços do Google, como Gmail, Google Maps e Google Drive.
- **Atualizações regulares:** o Google lança atualizações regulares para o Android, introduzindo novas funcionalidades, melhorias de desempenho e correções de segurança.

3.1 Infraestrutura Android

A plataforma Android, com sua base Linux, adota um modelo multiusuário para gerenciar aplicativos. Cada app funciona em uma máquina virtual própria, isolando-o dos demais e prevenindo interferências. Essa arquitetura, combinada com o uso de C/C++ para os componentes principais e Java para as APIs, proporciona um ambiente seguro e eficiente. O sistema gerencia de forma inteligente a criação e o término de processos, garantindo que os recursos do dispositivo sejam utilizados de forma otimizada.

A arquitetura de pilha de software do Android Open Source Project (AOSP) está dividida em sete camadas.

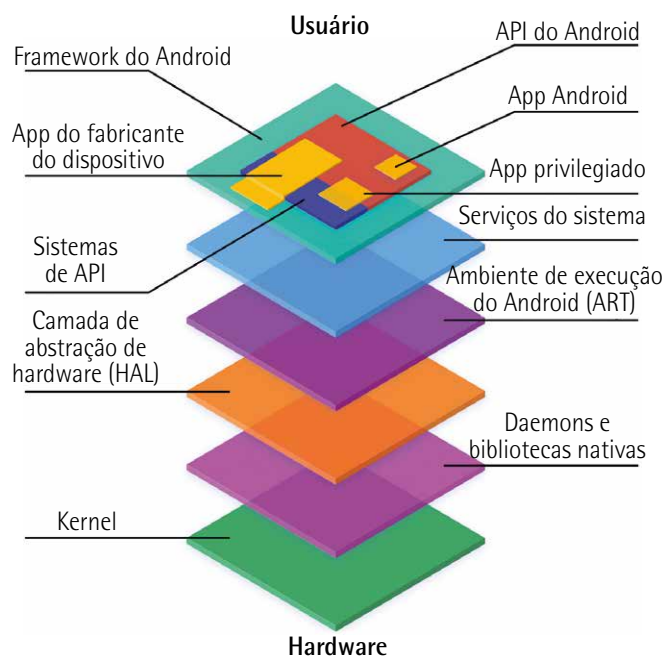


Figura 42 – A pilha de software do AOSP

Disponível em: <https://tinyurl.com/26zwm4zz>. Acesso em: 6 mar. 2025.

As camadas são:

- **App Android:** a camada app é a interface direta com o usuário no sistema operacional Android. É aqui que as aplicações que utilizamos diariamente são construídas. Inclui tanto os aplicativos pré-instalados, como tela inicial, contatos, câmera e galeria; quanto os aplicativos de terceiros baixados da Play Store, como jogos e aplicativos de bate-papo. É a camada superior da arquitetura do sistema Android. Os aplicativos desenvolvidos são executados nessa camada, utilizando as APIs fornecidas pelo Android. A camada de aplicação interage com a camada de framework de aplicação, que fornece serviços e bibliotecas necessários para o funcionamento dos aplicativos.
- **App privilegiado:** a camada de aplicativos privilegiados do Android é reservada para softwares que requerem um nível de acesso mais profundo ao sistema operacional. Esses aplicativos, desenvolvidos por fabricantes ou Google, possuem permissões especiais que permitem controlar diretamente hardware e executar funções críticas. Essa necessidade surge da complexidade dos dispositivos modernos, que exigem aplicativos capazes de interagir com hardwares específicos, como câmeras de alta resolução ou sensores biométricos. Além disso, aplicativos como o gerenciador de configurações e serviços de atualização do sistema operam nessa camada para garantir a integridade e o bom funcionamento do dispositivo.
- **Aplicação do fabricante:** também conhecida como skin ou interface personalizada, é uma extensão da camada de aplicação padrão do Android, desenvolvida por cada fabricante de dispositivos (Samsung, Xiaomi, Huawei etc.). Essa camada proporciona uma experiência de usuário única, com funcionalidades e design exclusivos, aproveitando ao máximo o hardware específico do dispositivo e diferenciando os dispositivos de uma marca para outra.
- **API System:** APIs do Android disponíveis apenas para parceiros e OEMs para inclusão em aplicativos agrupados. Essas APIs são marcadas como `@SystemApi` no código-fonte. Elas permitem a personalização do sistema, possibilitando que os fabricantes customizem a interface do usuário, adicionem funcionalidades exclusivas e integrem seus próprios serviços. Elas também otimizam o hardware, fornecendo acesso direto a componentes específicos, permitindo que os fabricantes melhorem o desempenho do dispositivo e criem experiências mais personalizadas para os usuários. Facilitam a integração de serviços proprietários, como assistentes virtuais, lojas de aplicativos e outros recursos exclusivos. Essas APIs são essenciais para o gerenciamento de dispositivos, incluindo atualizações de software, configuração de rede e outras tarefas administrativas.
- **API do Android:** a camada de API do Android é um conjunto de ferramentas e recursos que capacita os desenvolvedores a criar aplicativos para a plataforma Android. Funciona como uma ponte entre o hardware do dispositivo e os aplicativos. Essa camada oferece um conjunto de funções e serviços acessíveis aos desenvolvedores. Ao fornecer ferramentas e recursos abstratos, ela simplifica o processo de desenvolvimento, assegura a portabilidade dos aplicativos e permite a criação de aplicativos personalizados. As APIs do Android são divididas em várias categorias, incluindo:
 - **APIs de interface do usuário:** permitem a criação de interfaces gráficas, incluindo componentes como botões, listas e layouts.

- **APIs de serviços de sistema:** fornecem acesso a serviços essenciais do sistema, como gerenciamento de energia, notificações e serviços de localização.
- **APIs de acesso a dados:** incluem bibliotecas para acessar bancos de dados locais, como SQLite, e serviços de armazenamento em nuvem.
- **APIs de conectividade:** permitem a comunicação com outros dispositivos e redes, incluindo Bluetooth, Wi-Fi e NFC.
- **APIs de multimídia:** oferecem suporte para reprodução e gravação de áudio e vídeo, além de manipulação de gráficos.
- **Framework do Android:** estrutura de software que fornece um conjunto de ferramentas, bibliotecas e diretrizes organizadas de forma coerente para facilitar o desenvolvimento de aplicativos, sistemas e softwares em geral. É a base do sistema operacional. Oferece um conjunto de classes e interfaces que os desenvolvedores utilizam para criar aplicativos. É o elo entre a complexidade do sistema operacional e a lógica do aplicativo. Ao abstrair muitos dos detalhes de implementação, essa camada facilita o desenvolvimento. Composta por bibliotecas e APIs, ela oferece componentes e serviços essenciais para a construção de aplicativos. Entre esses componentes estão atividades, serviços, provedores de conteúdo e receptores de broadcast, que são os blocos de construção básicos de qualquer aplicativo Android. Ao disponibilizar classes e métodos predefinidos, a camada de framework simplifica o processo de desenvolvimento, permitindo que os desenvolvedores se concentrem na lógica do aplicativo em vez de se preocupar com os detalhes de implementação. Aqui estão alguns dos principais componentes da camada de framework do Android:
 - **Activities:** representam as telas individuais de um aplicativo.
 - **Services:** executam tarefas em segundo plano, como reproduzir música ou baixar arquivos.
 - **Content Providers:** permitem que diferentes aplicativos compartilhem dados.
 - **Broadcast Receivers:** respondem a eventos do sistema, como a chegada de uma nova mensagem.
 - **Intents:** objetos que permitem que diferentes componentes de um aplicativo ou de aplicativos diferentes se comuniquem entre si.
- **Serviços do sistema:** essa camada opera abaixo da camada de framework, interagindo diretamente com o hardware e outros componentes do sistema. Ela é composta por um conjunto de processos e serviços que funcionam em segundo plano, fornecendo funcionalidades essenciais para o funcionamento do sistema. Esses serviços são responsáveis por tarefas como:
 - **Gerenciamento de energia:** otimização do consumo da bateria, controle de brilho da tela etc.
 - **Conectividade:** gerenciamento de redes Wi-Fi, Bluetooth, dados móveis etc.
 - **Gerenciamento de processos:** controle da execução de aplicativos e serviços em segundo plano.

- **Gerenciamento de armazenamento:** controle do armazenamento interno e externo do dispositivo.
- **Gerenciamento de áudio e vídeo:** reprodução de mídia, captura de áudio e vídeo etc.
- **Localização:** serviços de GPS e localização.
- **Ambiente de execução do Android (ART):** executa os aplicativos. Ele substituiu a antiga máquina virtual Dalvik a partir do Android 5.0 (Lollipop) e é um ambiente de execução Java fornecido pelo AOSP. O ART traduz o bytecode (atua como uma ponte entre o código-fonte escrito pelo desenvolvedor e o código de máquina executável pelo hardware) do aplicativo em instruções específicas do processador, que são executadas pelo ambiente de execução do dispositivo. Quando um aplicativo é instalado, o ART compila seu bytecode em código de máquina nativo. Assim, quando o aplicativo é executado, ele roda diretamente a partir do código nativo compilado, eliminando a necessidade de interpretação ou compilação a cada execução.
- **Camada de abstração de hardware (HAL):** interface padrão que permite aos fornecedores de hardware implementar funcionalidades específicas do dispositivo sem afetar ou modificar o código das camadas superiores, proporcionando independência em relação às implementações de drivers de nível inferior. Ela permite que o software do Android interaja com diversos componentes de hardware, como câmera, sensor de movimento, Bluetooth, Wi-Fi, entre outros, de forma padronizada e independente das especificações de cada fabricante. Por exemplo, quando um aplicativo quer acessar a câmera do dispositivo, em vez de lidar diretamente com os registradores e comandos específicos da câmera, o aplicativo chama uma função da HAL que abstrai essa complexidade. A HAL chama o driver da câmera apropriado para realizar a operação.
- **Daemons e bibliotecas nativas:** os Daemons são processos que rodam em segundo plano, independentes da interface do usuário e executam tarefas específicas. No Android, eles são responsáveis por funções como gerenciamento de energia, conectividade de rede, armazenamento e outros serviços críticos. As bibliotecas nativas são conjuntos de código compilados para a arquitetura do dispositivo (ARM, x86 etc.), geralmente escritas em C ou C++. Essas bibliotecas fornecem funcionalidades específicas, como manipulação de gráficos, codecs de áudio e vídeo, e acesso a hardware.
- **Kernel do Android:** um núcleo do sistema operacional Linux, altamente personalizado e adaptado para dispositivos móveis. Ele fornece serviços essenciais para a execução de processos, gerenciamento de memória, sistema de arquivos, drivers de dispositivos e muito mais. Sendo a base de todo o sistema, o Kernel é a primeira camada de software a ser carregada quando o dispositivo é ligado, servindo como alicerce para todas as outras camadas do sistema. São algumas funções do Kernel:
 - **Gerenciamento de processos:** cria, programa e termina processos, além de gerenciar a alocação de recursos para cada processo.
 - **Gerenciamento de memória:** aloca e libera memória para os processos, garantindo que eles tenham os recursos necessários para executar.

- **Gerenciamento de dispositivos:** interage com os diversos componentes de hardware do dispositivo, como CPU, memória, armazenamento, tela, câmera e outros sensores.
- **Sistema de arquivos:** fornece uma interface para acessar e gerenciar os dados armazenados no dispositivo.
- **Interrupções:** responde a interrupções de hardware, como as geradas por dispositivos de entrada ou por eventos de temporização.
- **Comunicação interprocesso (IPC):** permite que diferentes processos se comuniquem entre si.

3.2 Plataforma de desenvolvimento

A plataforma de desenvolvimento do Android é um conjunto de ferramentas e recursos que permite aos desenvolvedores criar aplicativos para dispositivos que utilizam o sistema operacional Android.

Para entender o processo de desenvolvimento, é importante considerar várias etapas e o uso de diversas ferramentas e tecnologias. Primeiro, escolhe-se o ambiente de desenvolvimento. Em seguida, define-se a linguagem de programação e selecionam-se as bibliotecas e os frameworks necessários para o projeto. Por fim, após a conclusão do desenvolvimento, o aplicativo é distribuído.

3.2.1 Ambiente de Desenvolvimento Integrado (IDE)

Um Ambiente de Desenvolvimento Integrado (IDE) é um software que reúne diversas ferramentas e funcionalidades em uma única interface, com o objetivo de facilitar e otimizar o processo de desenvolvimento de software. Imagine-o como um estúdio completo para um programador, fornecendo todas as ferramentas necessárias para criar, testar e depurar um programa.

O Android Studio é a IDE oficial para o desenvolvimento de aplicativos Android, criada pelo Google. Ele fornece um conjunto abrangente de ferramentas e recursos que simplificam todo o processo de desenvolvimento, desde a concepção inicial até a publicação dos aplicativos na Google Play Store.

O Android Studio foi apresentado pela primeira vez em 2013 na conferência Google I/O, como um sucessor mais moderno e completo do Eclipse ADT (Android Development Tools), que era a ferramenta padrão na época. A criação de um IDE surgiu da necessidade de uma ferramenta mais integrada e eficiente para atender às crescentes demandas do desenvolvimento Android. Em 2014, foi lançada a primeira versão estável. Desde então, o Android Studio tem recebido atualizações frequentes, com novas funcionalidades, melhorias de desempenho e suporte a novas tecnologias.

O Software Development Kit (SDK) para Android é um conjunto de ferramentas, bibliotecas e APIs que os desenvolvedores utilizam para criar aplicativos para a plataforma Android. É como uma caixa de ferramentas completa, fornecendo todos os recursos necessários para construir, testar e depurar aplicativos. O Android Studio utiliza o SDK para fornecer um ambiente de desenvolvimento integrado e completo. Ao instalar o Android Studio, o SDK é instalado automaticamente ou pode ser configurado para usar um SDK existente.

3.2.2 Linguagens de programação

O Android Studio suporta várias linguagens de programação, permitindo que os desenvolvedores escrevam o código que define a lógica e o comportamento dos aplicativos. Aqui estão as principais linguagens usadas no Android:

- **Java:** por muitos anos, o Android Studio teve o Java como sua principal linguagem de programação para o desenvolvimento de aplicativos Android. Essa escolha foi motivada por razões históricas e técnicas. A plataforma Android foi construída sobre a JVM, a máquina virtual que executa o código Java, proporcionando uma base sólida e um vasto ecossistema de bibliotecas e ferramentas já existentes para Java. Java era uma linguagem madura e amplamente utilizada na indústria, com uma grande e ativa comunidade de desenvolvedores, o que facilitava a transição para o desenvolvimento Android. A compatibilidade com Java permitiu que muitos desenvolvedores familiarizados com a linguagem comesçassem a criar aplicativos Android rapidamente. No entanto, essa situação mudou com a introdução do Kotlin.
- **Kotlin:** em 2017, o Google anunciou o Kotlin como a linguagem oficial para o desenvolvimento Android. Embora o Java tenha sido a linguagem inicial, o Kotlin se tornou a linguagem preferencial devido às suas vantagens em produtividade, segurança e modernidade. A decisão do Google de adotar o Kotlin reflete a busca por um ambiente de desenvolvimento mais eficiente e expressivo. Hoje, embora ainda seja possível desenvolver aplicativos Android em Java, o Kotlin é fortemente recomendado e amplamente utilizado pela maioria dos desenvolvedores.
- **C++:** utilizada com o Android NDK (Native Development Kit), C++ desempenha um papel crucial em determinados cenários. A combinação de C++ com o Android Studio oferece vantagens únicas, especialmente em situações que exigem alto desempenho, controle preciso sobre o hardware ou compatibilidade com código legado.
- **XML:** fundamental para definir a estrutura visual de aplicativos Android. Ele utiliza tags para descrever a hierarquia de elementos da interface, como botões e textos. O Android Studio facilita a criação de layouts XML com um editor visual intuitivo. No entanto, a sintaxe XML pode ter layouts complexos. Para superar essa limitação, o Jetpack Compose surge como uma alternativa moderna e poderosa. Com sua sintaxe declarativa e funcional, ele permite que os desenvolvedores descrevam a interface de forma mais intuitiva e concisa. Ao invés de escrever código imperativo para manipular cada elemento da tela, o Compose permite que declare o estado desejado da interface e o sistema se encarrega de atualizar a tela automaticamente. Essa abordagem simplifica o desenvolvimento e torna o código mais fácil de entender e manter.

3.2.3 Bibliotecas e frameworks

As bibliotecas são pacotes de código que fornecem funcionalidades específicas, como conexão à internet, armazenamento de dados e manipulação de imagens. Podem ser adicionadas a um projeto para oferecer esses recursos sem precisar criar tudo do zero. Já os frameworks são estruturas mais completas que fornecem base para o desenvolvimento de aplicativos, definindo a arquitetura e as

melhores práticas. Oferece um conjunto de componentes e ferramentas para construir aplicativos de forma mais organizada e eficiente.

Algumas das principais bibliotecas e frameworks utilizados no Android Studio:

- **Jetpack Compose:** framework moderno para construir interfaces de usuário de forma declarativa usando Kotlin. Em vez de definir layouts em XML, os desenvolvedores escrevem funções Kotlin que descrevem a UI. Compose permite criar interfaces flexíveis e reativas, em que as mudanças no estado do aplicativo são refletidas automaticamente na UI, simplificando a atualização dos componentes visuais. Com compose, há menos necessidade de escrever muitas linhas de código para realizar tarefas básicas, resultando em um código mais limpo e conciso. Como compose é escrito em Kotlin, ele se integra perfeitamente com o restante do código do aplicativo, aproveitando todos os benefícios da linguagem Kotlin.
- **AndroidX:** extensão das bibliotecas de suporte do Android, que oferece compatibilidade com versões anteriores do sistema. Isso facilita a criação de aplicativos que funcionam em diversas versões do Android, garantindo que eles sejam compatíveis com uma ampla gama de dispositivos.
- **Glide:** biblioteca para carregar e exibir imagens de forma otimizada.
- **Room:** biblioteca Object-Relational Mapping (ORM) para acessar bancos de dados SQLite de forma mais simples.
- **LiveData e ViewModel:** componentes do Android Jetpack que ajudam a gerenciar o ciclo de vida dos componentes e a manter os dados atualizados.
- **Navigation Component:** facilita a navegação entre as diferentes telas de um aplicativo.
- **WorkManager:** agenda tarefas que devem ser executadas em segundo plano, mesmo se o aplicativo estiver fechado.

4 ACTIVITY

No Android, uma activity é basicamente uma tela do aplicativo. É onde o usuário interage com o app. Dentro de uma activity, podemos adicionar botões, imagens, menus e outros elementos. Qualquer item com o qual o usuário possa interagir estará dentro de uma activity.

Um aplicativo pode ter várias activities, mas também pode funcionar com apenas uma, dependendo da necessidade. A activity principal, geralmente chamada de MainActivity, é a primeira tela que o usuário vê ao abrir o app. Cada activity pode abrir outras activities para realizar diferentes tarefas. Por exemplo, em um app de viagens de automóveis, a activity principal pode mostrar um mapa e o local do destino. Quando clica em viagens, uma nova activity é aberta mostrando as suas viagens realizadas, ficando sobre a principal.

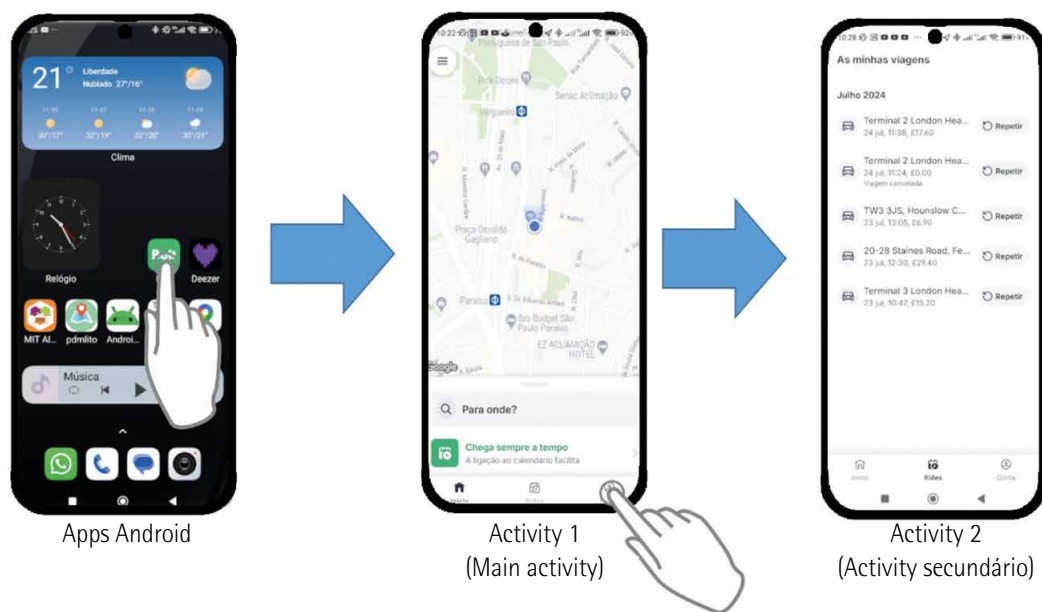


Figura 43 – Funcionamento das activities

A declaração `import android.app.Activity` dá acesso à classe base activity. Seria como comprar um carro básico e adicionar acessórios como ar-condicionado ou rodas diferentes. A declaração `import` é como adicionar esses acessórios ao código. Essas declarações são comuns e podem aparecer várias vezes no mesmo código.

4.1 Intents e intent filters

Um intent é como uma mensagem enviada para outro componente do aplicativo, pedindo para ele realizar uma tarefa específica. Essa é a principal forma de comunicação entre diferentes partes do seu app. Temos três utilidades principais do intent:

- **Iniciando uma nova tela (activity):** ao clicar em um botão que leva para outra tela, envia-se uma mensagem (intent) para o sistema Android, pedindo para abrir uma nova tela. Essa mensagem contém todas as informações necessárias para que a nova tela seja exibida corretamente.
- **Iniciando um serviço em segundo plano:** para iniciar uma tarefa em segundo plano, como baixar um arquivo, também usa um intent. Nesse caso, a mensagem é enviada para o sistema, solicitando que um serviço seja iniciado para realizar essa tarefa sem interromper a experiência do usuário.
- **Enviando uma mensagem para outros apps:** as transmissões são como avisos públicos que qualquer app pode receber. É como um rádio: ao transmitir uma mensagem, qualquer app ligado nessa frequência pode ouvi-la. Quando se tira uma foto, o sistema envia uma transmissão para todos os apps, informando a nova imagem. Além disso, você pode criar suas próprias transmissões para notificar outros apps sobre eventos específicos do seu aplicativo, como a conclusão de um download ou a mudança de um determinado estado. Essa flexibilidade permite a integração de seu app com outros de forma personalizada.

Existem dois tipos de intents: explícitos e implícitos.

- **Intents explícitos:** indica exatamente qual componente do app deve ser executado. É como dar um endereço completo para alguém. São ideais para navegação interna no app ou para iniciar serviços específicos.
- **Intents implícitos:** descreve-se a ação que deseja realizar e o sistema Android encontra o app mais adequado para essa tarefa. É como pedir para alguém levá-lo ao aeroporto sem especificar o táxi. São úteis para delegar tarefas a outros apps, como abrir um mapa, enviar um e-mail ou visualizar uma imagem.

Na figura 44 temos um exemplo trabalhando com um intent implícito e enviado pelo sistema para iniciar outra atividade:

- A atividade A cria um intent com uma descrição da ação e o envia para `startActivity()`.
- O sistema Android procura em todos os aplicativos um filtro de intent que corresponda à intent.
- Quando uma correspondência for encontrada, o sistema inicia a atividade correspondente (atividade B) chamando o método `onCreate()` e passando o intent.

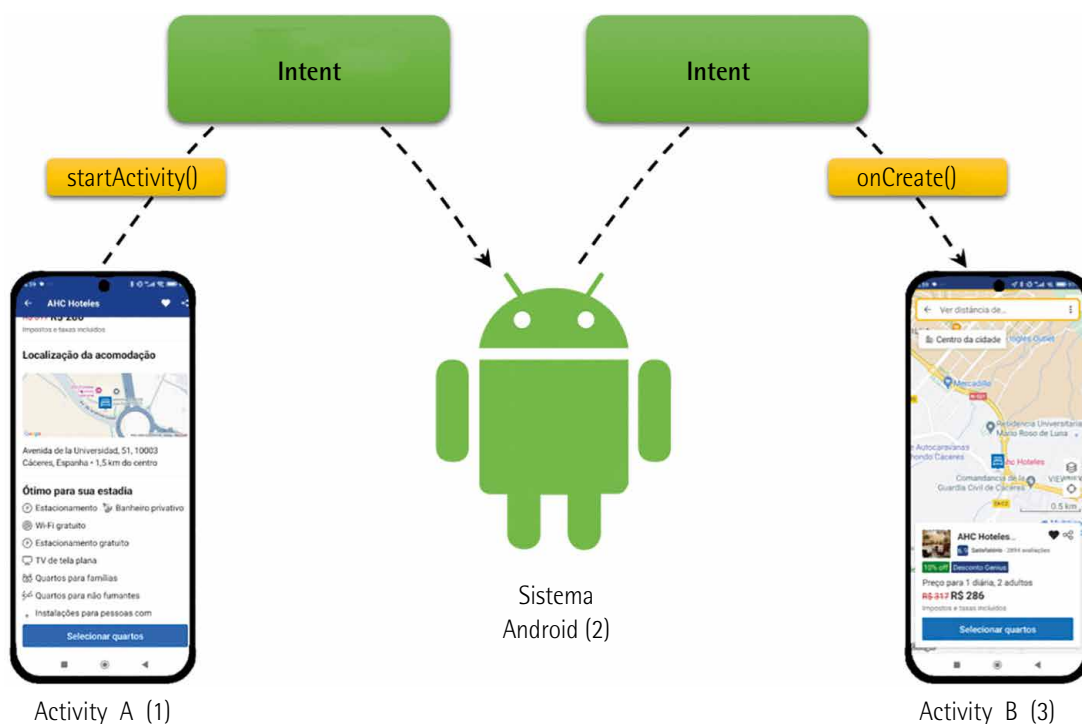


Figura 44 – Exemplo de intent

4.2 Componentes

Além da *activity* que vimos, o Android possui mais três componentes, os quais são os blocos de construção essenciais de um aplicativo Android. Cada um deles serve como um ponto de entrada pelo qual o sistema ou o usuário pode interagir com o aplicativo. Organizando, os componentes são: *activity*, serviços, broadcast receivers e content providers.

- **Atividades (activity):** componentes visuais de um aplicativo Android, responsáveis por apresentar a interface do usuário. Elas gerenciam a interação com o usuário por elementos como botões e campos de texto. Cada *activity* opera de forma independente, mas trabalha em conjunto, formando a jornada do usuário dentro do aplicativo.
- **Serviços (services):** componentes do Android que operam em segundo plano, permitindo a execução de tarefas de longa duração sem interromper a experiência do usuário. Eles executam tarefas importantes, como baixar músicas ou sincronizar dados, enquanto o usuário utiliza outras partes do aplicativo. Eles podem ser iniciados e executados de forma independente ou oferecer uma interface para outros componentes interagirem com eles. Existem dois tipos principais: serviços iniciados, que continuam em execução até serem explicitamente parados; e serviços vinculados, que fornecem um mecanismo para que outros componentes se conectem e solicitem serviços.
- **Broadcast receivers:** componentes do Android que monitoram e respondem a transmissões (broadcasts) enviadas pelo sistema ou por outros aplicativos. Eles permitem que um aplicativo seja notificado e execute ações específicas em resposta a eventos do sistema, como mudanças no estado da bateria ou a chegada de uma nova mensagem. Os broadcast receivers facilitam a comunicação assíncrona, sem a necessidade de interação em tempo real, entre aplicativos e o sistema, permitindo que os aplicativos reajam a eventos sem precisar estar em execução contínua. Por exemplo, um aplicativo pode usar um broadcast receiver para responder a eventos como a chegada de uma mensagem SMS ou a mudança no estado da rede.
- **Content providers:** componentes que fornecem uma interface padronizada para acessar e gerenciar dados compartilhados entre diferentes aplicativos. Eles atuam como uma camada de abstração sobre os dados, permitindo que outros aplicativos interajam com eles de forma segura e controlada. Por exemplo, um content provider pode ser usado para acessar os contatos do dispositivo ou para compartilhar dados entre diferentes aplicativos.



Saiba mais

Para mais detalhes, veja o site oficial do Android:

DEVELOPERS. *Fundamentos de aplicativos*. [s.d.]. Disponível em: <https://shre.ink/M6o9>. Acesso em: 6 mar. 2025.

4.3 Ciclo de vida

O ciclo de vida de uma atividade é a sequência de estados por que uma atividade passa durante seu tempo de execução em um aplicativo Android desde a criação até a destruição. Imagine uma atividade como uma tela em um aplicativo: ela pode ser criada, exibida, pausada, retomada e destruída. Cada uma dessas transições é acompanhada por chamadas a métodos específicos, que permitem a execução de ações personalizadas em cada etapa do ciclo.

Na figura 45 temos o ciclo de vida de uma activity que engloba os estados de criação, início, pausa, interrupção e destruição. A configuração inicial da activity ocorre no momento da criação. Após ser iniciada, ela se torna visível ao usuário e permanece ativa até ser pausada ou interrompida, por exemplo, quando é vista atrás de uma janela de diálogo. Quando é interrompida, não fica visível para o usuário. Por último, ela será destruída.

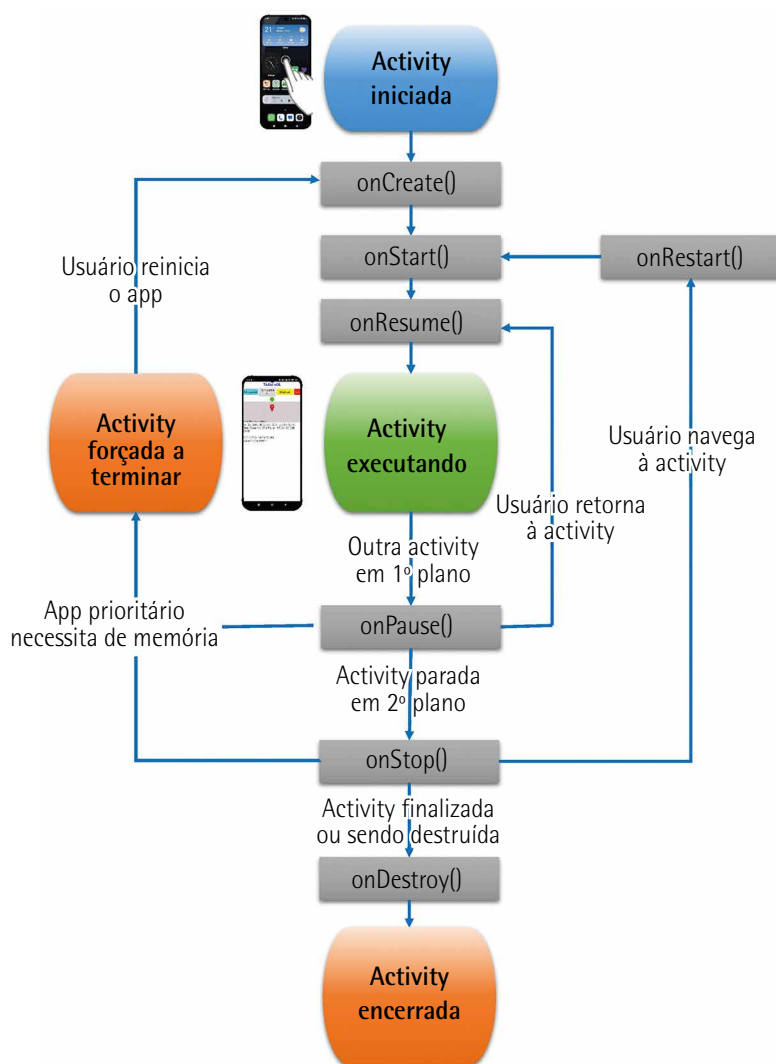


Figura 45 – Ciclo de vida de uma activity

No ciclo de vida de uma activity, existem métodos especiais conhecidos como callbacks, que são chamados pelo sistema quando a activity muda de estado. Esses callbacks permitem que se gerencie o comportamento da activity em resposta a essas mudanças, garantindo que o aplicativo funcione de maneira eficiente e responsiva. Eles são úteis para o gerenciamento de recursos, como liberar memória quando a activity não está mais visível, otimizando o desempenho do aplicativo.

Os callbacks também são utilizados para a preservação de estado, permitindo salvar o estado da activity antes que ela seja pausada ou destruída, garantindo que o usuário não perca seus dados ao retornar à activity. Além disso, permitem que o aplicativo responda a eventos do sistema, como mudanças na orientação da tela ou falta de memória. Aqui estão os principais callbacks do ciclo de vida da activity:

- **onCreate()**: chamado quando a activity é criada pela primeira vez. É onde você deve inicializar os componentes essenciais da activity, como a interface do usuário.
- **onStart()**: chamado quando a activity está prestes a se tornar visível para o usuário. O método onStart é executado logo após o onCreate. A grande diferença é que, durante o onCreate, o aplicativo não está visível ao usuário, mas no onStart sim. Esse método é executado rapidamente, assim como o onCreate e o aplicativo não fica parado nesse estado. Logo após a execução do onStart, o aplicativo passa para o estado de resumido, chamando o método onResume. Aqui, pode realizar configurações que devem estar prontas quando a activity se tornar visível.
- **onRestart()**: chamado quando a activity foi parada e está sendo reiniciada. Isso ocorre, por exemplo, quando o usuário volta para a activity após ela ter sido interrompida.
- **onResume()**: chamado quando a activity começa a interagir com o usuário. Neste estado, a activity está no topo da pilha e pronta para receber a entrada do usuário.
- **onPause()**: chamado quando a activity está parcialmente visível, geralmente quando outra activity está em primeiro plano. O método é acionado pelo sistema como um primeiro indicador de que o usuário está saindo do aplicativo, isso não quer dizer que o usuário está fechando-o, ele pode abrir outro aplicativo, deixando este em segundo plano. Aqui, deve salvar o estado atual da activity.
- **onStop()**: chamado quando a activity não está mais visível para o usuário. É um bom momento para liberar recursos desnecessários enquanto a activity não está visível.
- **onDestroy()**: acionado pelo sistema momentos antes de a activity ser destruída, ou seja, totalmente finalizada. Isso pode acontecer quando o usuário encerra totalmente o aplicativo ou quando o aplicativo está no estado parado e o sistema precisa de memória. Ele pode eleger sua activity para destruir e ganhar um pouco de espaço. É o último callback que a activity recebe, em que se pode realizar a limpeza final de recursos.



Saiba mais

Para mais detalhes sobre o ciclo de vida, veja o site oficial do Android:

DEVELOPERS. *Ciclo de vida da atividade*. [s.d.]b. Disponível em: <https://shre.ink/M6At>. Acesso em: 6 mar. 2025.

No Jetpack Compose, o gerenciamento do ciclo de vida é tratado de maneira diferente. Compose é um framework declarativo, em que se descreve a interface do usuário em termos de seu estado atual e o Compose atualiza a interface do usuário quando o estado muda. Em vez de usar callbacks do ciclo de vida como `onCreate()` e `onResume()`, define-se composables que reagem automaticamente às mudanças de estado.

4.4 Layout

No Android tradicional, uma activity possui um layout que define a interface do usuário. Esse layout é geralmente descrito em arquivos XML, em que se define a estrutura da interface usando uma hierarquia de Views e ViewGroups. Por exemplo, pode ter um arquivo XML que define um `ConstraintLayout` contendo um `TextView`. Esse layout é carregado na Activity usando o método `setContentView()` dentro do callback `onCreate()`.

No exemplo da figura 46, pode-se ter um arquivo XML que define um `LinearLayout` contendo um `TextView` e um `Button`. Esse layout é carregado na activity usando o método `setContentView()` dentro do callback `onCreate()`.

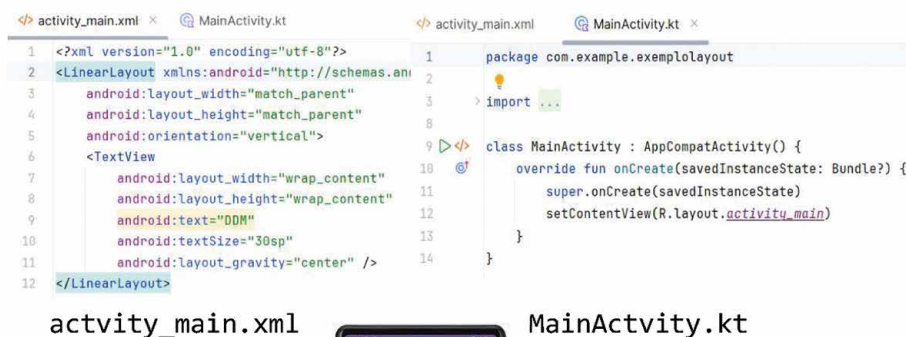


Figura 46 – Uso do layout em uma activity

Com o Jetpack Compose, o processo é diferente porque ele utiliza uma abordagem declarativa para construir a interface do usuário. Em vez de definir a interface em XML, escreve funções Kotlin chamadas composables que descrevem a interface do usuário de forma declarativa. No exemplo da figura 47, a função `TelaActivity1` é uma composable que define uma coluna contendo um texto e um botão. Em vez de usar `setContentView()`, usa-se o método `setContent` para definir o conteúdo da activity com a composable `TelaActivity1`. É menos trabalhoso utilizar o Jetpack Compose.

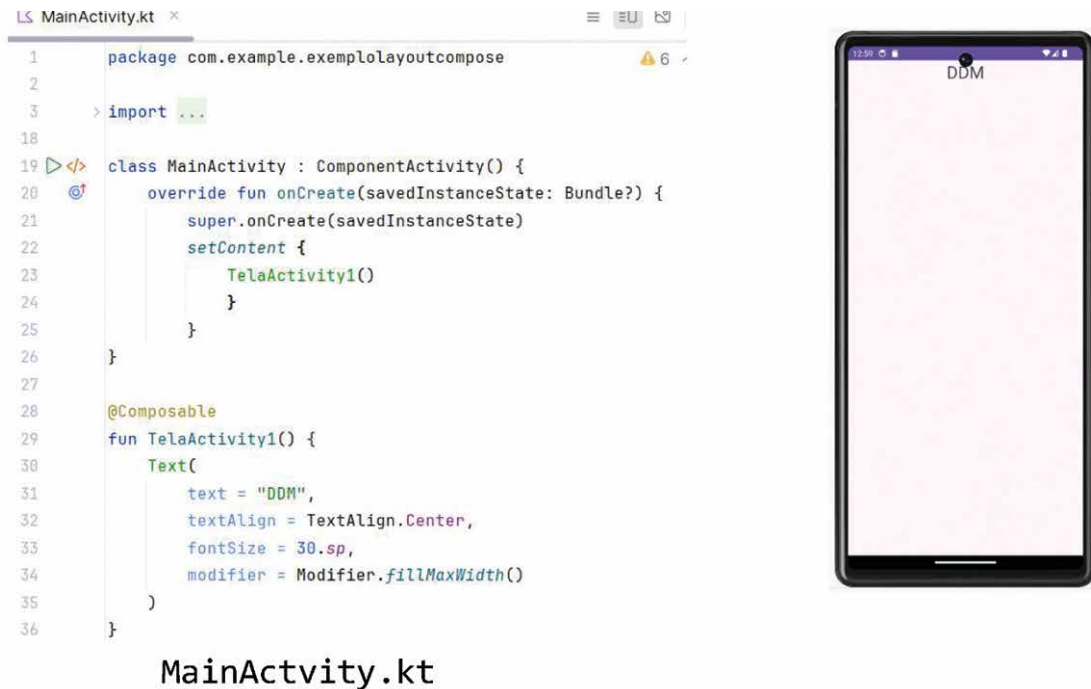


Figura 47 – Tela da activity utilizando Jetpack Compose

4.5 Widgets

Os widgets são amplamente reconhecidos como elementos essenciais para a personalização da tela inicial. Eles funcionam como visualizações compactas de dados e funcionalidades mais importantes de um aplicativo, acessíveis diretamente na tela inicial do usuário. Os usuários podem mover os widgets entre os painéis da tela inicial e, quando possível, redimensioná-los para ajustar a quantidade de informações exibidas, conforme suas preferências.

No desenvolvimento de um aplicativo, a interface do usuário é construída como uma árvore, em que cada elemento visual, ou widget, é um nó. Essa estrutura hierárquica, chamada de hierarquia de visualização, permite criar layouts complexos e personalizados. A raiz da árvore é o layout principal e os demais widgets são seus filhos, netos e assim por diante. Essa hierarquia garante a correta disposição e relacionamento entre os elementos visuais, otimizando o desenho da interface.

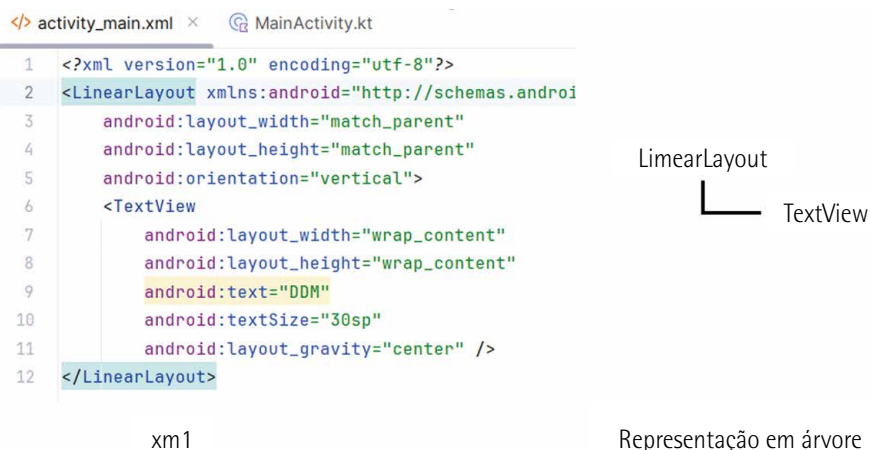


Figura 48 – Árvore de hierarquia

Na figura 48 há o exemplo de um layout simples com um texto dentro de um layout linear. O LinearLayout é o nó raiz, o layout principal, e o TextView é o widget, filho do LinearLayout, ou seja, está dentro do layout linear.

4.6 Temas

Estilos no Android definem a aparência de elementos individuais na interface, como botões ou textos. Temas aplicam essas definições a componentes maiores, como atividades inteiras ou até mesmo o aplicativo todo. Ambos são declarados em arquivos XML e permitem personalizar a interface de forma eficiente. Estilos e temas são declarados em um arquivo de recursos de estilo localizado em res/values/.

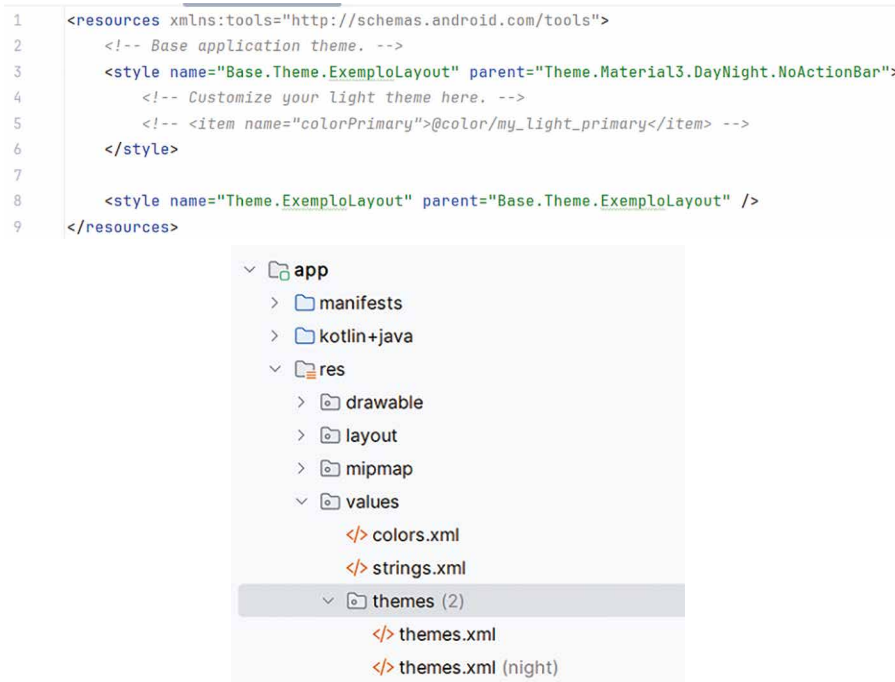


Figura 49 – Arquivo XML em que temas podem ser personalizados

Os temas garantem uma consistência visual, proporcionando uma aparência uniforme em toda a aplicação. Eles permitem que os desenvolvedores adaptem a aparência do aplicativo para refletir a identidade da marca. Além disso, os temas melhoram a acessibilidade, aprimorando a experiência do usuário ao permitir ajustes como modos claro e escuro para diferentes condições de iluminação e necessidades visuais.



Saiba mais

Uma excelente explicação sobre como decidir e aplicar os temas pode ser visto consultando:

DEVELOPERS. *Temas*. [s.d.].c. Disponível em: <https://shre.ink/M63T>. Acesso em: 6 mar. 2025.

Usando Jetpack Compose, o arquivo `theme.kt` é essencial no desenvolvimento de aplicativos Android, pois ele define e gerencia a aparência e o comportamento temático da aplicação. Exploraremos como ele funciona e como é utilizado para personalizar a interface do usuário do seu aplicativo. Nele, definem-se as principais características do seu aplicativo:

- **CompositionLocalProvider**: compartilha dados entre composables sem a necessidade de passá-los explicitamente em cada nível da árvore de composição.
- **MaterialTheme**: aplica o tema definido aos componentes dentro de seu escopo.
- **Colors DarkColorPalette e LightColorPalette**: definem as cores para o modo escuro e claro, respectivamente. Nada impede a criação de cores de alto contraste.
- **Typography**: o `TextStyle` define o estilo do texto, incluindo fonte, tamanho, cor e outros atributos.
- **Shapes**: definem as formas dos componentes, como cantos arredondados, formas personalizadas etc.

Na figura 50, o tema é definido no arquivo `theme.kt`, localizado na pasta `ui.theme`. Esse tema é utilizado para formatar a tela `Activity1` do programa principal. Note que o `MeuTema` envolve a chamada da `Activity1`.

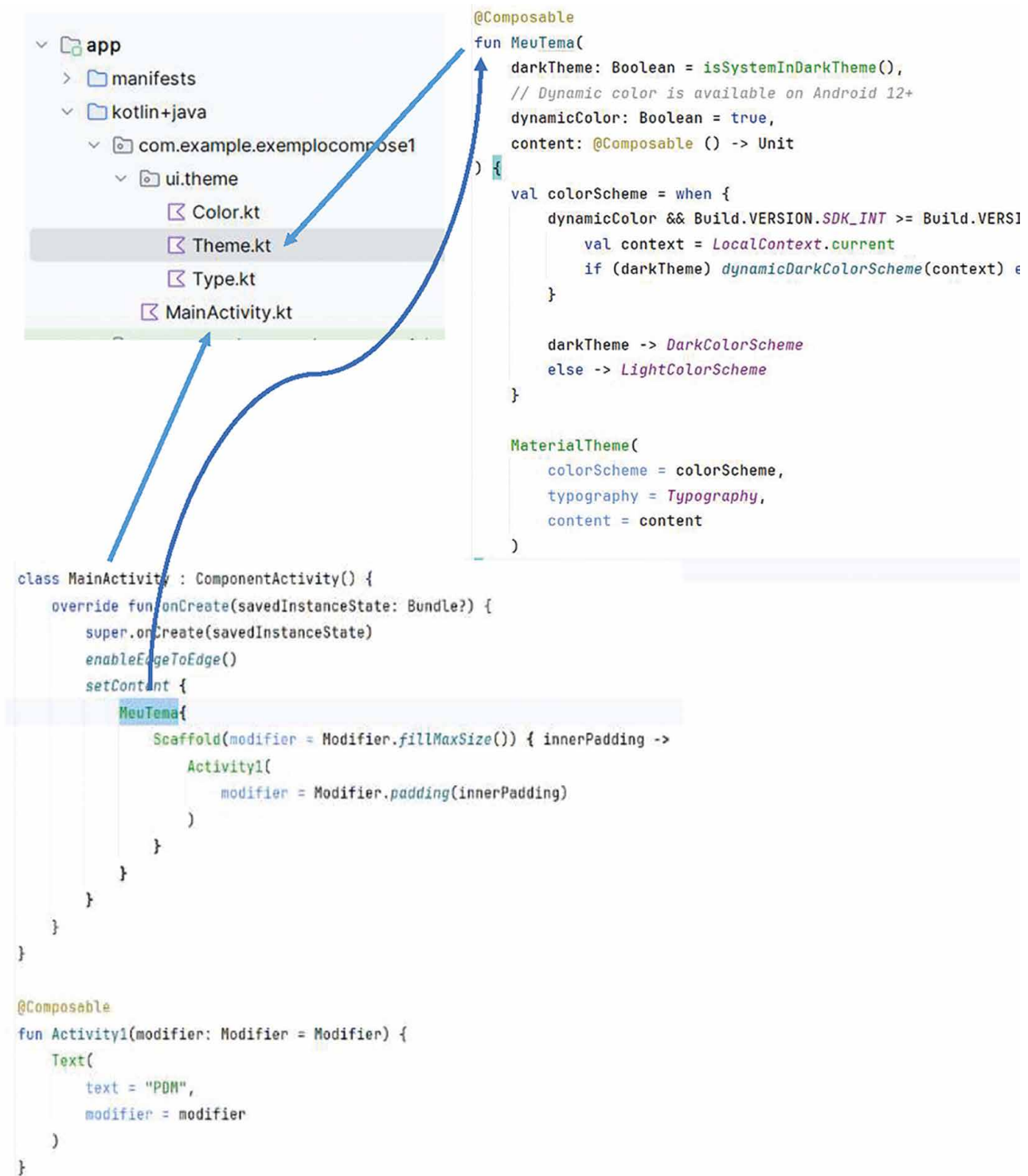


Figura 50 – Construção do Android para montar o tema de um aplicativo



Resumo

Esta unidade começa com uma introdução ao desenvolvimento de aplicativos móveis, abordando conceito, histórico, aplicações e suas características. Vimos como os aplicativos móveis evoluíram desde simples ferramentas até soluções complexas que atendem a diversas necessidades dos usuários. As características dos aplicativos móveis incluem a capacidade de operar em diferentes sistemas operacionais e a integração com sensores e serviços dos dispositivos móveis, proporcionando uma experiência personalizada.

Em seguida, estudamos os conceitos de orientação a objetos e uma introdução à programação estrutural. Foram detalhados os principais elementos da programação orientada a objetos, como objetos, classes, atributos, métodos, herança e polimorfismo. Esses conceitos são fundamentais para a criação de aplicativos baseada em views, facilitando a sua construção. A orientação a objetos permite que os desenvolvedores criem estruturas de código que refletem melhor o mundo real, tornando o desenvolvimento mais intuitivo e organizado.

Por fim, a unidade apresenta uma introdução ao Android, destacando sua infraestrutura e plataforma de desenvolvimento. Também aborda os componentes essenciais do Android, como activities, intents, layouts e widgets. Estudamos o ciclo de vida das activities e a importância dos temas para garantir uma interface de usuário consistente. Esses conhecimentos são cruciais para qualquer desenvolvedor que deseja criar aplicativos robustos e eficientes para a plataforma Android.



Exercícios

Questão 1. (Cesgranrio 2021, adaptada) Atualmente, a linguagem Kotlin é considerada a principal linguagem de programação para o desenvolvimento de aplicativos Android. Ela oferece uma variedade de tipos de dados para representar diferentes tipos de informações.

Suponha que na linguagem Kotlin queremos criar uma variável cujo valor nunca pode ser mudado. Na prática, temos uma constante. O nome da variável deve ser `idademinima`, seu tipo básico será inteiro de 32 bits e seu valor deve ser 18. Para que isso aconteça, qual das seguintes instruções devemos usar?

- A) `val idademinima: Int = 18.`
- B) `val idademinima: Integer = 18.`
- C) `val idademinima = 18: Integer.`
- D) `var idademinima: Int = 18.`
- E) `var idademinima: Integer = 18.`

Resposta correta: alternativa A.

Análise da questão

Na linguagem Kotlin, o comando de declaração de uma variável depende de sua mutabilidade. Para variáveis mutáveis, usamos a palavra-chave `var` no início da declaração. Para variáveis imutáveis, usamos a palavra-chave `val`. Na questão, queremos declarar uma constante (variável imutável) e, portanto, devemos usar a palavra `val` no início do comando.

A sintaxe de declaração de uma variável imutável é a exposta a seguir.

```
val nome_variável: Tipo = valor
```

O tipo de dado primitivo que representa números inteiros é o `Int`. Após o operador de atribuição, `=`, devemos posicionar o valor associado à variável. O comando completo de declaração da variável do enunciado é exposto a seguir:

```
val idademinima: Int = 18
```

Questão 2. (FGV 2025, adaptada) A programação orientada a objetos (POO) é um paradigma que organiza o código em torno de objetos e classes e utiliza conceitos como herança e polimorfismo para promover a reutilização e a flexibilidade.

Assinale a opção que descreve corretamente o conceito de polimorfismo em POO.

- A) É a capacidade de uma classe herdar os atributos e os métodos de outra classe, permitindo a reutilização de código.
- B) É a capacidade de criar múltiplas classes que compartilham o mesmo nome, mas têm implementações completamente diferentes.
- C) É a capacidade de um método ou um objeto se comportar de diferentes maneiras dependendo do contexto ou do tipo do objeto.
- D) É a técnica de usar métodos com o mesmo nome, mas assinaturas diferentes em uma mesma classe.
- E) É a técnica de restringir o acesso aos atributos de uma classe, garantindo a segurança dos dados.

Resposta correta: alternativa C.

Análise da questão

O polimorfismo, um dos pilares da programação orientada a objetos, permite que métodos ou objetos se comportem de formas diferentes com base no contexto ou no tipo do objeto. Há dois tipos principais de polimorfismo, elencados a seguir.

Polimorfismo de sobrecarga: ocorre quando múltiplos métodos com o mesmo nome, mas com parâmetros diferentes, coexistem na mesma classe. Isso permite que um método assuma comportamentos distintos, em função dos diferentes parâmetros que ele recebe.

Polimorfismo de sobreposição: ocorre quando um método de uma classe-filha sobrepõe um método homônimo de uma classe-mãe. Isso é especialmente útil para personalizar o comportamento de métodos em subclasses.
