Logotipo

Descrição gerada automaticamente

UNIVERSIDADE PAULISTA – UNIP

CIÊNCIAS DA COMPUTAÇÃO

CAMPUS SÃO JOSÉ DO RIO PRETO

“COMPARAÇÃO ENTRE KOTLIN E FLUTTER NO DESENVOLVIMENTO DE APLICAÇÕES MOBILE”

JOÃO VITOR RODRIGUES SABINO – N011FJ0

PEDRO HENRIQUE MEDINA DE SOUZA – N011FF7

ABNER MIGUEL MUNHOZ RODRIGUES - N014180

SÃO JOSÉ DO RIO PRETO – SP

2025

**LISTA DE FIGURAS**

[Figura 1 - Distribuição de um código criado em Java para diversos dispositivos. 11](#_Toc197899120)

[Figura 2 – Atualização do Widget no Flutter 16](#_Toc197899121)

[Figura 3 - Material 3 Widgets 19](#_Toc197899122)

[Figura 4 – Exemplificação de um hash SHA-1 24](#_Toc197899123)

**LISTA DE SÍMBOLOS, NOMECLATURAS E ABREVIAÇÕES**

*3D – Tridimensional*

*ADB - Android Debug Bridge*

*AOSP - Android Open Source Project*

*AOT – Ahead-of-Time*

*API - Application Programming Interface*

*APP – Application*

*AVD – Android Virtual Device*

*GC – Garbage Collection*

*GPS – Global Positioning System*

*I/O - Input/Output*

*IDE - Integrated Development Environment*

*IOS – Iphone Operating System*

*JIT – Just-in-Time*

*JIT – Just-In-Time*

*JVM – Java Virtual Machine*

*PIB – Produto Interno Bruto*

*RAM – Random Acess Memory*

*SDK – Software Development Kit*

*UI – User Interface*

*UTF – Unicode Transformation Format*

*VCS - Version Control System*

*VM – Virtual Machine*

**SUMÁRIO**

[1 INTRODUÇÃO 5](#_Toc198236903)

[1.1 Objetivo Geral 7](#_Toc198236904)

[1.2 Objetivo Específicos 7](#_Toc198236905)

[1.3 Justificativa 8](#_Toc198236906)

[1.4 Metodologia 9](#_Toc198236907)

[2 TECNOLOGIAS 9](#_Toc198236908)

[2.1 Java 9](#_Toc198236909)

[2.2 Funcionamento do Java 10](#_Toc198236910)

[2.3 Kotlin 11](#_Toc198236911)

[2.3.1 História do *Kotlin*. 11](#_Toc198236912)

[2.3.2 Funcionamento do *Kotlin* 12](#_Toc198236913)

[2.4 Dart 12](#_Toc198236914)

[2.4.1 Comparação entre *Java* e *Dart* 13](#_Toc198236915)

[*2.4.2* *Dart SDK* 14](#_Toc198236916)

[2.5 Flutter 15](#_Toc198236917)

[*2.5.1* *Como Flutter funciona?* 15](#_Toc198236918)

[2.5.2 Desenvolvimento de aplicações multiplataformas 16](#_Toc198236919)

[2.5.3 Vantagens da programação com *Flutter* em projetos multiplataformas 16](#_Toc198236920)

[2.6 Material Design 17](#_Toc198236921)

[2.6.1 *Jetpack Compose* 19](#_Toc198236922)

[2.7 Android 20](#_Toc198236923)

[2.7.1 *Android* SDK 21](#_Toc198236924)

[2.8 Android Emulator 22](#_Toc198236925)

[2.9 Git 22](#_Toc198236926)

[*2.9.1* *GitHub* 24](#_Toc198236927)

[3 REFERÊNCIAS 25](#_Toc198236928)

# INTRODUÇÃO

Através de um relatório anual liberado pela *GSMA*, no fim de 2023, 5.6 bilhões de pessoas (cerca de 69% da população mundial) se inscreveu em um serviço de internet móvel, demonstrando um crescimento de 1.6 bilhões de pessoas desde 2015. O crescimento do envolvimento da internet móvel cotidiano se mostrou ainda mais rápido, ainda no fim de 2023, cerca de 58% da população mundial usou serviços móveis, sendo o equivalente a 4.7 bilhões de usuários, e demonstrando um aumento de 2.1 bilhões desde 2015 (GSMA, 2024).

O impacto do uso dos celulares não se limitou meramente as questões de usabilidade no cotidiano das pessoas, mas sim gerou uma grande mudança na economia. Em 2023 foi evidenciado a importância dos dispositivos móveis no PIB global onde a contribuição totalizou 5,7 trilhões de dólares em valor econômico agregado e gerou cerca de 35 milhões de empregos (GSMA, 2024).

Em 2022 através de uma pesquisa com o objetivo de coletar informações sobre o uso da internet durante a pandemia, foi revelado que 77% das pessoas pertencentes a classes sociais D (renda familiar entre 1 e 3 salários-mínimos) e E (renda familiar de até 1 salário-mínimo) acessavam a internet apenas pelo celular. Em relação a atividades realizadas na internet 83% dos usuários que frequentam escola ou universidade realizaram atividades ou pesquisas escolares pela internet, 89% assistiram a filmes ou séries on-line 86% ouviram música on-line, 51% dos usuários de Internet compraram produtos e serviços pela Internet, e 81% dos usuários de Internet das classes AB e 30% dos das classes DE compraram produtos e serviços pela Internet. Ao mesmo tempo, informações da *GSMA Intelligence* indicam que 97,1% das conexões móveis no Brasil já podem ser classificadas como “banda larga”, ou seja, utilizam redes móveis 3G, 4G ou 5G para se conectar (CETIC, NIC, CGI, 2022; KEMP, 2025).

A partir do que foi constato pode se afirmar que dispositivos móveis são não só de grande importância para o cotidiano, como representam grande parte do acesso frequente a internet.

Portanto, o conhecimento de qual tecnologia melhor pode ser aplicada na área de desenvolvimento de serviços móveis se torna imprescindível, dada a grande diferença que variadas linguagens e ferramentas tem ao serem aplicadas na construção de aplicativos móveis.

Empresas tanto de grande e médio porte podem sentir diferenças gritantes em seu desempenho e ganho financeiro ao escolher, erroneamente ou assertivamente que linguagem aplicar na criação de um aplicativo para *mobile*. Diante do exposto uma comparação que demonstre a diferença entre ferramentas muito utilizadas no ambiente de desenvolvimento de aplicativos móveis, se torna de grande relevância, tornando assim, a base comparativa desse projeto de avaliar as especificidades e dissemelhanças, entre o *Flutter* e o *Kotlin* de grande valor.

## Objetivo Geral

O objetivo do trabalho é trazer uma comparação entre duas tecnologias para o desenvolvimento de aplicações *mobile*, *Flutter* e *Kotlin*. O projeto visa criar uma análise aprofundada das principais características que diferenciam essas tecnologias, comparando as semelhanças e diferenças que cada linguagem apresenta em desempenho, eficiência e uso. Apresentando assim as vantagens e desvantagens do uso de cada tecnologia.

## Objetivo Específicos

Construir uma comparação que demonstre as diferenças entre o *framework Flutter*, e o *Kotlin*, uma linguagem utilizada para desenvolvimento de aplicativos nativos *Android* e como ambos organizam e compõem uma aplicação feita para o ambiente *Android*.

O projeto irá criar uma análise que levante as diferenças entre as duas tecnologias, com o intuito de demonstrar as especificidades de cada linguagem, e assim instruir qualquer indivíduo que deseje saber sobre qual recurso será melhor em determinada situação. Seja por diferenças de desempenho e velocidade ou pela facilidade do uso de certos recursos presentes em ambientes *mobile*.

A primeira comparação a ser feita será a criação de uma aplicação que permita ao usuário visualizar e interagir com animações e objetos em 3D/2D. Dois aplicativos serão criados, um utilizando *Flutter* e outro o *Kotlin.* A aplicação trará as diferenças de como ambas as linguagens lidam com recursos 3D/2D, animações em tela, e as altas taxas de atualização.

A segunda aplicação irá demonstrar como o *Flutter* e o *Kotlin* lidam com os recursos de *hardware* de um dispositivo *Android*, sendo utilizados: Conexão *Wi-Fi*, *Bluetooth*, câmera, *feedback* tátil, microfone e áudio, GPS, *flash*, sensores de movimento (giroscópio), e leitor biométrico. Cada aplicativo trará telas que implementam as tecnologias do *Android*, demonstrando as diferenças tanto na usabilidade, quanto na disponibilidade de recursos adicionais, e nas facilidades ou dificuldades de integração e utilização desses recursos.

## Justificativa

O *Kotlin* é uma linguagem muito popular, em 2024 cerca de 2.5 milhões de desenvolvedores utilizaram o *Kotlin* e mais de 400 universidades utilizam na sua grade de ensino, sendo a quinta linguagem com maior crescimento no *GitHub*, onde possui 1.65 milhões de repositórios que utilizam *Kotlin* como sua linguagem primária (KOTLIN FOUNDATION, 2024).

O mesmo pode ser dito do *Flutter* que, desde o seu lançamento, demonstrou um crescimento significativo e consistente. Com cerca de 46% dos desenvolvedores multiplataforma optando pelo *Flutter* como seu *framework* para desenvolvimento de aplicações, possuindo ativamente mais de 2 milhões de desenvolvedores ao redor do mundo, e responsável por impulsionar mais de 30% de todos os novos aplicativos em dispositivos IOS. Responsável pela melhora na eficiência no desenvolvimento de diversas empresas de grande porte como *eBay*, *BMW*, *Tencent*, *Universal Studios* e *Nubank* (FLUTTER, 2025; THOMSEN, 2024; STATISTA, 2023).

Portanto sendo o *Flutter* e *Kotlin* ferramentas muito utilizadas uma análise comparativa entre os dois se mostra muito valiosa, para que seja possível se ter uma compreensão sobre como essas ferramentas se comportam e em quais situações é mais vantajoso se utilizar de uma ou de outra.

Para que seja feita a comparação entre as duas linguagens serão feitos dois programas. Um terá como foco a utilização de recursos de animação e interação com objetos 3D/2D, sendo avaliado quais bibliotecas e recursos estão disponíveis em cada plataforma para auxiliar o desenvolvimento. A segunda aplicação constituirá em um sistema que aplique diversos recursos nativos do *Android*, e mostre as diferenças de implementação e utilização dessas funcionalidades no *Flutter* e no *Kotlin*.

## Metodologia

O projeto irá criar uma comparação entre o *Flutter*, um *framework* que possui como linguagem o *Dart*, e o *Kotlin* uma linguagem nativa *Android* baseada no *Java*. Serão desenvolvidas duas aplicações, a primeira irá utilizar de bibliotecas nativas ou externas das plataformas para construir uma interface visual, a segunda apresentará várias telas que permitem o uso de recursos de *hardware* nativos do *Android*. Ao todo serão feitas quatro aplicações, que irão utilizar a *VM* do *Android* para computador e o próprio sistema do *Android*, para executar.

Será feita uma análise comparativa levando em consideração diversos fatores, com métricas e parâmetros sendo feitos para definir as diferenças entre as duas tecnologias.

# TECNOLOGIAS

Este capítulo mostra as principais linguagens e tecnologias utilizadas no desenvolvimento de aplicativos *mobile Android*, com foco em *Java*, *Kotlin*, *Dart*, *Flutter* e outras ferramentas para auxiliar no desenvolvimento, como *Git* e *Github*. Serão abordadas suas origens, funcionamentos e aplicações em dois diferentes tipos de aplicativo.

## Java

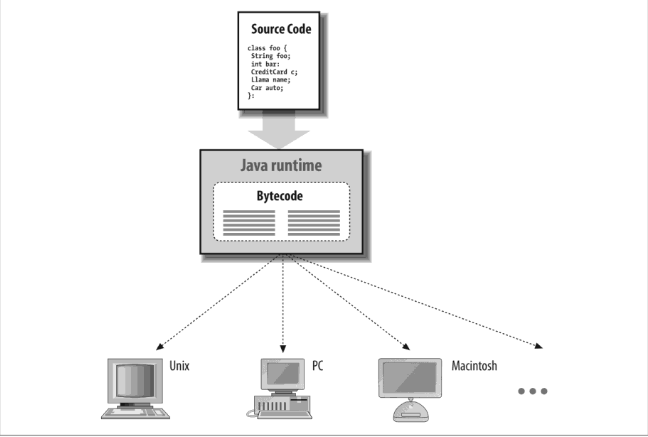
*Java* é uma linguagem de programação criada em 1995 pela *Sun Microsystems* com o principal objetivo de facilitar a criação de aplicações que pudessem ser executadas em diferentes plataformas. Logo em seu lançamento, o *Java* foi muito aclamado e elogiado, sendo considerado como "a linguagem moderna”. Pouco tempo depois, em 2003, a *Google* escolhe a linguagem como padrão para o desenvolvimento de aplicativos *mobiles* pelo *Android*. Com seu sucesso crescente e o aumento indiscutível de ferramentas e programas que utilizavam o *Java*, a *Oracle* adquiriu a *Sun Microsystems* em 2008, propondo melhorar e expandir ainda mais a linguagem (JAVA, 2025; NIEMEYER e KNUDSEN, 2005; THIAGO, 2012).

## **Funcionamento do Java**

*Java* é uma linguagem compilada e ao mesmo tempo interpretada. Diferente das linguagens C e C++ onde suas instruções são transformadas em instruções pré-definidas e enviadas diretamente ao processo, o *Java* transforma suas orientações em *bytecodes* que são lidos por uma VM, isso faz com que todas a sua codificação seja universal (sem a necessidade de se adaptar ao processador utilizado) (NIEMEYER e KNUDSEN, 2005; ASCENCIO e VENERUCHI, 2007).

Pode-se ver na Figura 1 como o código do *Java* é transmitido para todos os dispositivos sem mudanças graças a utilização da *Virtual Machine*.

Figura 1 - Distribuição de um código criado em Java para diversos dispositivos

Fonte: *Learning Java, 2005.*

Na Figura 1 foi observado como o código fonte em Java é transformado em bytecode que vai ser utilizado pela *Virtual Machine*,podendo ser executado em ambientes diferentes.

## Kotlin

### História do *Kotlin*.

*Kotlin* é uma linguagem de programação Orientada a Objetos / Funcional inspirada em *Swift*, *C#*, *Scala*, *Groovy* que foi criada em 2011 pela empresa desenvolvedora de *Softwares JetBrains* na Rússia. O líder do projeto decidiu nomear a linguagem em homenagem a Ilha de *Kotlin*, sediada próximo as instalações da *JetBrains* em São Petesburgo. O objetivo do desenvolvimento do *Kotlin* era facilitar a criação das *IDE’s* comercializadas e/ou utilizadas pela desenvolvedora, deixando o código menor e mais intuitivo para os programadores (SAMUEL e BOCUTIU, 2017; GUANARABRA, 2018).

A linguagem *Kotlin* utiliza a *JVM*, portanto, ela também consegue ler os *Bytecodes* do *Java*, sendo possível a utilização de funções e ferramentas da outra linguagem em seu código com poucas alterações (MOSKALA e WOJDA, 2017).

O *Kotlin* foi sendo desenvolvido lentamente até o ano de 2016, quando foi lançado o *Kotlin* 1.0, sendo a primeira versão estável da linguagem. As ferramentas da linguagem já podiam ser utilizadas nas *IDE’s* da *JetBrains* como o *Android Studio* a partir de *plugins* (RESENDE, 2018).

Em 2017, devido a processos e complicações na relação da *Google* com o *Java*, a *Google* escolheu o *Kotlin* como a segunda linguagem padrão utilizada na programação de aplicativos para o *Android*. A escolha se deveu a vários fatores, como mencionado anteriormente, a relação entre a desenvolvedora do *Android* com a *Oracle* estava avariada devido ao processo por direitos de uso de *API’s* da *JVM* na máquina virtual *Dalvik*. A busca silenciosa da *Google* por uma linguagem que transmitisse os mesmos *Bytecodes* da linguagem já utilizada levou a escolha do *Kotlin*, uma ferramenta com todos os requisitos desejados na programação *Android* e sem ligação nenhuma com a *Oracle*, além do fato da *JetBrains* já ser uma grande parceira dos sistemas *mobiles* do *Android* (SAMUEL e BOCUTIU, 2017; GUANARABRA, 2018).

### Funcionamento do *Kotlin*

De acordo com Marcin Moskala, *Kotlin* é uma linguagem descrita como “segura, expressiva, concisa, versátil e de fácil uso”, podemos confirmar essa descrição ao olharmos as características abaixo:

Uma linguagem que oferece segurança (*safety*), pois contém ferramentas imutáveis (uma vez definida o tipo de um objeto, ele não pode ser alterado durante a utilização do *Software*) e com nulidade (função que faz com que variáveis possam carregar um valor nulo). É uma linguagem estática, ou seja, previne erros no *runtime* da aplicação e acusa as divergências quando elas são compiladas (MOSKALA e WOJDA, 2017).

* Uma linguagem facilmente *debuggada*, onde erros podem ser encontradas na durante o desenvolvimento, sem *crashar* a aplicação quando executada (MOSKALA e WOJDA, 2017; ISAKOVA e JEMEROV, 2017).
* Uma linguagem concisa que prioriza a simplicidade na programação e o melhor entendimento do código por parte de seus desenvolvedores, como prova dessa afirmação, comparando o *Kotlin* 1.0 com o *Java* 7 (sua linguagem de base), a maior parte das expressões do *Java* foi eliminada para aumentar a intuitividade do *Kotlin* (MOSKALA e WOJDA, 2017).
* Uma linguagem versátil que pode ser utilizada em diversas plataformas e programas, alguns exemplos são: aplicações para sistemas *mobile*, aplicações para sistemas *Desktop*, aplicações em servidores, *backend* e *frontend* (MOSKALA e WOJDA, 2017).

## Dart

A linguagem *Dart* foi criada em 2011, porém, somente em 2012, durante o *Google I/O,* a *Google* a anunciou oficialmente sua nova linguagem de programação, voltada para a *Web* e para aplicativos *mobiles*. A proposta do *Dart* era ser uma linguagem capaz de substituir e superar o *JavaScript*, pois, a *Google* não estava contente com a velocidade de evolução da linguagem e a sua má performance em projetos de grande escala (MIRGHANI, 2020; BRACHA, 2015; BITECOURT, 2022).

Durante a criação do *Dart*, os desenvolvedores focaram em manter a dinâmica do *JavaScript*, criando uma linguagem no estilo C, com escalabilidade e sendo orientada a objetos. Já no final de 2013, após alguns ajustes e correções da versão beta da linguagem, foi lançada para o público a versão 1.0 do *Dart* (MIRGHANI, 2020; BRACHA, 2015; BITECOURT, 2022).

Em 2014, a *Google* conseguiu concluir um de seus objetivos, segundo Julio Bitecourt, o *Dart* 1.1 conseguiu uma melhora de performance em suas aplicações 25% em relação ao *JavaScript*.

### Comparação entre *Java* e *Dart*

Como o *Java* e o *Dart* são as linguagens de origem das nossas *Kotlin* e do *Fluttler*, respectivamente, vamos fazer uma pequena comparação entre elas:

* Desenvolvedora: *Java* foi criado pela *Sun Microsystems* (agora *Oracle*), enquanto *Dart* foi desenvolvido pelo *Google*.
* Visão da linguagem: *Java* é uma linguagem interativa atualizada, já *Dart* é visto como uma linguagem para o futuro.
* Sintaxe: *Java* e *Dart* possuem estruturas similares. *Java* utiliza palavras-chave, enquanto *Dart* não, dependendo de letras especiais e comandos de bibliotecas.
* Semântica: *Java* possui regras rígidas como o *else* emparelhando com o *if* mais próximo. *Dart* usa pacotes de semântica como o *Flutter Semantics* *Package*.
* Portabilidade: *Java* depende da JVM e compiladores JIT, enquanto *Dart* funciona em múltiplas plataformas.
* Resultados falsos: *Java* lida com múltiplos tipos falsos (*null*, *false*, 0), *Dart* considera apenas *false*.
* Comentários: Ambos utilizam “ // ” para comentários.
* Execução e Código: *Java* usa *UNICODE* e *ASCII*; *Dart* utiliza *UTF*-32.
* Tratamento de Exceções: *Java* possui modelo tradicional de exceções e eventos, *Dart* usa *Future*, *Stream*, e uma abordagem moderna de exceções.
* Tipos de Dados: *Java* define tipos primitivos e complexos, suporta *arrays* e *strings*. *Dart* já traz tipos embutidos como *numbers, strings, booleans, lists,* *maps*, entre outros.
* Expressões e Atribuições: Em *Dart*, muitos operadores podem ser sobrescritos.
* Controle de Fluxo: Ambos possuem estruturas como *if-else*, *for*, *while*, mas *Dart* adiciona controle por *assert*, além de switch e case.
* Bibliotecas: *Java* e *Dart* usam o conceito de *import* para bibliotecas, mas *Dart* enfatiza estruturas como *Function*, *Callable* classes e *APIs* públicas.
* Escopo Léxico e Subprogramas: Em ambas as linguagens, os parâmetros são passados por valor e referência. *Dart* facilita a criação de funções anônimas e *closures*.
* Exemplo de Estrutura:
  + Em *Java*: Definição de classe *StackClass* com métodos como *push*, *pop* e *empty*.
  + Em *Dart*: Definição de funções simples (*printInteger*) e execução do *app* através da função *main*.

### *Dart SDK*

O *Dart SDK* possui todas as ferramentas e bibliotecas necessárias para se programar com o Dart. O *SDK* se divide em dois diretórios, um deles sendo o “*lib*” onde ficam algumas bibliotecas que fornecessem o essencial para o programador no desenvolvimento do projeto e o outro diretório sendo o “*bin*” onde ficam as linhas de comandos necessárias para se programar com a linguagem (LOUGHEED, 2025).

As principais bibliotecas incluídas no *Dart SDK* são:

* ***dart:core*:** Esta biblioteca possui os tipos fundamentais da linguagem, as coleções que o programador usa para agrupar dados e outras funcionalidades básicas. Esta biblioteca é importada automaticamente para qualquer projeto feito com *Dart* (LOUGHEED, 2025);
* ***dart:async*:** Esta biblioteca traz as ferramentas necessárias para se trabalhar com operações que levam em consideração o “tempo” para serem concluídas, como as classes *Future* e *Stream* (LOUGHEED, 2025);
* ***dart:math*:** Esta biblioteca adiciona constantes matemáticas como π e funções para cálculos, além de gerar números aleatórios (LOUGHEED, 2025);
* ***dart:convert*:** Esta biblioteca ajuda a transformar dados de um formato para outro, como converter informações para *JSON* ou trabalhar com textos em *UTF-8* (LOUGHEED, 2025).
* ***dart:io*:** Esta biblioteca é utilizada em programas que rodam no ambiente *Dart VM* (como aplicativos feitos utilizando *Flutter*, servidores e scripts de linha de comando) e precisam interagir com arquivos, rede, etc (LOUGHEED, 2025).

## Flutter

*Flutter* é um *framework* *open source* criado a partir da linguagem *Dart* em 2015 pela *Google*, com a intenção de ser uma ponte para a programação de alta performance *Mobile*, onde os programas iriam ser executados em múltiplas plataformas com apenas um código base, poupando tempo o dinheiro das desenvolvedoras.

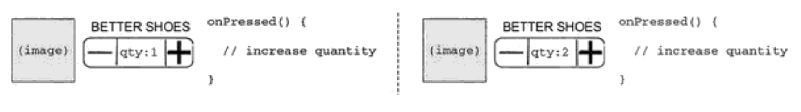
A grande influência para a criação do *framework* utilizando o *Dart* como linguagem padrão, foram os seus múltiplos SDK’s, que permitia que o código fosse compilado usando o código nativo de cada plataforma (KAUSHIK, 2023; TASHILDAR, SHAH, GALA, GIRI e CHAVHAN; 2020; WINDMILL, 2020).

Podemos resumo que o *Flutter* é um SDK completo para a criação de aplicações. Ele prove todas as necessidades de um desenvolvedor como um mecanismo de renderização, componentes para UI, outros *framework*s para teste, *router* e diversas outras ferramentas (WINDMILL, 2020).

### **Como Flutter funciona?**

Como nas linguagens onde tudo são objetos, no *Flutter* tudo são *Widgets* e *Widgets* dentro de *Widgets*. As *Widgets* é tudo o que irá compor a interface do programa para o usuário. Uma *Widget* possui um estado, por exemplo, o número de itens em um carrinho pode ser um estado. Na Figura 2, o campo “qty” é o estado da widget e ao alterarmos ele de 1 para 2 o estado é modificado e toda cadeia de *widgets* é lida novamente. É importante ressaltar que as *widgets* e sua atualização de estado são os pilares para os desenvolvedores que utilizam o *Flutter* (KAUSHIK, 2023; WINDMILL, 2020; COIMBRA DE ARÁUJO, 2021).

Figura – Atualização do Widget no Flutter



Fonte: WINDMILL, 2020.

O *Flutter* possui um catálogo com 14 tipos de Widgets, onde elas se dividem em dois grupos:

* Cupertino: Sendo as widgets utilizadas no desenvolvimento de aplicativos para IOS;
* Componentes de matérias: São widgets que seguem o padrão de trabalho da *Google*.

### Desenvolvimento de aplicações multiplataformas

O desenvolvimento multiplataforma se refere a criação de uma única base de código capaz de ser executada em diversos sistemas operacionais, abrangendo *Android*, iOS e plataformas web ou desktop.

A principal vantagem dessa abordagem reside na otimização de tempo e recursos. Ao compartilhar uma parcela significativa do código entre as plataformas, os custos de desenvolvimento e manutenção tendem a ser menores, conferindo maior agilidade e economicidade ao processo. Consequentemente, experiência do usuário tende a apresentar maior uniformidade em diferentes dispositivos (AWS, 2025).

### **Vantagens da programação com *Flutter* em projetos multiplataformas**

* Desempenho Próximo ao Nativo: *Flutter* compila código *Dart* para código de máquina nativo, resultando em alta performance e execução eficiente nos dispositivos (AWS, 2025).
* Renderização Consistente com Skia: A utilização da biblioteca gráfica Skia garante renderização rápida, consistente e personalizável da interface do usuário em diversas plataformas (AWS, 2025).
* Ferramentas de Desenvolvimento Eficazes: Recursos como "*Hot Reload*" e o "*Widget Inspector*" otimizam o desenvolvimento, permitindo visualização instantânea de alterações e inspeção facilitada de *layout* (AWS, 2025).

## Material Design

O *Material Design*, anunciado em 2014, pela *Google*, foi criado para unificar estilos e padrões das aplicações *Google* e do sistema *Android*. O *Google* iniciou o projeto por meio de uma atualização de design de ícones e logos dentro dos produtos mais utilizados, seguindo padrões que incluem uma hierarquia tipográfica e consistência em botões e iconografia. Após isso, foi realizado uma integração dessa nova atualização com elementos de *UI*, que seria utilizada tanto em dispositivos *Desktop* quanto *Mobile*. Como o projeto foi lançado com uma licença *open source*, foi possível disponibilizar o acesso para qualquer desenvolvedor ou empresa que deseja seguir e implementar esse *Design System* (GIBSON, 2024).

O *Material Design* é caracterizado como um *Design System* (DS), que consiste em um conjunto de modelos, documentações, elementos e boas práticas para o desenvolvimento de interfaces digitais consistentes. Os modelos são caracterizados como elementos repetitivos, que são utilizados para criar uma interface, incluindo, mas não se limitando a interações, fluxos, botões, paleta de cores, ícones, sistemas tipográficos e campos de texto. As práticas dizem como deve ser utilizado e de qual forma, como por exemplo para criar ou utilizar os modelos (KHOLMATOVA, 2017).

A partir de seu lançamento inicial, em 2014, o *Material Design* evoluiu em 3 eras: Material 1 (2014 - 2017), Material 2 (2018 - 2020) e Material 3 (2021 - 2024).

Na primeira era, o *Material Design* se baseou em princípios modernistas, minimalistas e na metáfora do papel *“paper”* como princípio estruturante. Protótipos físicos permitiram explorar hierarquias de camadas, efeitos de sombra e de elevação. A ênfase no minimalismo modernista e em grades rígidas de tipografia conferiu coesão e clareza às interfaces, e o mantra *“motion carries meaning”* estabeleceu parâmetros físicos para animações, garantindo que comportamentos e transições respeitassem as expectativas perceptivas dos usuários (ROBERTSON, 2024).

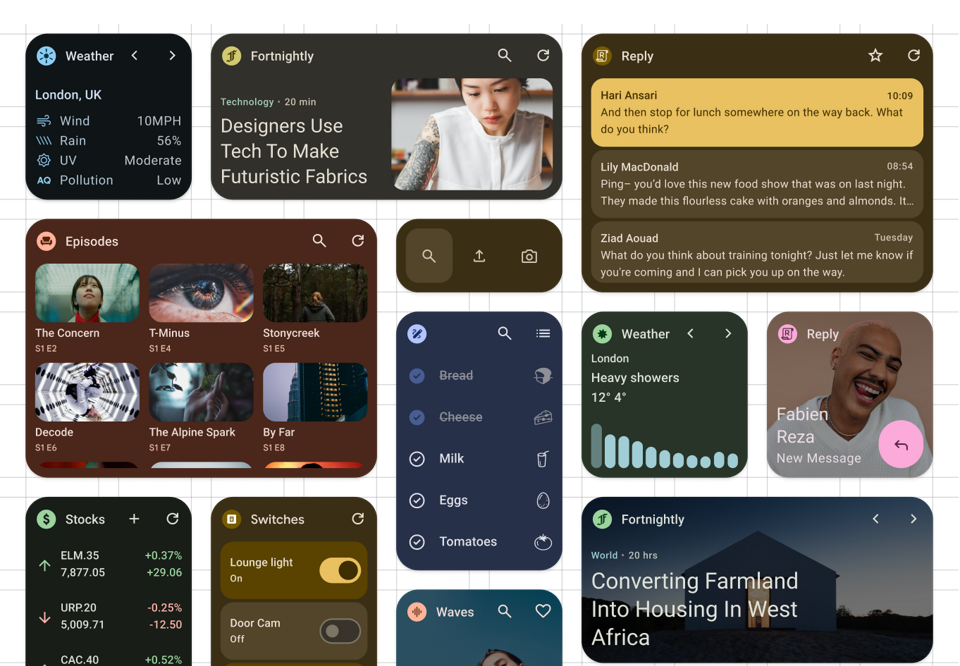
A segunda era concentrou-se na ampliação e consolidação do catálogo de componentes e na integração efetiva de habilidades entre designers e engenheiros. Foram implementados dezenas de componentes em código, acompanhados de um sistema de customização de cores, formas e tipografia.

Isso viabilizou temas personalizados, permitindo que desenvolvedores implementassem sua própria marca e temas. Paralelamente, foram realizados processos rigorosos de testes de acessibilidade, incluindo navegação por teclado e avaliação de contraste de cores, que asseguraram princípios de usabilidade e inclusão digital (JEPSEN, 2024).

A terceira e última era, denominada *Material You*, aprofundou conceitos de personalização dinâmica e de expressão do sistema. A extração de paletas a partir das imagens de fundo do usuário, a introdução de formas orgânicas e tipografia variável promoveram interfaces sensíveis ao contexto e às preferências individuais.

Essa fase também reforçou a cocriação com a comunidade de desenvolvedores e designers, reconhecendo a “vulnerabilidade” como vetor de inovação, algo inevitável e absolutamente necessário para evoluir, permitindo que desenvolvedores experimentem livremente, e que com isso, será possível gerar dados de uso valiosos para a evolução contínua do sistema (DOLBERG, 2024).  
 Na figura 3, é mostrado um exemplo de componentes em estilo de widgets utilizando o Design System do Material Design em sua terceira versão.

Figura - *Material 3 Widgets*

 Fonte: *ANDROID* *DEVELOPERS*, 2024.

Na Figura 3 a aplicação de conceitos‑chave do *Material Design*: hierarquia tipográfica, uso consistente de ícones e cores, e sombreamento mantendo uma hierarquia de camadas.

### *Jetpack Compose*

O *Jetpack Compose* é um *kit* de ferramentas utilizado para desenvolver aplicativos para o *Android*. Ele funciona de forma declarativa, sem a necessidade de controlar e modificar de forma imperativa, as telas de um aplicativo.

Na construção imperativa, a hierarquia de componentes utilizados em uma tela é representada em uma arvore de *widgets.* Conforme eventos, ou interações dos usuários, é necessário atualizar essa arvore com as novas mudanças de interface, por exemplo, percorrendo essa arvore e alterando o valor manualmente (*ANDROID* *DEVELOPERS*, 2025).

Porém, essa abordagem aumenta a probabilidade de serem adicionados erros na aplicação, em muitos casos, esquecendo de atualizar um componente em um evento, ou alterar um valor de um componente que já foi removido da arvore de componentes (RICHARDSON, 2020; *ANDROID* *DEVELOPERS*, 2025).

Em um contexto imperativo, temos uma *ViewModel*, e um *XML* para *layout* das telas. A responsabilidade da *ViewModel* é prover dados para o *layout*, gerando um acoplamento, tendo a necessidade de a *ViewModel* conhecer a estrutura e o conteúdo desse *layout* *XML*. Porém, conforme a aplicação cresce, isso pode se tornar um problema, pois será necessário certificar que as dependências estejam corretas e atualizadas, não somente em seu próprio contexto, mas sim durante o ciclo de vida da aplicação, pois caso seja removido da arvore de componentes, esse acoplamento pode quebrar gerando erros de *NullPointerException* (RICHARDSON, 2020; *ANDROID* *DEVELOPERS*, 2025).

Então, com o *Jetpack Compose*, essa complexidade vai ser reduzida, já que a criação da interface será feita de forma declarativa, onde todo o conteúdo visual será fornecido e atualizado, sem a necessidade de atualizações manuais, por exemplo, por meio de *setters*, no *layout* da tela. Isso elimina o acoplamento com *layout* XML e diminui a probabilidade de erros. A *ViewModel* continua sendo a responsável por controlar o estado na tela, porém agora, desacoplada da representação visual, o que torna o código mais limpo e resiliente a atualizações durante o ciclo de vida da aplicação (RICHARDSON, 2020; *ANDROID* *DEVELOPERS*, 2025).

## Android

O *Android* é um sistema operacional utilizado em dispositivos móveis *Touchscreen*, que teve a sua criação no ano 2003, em Palo Alto, California, sendo um de seus fundadores, Andy Rubin, que tinha como objetivo desenvolver dispositivos móveis inteligentes e com customizações definidas conforme escolhas do usuário (KRAJCI; CUMMINGS, 2013).

No começo, o foco não era em dispositivos móveis, mas sim em câmeras digitais, porém, após analisar esse mercado e notar uma queda de vendas, por motivos de fabricantes estarem inserindo as câmeras nos dispositivos móveis, foi realizado uma nova mudança para focar no desenvolvimento em celulares (HAASE, 2021).

Em 2005, a *startup* estava com falta de recursos para realizar o desenvolvimento do SO, então, foram apresentadas ideias e possíveis aplicações, então, a *Google*, decidindo entrar no mercado de dispositivos móveis, realizou a compra do *Android*, iniciando uma nova divisão de desenvolvimento de voltado a celulares (HAASE, 2021).

Nos anos de 2007 e 2008 foram disponibilizados novas atualização sobre o futuro do *Android*, sobre disponibilização, em seu núcleo, como código *open-source*, por meio do *Android Open Source Project* (AOSP). Com o lançamento do *Android*, houve muitos benefícios para se desenvolver para o SO, na qual, por exemplo, era possível disponibilizar os apps em múltiplos dispositivos, tendo pouca ou nenhuma alteração em seus códigos. Isso permitia que os desenvolvedores pudessem focar em construir aplicativos ricos e impressionantes como é feito atualmente (KRAJCI; CUMMINGS, 2013).

### *Android* SDK

Para o sistema operacional *Android*, o desenvolvimento ocorre principalmente por meio de duas linguagens: *Kotlin* e *Java*. Para desenvolver um aplicativo *Android*, é necessário instalar a IDE *Android* *Studio*, uma versão do JDK (Geralmente já instalada com o *Android* *Studio*) e o SDK. O *Android* SDK consiste em um pacote com diversas ferramentas essenciais para o desenvolvimento, incluindo ferramentas de build para compilar aplicativos, ferramentas de plataforma para interagir com dispositivos *Android*, emuladores, o *API* *Level* e bibliotecas/*API*s que fornecem funcionalidades específicas para o desenvolvimento (*ANDROID* *DEVELOPERS*, 2024).

O *API* *Level* é um número que indica a revisão da *API* de *framework*, que pode mudar a cada lançamento do *Android*.

Para cada lançamento, são oferecidos pacotes e classes principais utilizadas para o desenvolvimento de apps, elementos e atributos XML, que são utilizados para *manifests* ou recursos e, por fim, um conjunto de *intents* e permissões configuráveis, que podem ser solicitadas por aplicativos, bem como a aplicação das permissões incluídas no sistema.

A cada nova versão do *Android*, a *API* é atualizada incrementalmente, com novas funcionalidades adicionadas e outras marcadas como descontinuadas, mas geralmente não são removidas, permitindo o uso em apps existentes. Raramente algum recurso é removido ou alterado drasticamente, apenas quando necessário por motivos de segurança ou robustez (*ANDROID* *DEVELOPERS*, 2025).

Uma atualização de *API* *level* mais importante é a do nível 21 (Lollipop), adicionada com o lançamento do *Android* 5.0, que trouxe inúmeras melhorias em gráficos, notificações, a adição da primeira versão do *Material Design* e performance substituindo o Dalvik para o ART, trazendo uma compilação AOT e JIT, na qual o código nativo do aplicativo será gerado no momento da instalação em vez de em tempo de execução. Também, com a implementação do ART foi melhorado a performance na coleta de Lixo reduzindo o número de pausas e duração dos eventos de GC (*ANDROID* *DEVELOPERS*, 2024).

Por fim, outra atualização de *API* *Level*, também importante, é a de nível 23 (*Marshmallow*) que implementou um novo sistema de solicitação de permissões “perigosas” e normais, que ao invés de, ao instalar um aplicativo, ser concedido em tempo de instalação, o acesso a todas as permissões, após essa separação, todas as permissões “perigosas” agora serão solicitadas ao usuário em tempo de execução. Essa abordagem reforça um controle e transparência ao usuário, permitindo que o aplicativo explique para o que será utilizado o acesso ao recurso e deixando que o usuário recuse ou aceite a permissão (*ANDROID* *DEVELOPERS*, 2024).

## Android Emulator

O *Android Emulator* é um emulador que é utilizado para simular um dispositivo *Android* em um computador.

Nesse emulador é possível criar um AVD (*Android Virtual Device)*, que representa configurações e características de um dispositivo, como um celular ou *tablet*, sendo elas, configurações de *hardware* como quantidade de *RAM*, resolução de tela, tipo de dispositivo, quantidade de armazenamento, configurações de rede etc.

No AVD é possível também selecionar diferentes tipos de *API*s (Versões do *Android*), sem precisar possuir um dispositivo físico com uma versão específica (*ANDROID*, 2025).

## Git

O *Git* é um sistema de controle de versão distribuído criado por Linus Torvalds no ano de 2005. Em 2005, a relação entre a comunidade responsável pelo desenvolvimento do núcleo do *Linux* e a empresa *BitMover* *Inc*, criadora do *BitKeeper* se rompeu, em razão da escolha da *BitMover Inc*., em tornar sua ferramenta em um produto pago. Esse episódio serviu de alerta para os desenvolvedores do *Linux*, especialmente para Linus Torvalds, que decidiu desenvolver uma nova ferramenta inspirada nas aprendizagens com o uso do *BitKeeper* (CHACON, STRAUB, 2014).

Um dos principais motivos para a criação do *Git* foi para lidar com as dificuldades do desenvolvimento do *kernel* do *Linux*, necessidades que surgiram por conta da exigência que o projeto de um sistema de controle de versão que fosse distribuído, que permitisse que diferentes desenvolvedores em múltiplas rotas de trabalho pudessem trabalhar com o mesmo código e que garantisse a integridade dos dados. Entre os objetivos desse novo sistema estavam a alta velocidade, simplicidade do projeto, forte suporte ao desenvolvimento não-linear com milhares de ramificações paralelas, uma arquitetura completamente distribuída e a capacidade de lidar de forma eficiente com projetos de grande porte, como o próprio núcleo do Linux, tanto em termos de desempenho quanto de volume de dados (CHACON, STRAUB, 2014).

A principal diferença entre o *Git* e qualquer outro *VCS* é a forma como é tratada os dados. De forma abstrata pode-se dizer que a maioria dos sistemas armazenam informação como uma lista de mudanças nos arquivos.

Estes sistemas tratam a informação como um conjunto de arquivos e as mudanças feitas em cada arquivo ao longo do tempo.

Em vez disso, o *Git* trata os dados mais como um conjunto de imagens de um sistema de arquivos em miniatura. Toda vez que que é feito um *commit*, ou um salvamento do estado de um projeto no *Git*, ele resumidamente cria uma foto de todos os arquivos e armazena uma referência para esse conjunto de arquivos.

Por questões de eficiência, se os arquivos não foram alterados, o *Git* não armazena o arquivo novamente, apenas um link para o arquivo idêntico anterior já armazenado. O Git trata os dados como um fluxo do estado dos arquivos (CHACON, STRAUB, 2014).

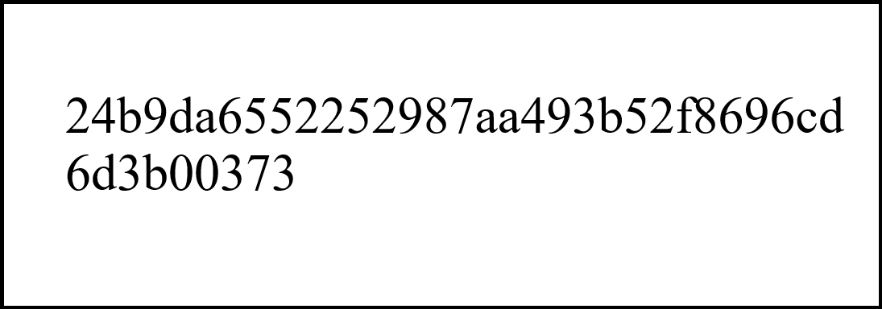
Grande parte das operações no *Git* só precisa de arquivos e recursos locais para operar, geralmente nenhuma informação é necessária de outro computador da rede. Como se tem toda a história do projeto no disco local, a maioria das operações parecem quase instantâneas. Para pesquisar o histórico do projeto, o *Git* não precisa ele lê diretamente do banco de dados. Caso seja necessário ver as alterações introduzidas entre a versão atual de um arquivo e o arquivo de um mês atrás, o *Git* pode procurar o arquivo de um mês atrás e fazer um cálculo de diferença local (CHACON, STRAUB, 2014).

Tudo passa por uma soma de verificações *checksum* antes de ser armazenado e é referenciado por esse *checksum*. Isso demonstra que é impossível mudar o conteúdo de qualquer arquivo ou pasta sem que *Git* saiba. Esta funcionalidade está incorporada no *Git* nos níveis mais baixos e é parte integrante de sua filosofia. Não se perderá informação durante a transferência e não será recebido um arquivo corrompido sem que o Git seja capaz de detectar.

O mecanismo utilizado para esta soma de verificação é chamado um *hash* *SHA-1*. Uma sequência de 40 caracteres composta de caracteres hexadecimais (0-9 e-f), calculada com base no conteúdo de uma estrutura de arquivo ou diretório no Git.

Na Figura 4 é exemplificado um hash SHA-1, que é utilizado para realizar uma verificação antes do conteúdo alterado ser armazenado (CHACON, STRAUB, 2014).

Figura – Exemplificação de um hash SHA-1



Fonte: Os próprios autores (2025).

O *Git* armazena tudo em seu banco de dados não pelo nome do arquivo, mas pelo valor de hash do seu conteúdo (CHACON, STRAUB, 2014).

### *GitHub*

O *GitHub* é uma plataforma baseada no conceito de nuvem onde é possível armazenar, compartilhar e trabalhar com outros desenvolvedores para desenvolver uma códigos. O conceito do *GitHub* é de trazer o controle de versão local do *Git*, para um repositório online onde as alterações feitas localmente podem ser salvas na nuvem e, também permitir os usuários trabalharem em equipe para o desenvolvimento do projeto. As alterações realizadas localmente podem ser sincronizadas com o repositório online para que todos que estejam trabalhando com os arquivos possam ter as versões mais atualizadas, e haja um controle sobre as mudanças do projeto (GITHUB, 2024; DOHMKE, 2023).

Em 2022 havia mais de 413 milhões de contribuições de código aberto na plataforma do GitHub, que atualmente conta com cerca de 100 milhões de desenvolvedores em todo o mundo, e tem mais de 90% das empresas da Fortune 100 utilizando a plataforma (GITHUB DOCS, 2025).

# REFERÊNCIAS

*ANDROID* *DEVELOPERS*, WIDGETS on *Android* | UI Design. 2024. Disponível em: http://developer.*android*.com/design/ui/widget. Acesso em: 3 maio 2025.

*ANDROID* *DEVELOPERS*. *ANDROID*. 2024. Disponível em: https://developer.*android*.com/. Acesso em: 11 maio 2025.

*ANDROID*, Create and manage virtual devices. 16 abr. 2025. Disponível em: https://developer.*android*.com/*studio*/run/managing-avds. Acesso em: 27 abr. 2025.

*ANDROID*, Run apps on the *Android* *Emulator*. 19 set. 2024. Disponível em: https://developer.*android*.com/*studio*/run/managing-avds. Acesso em: 27 abr. 2025.  
KOTLIN ORG. Kotlin roadmap. [S. l.], 27 fev. 2025. Disponível em: https://kotlinlang.org/docs/roadmap.html. Acesso em: 26 abr. 2025.

ARDITO, Luca; COPPOLA, Riccardo; MALNATI, Giovanni; TORCHIANO, Marco. Effectiveness of Kotlin vs. Java in *android* app development tasks. ScienceDirect. 3 jul. 2020. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0950584920301439. Acesso em: 26 abr. 2025.

AWS. O que é o Flutter?. 2025. Disponível em: https://aws.amazon.com/pt/what-is/flutter/. Acesso em: 11 maio 2025.

BITENCOURT, Julio. O guia de Dart: Fundamentos, prática, conceitos avançados e tudo mais. 2022.

BRACHA, Gilad. The Dart Programming Language. 2015. Disponível em: https://books.google.com.br/books?id=UHAlCwAAQBAJ&printsec=frontcover&hl=pt-BR&source=gbs\_ge\_summary\_r&cad=0#v=onepage&q&f=false. Acesso em: 26 abr. 2025.

CETIC, NIC, CGI. Painel TIC COVID-19. 5 abr. 2022. Disponível em: https://cetic.br/pt/publicacao/painel-tic-covid-19-pesquisa-online-com-usuarios-de-internet-no-brasil-4edicao/. Acesso em: 27 abr. 2025.

CHACON, Scott; STRAUB, Ben. Pro Git book. Apress, 2014.

COIMBRA DE ARAÚJO, Everton. Aprofundando em Flutter: Desenvolva aplicações Dart com Widgets. 2021.

DOHMKE, Thomas. 100 million *developers* and counting. 25 jan. 2023. Disponível em: https://github.blog/news-insights/company-news/100-million-*developers*-and-counting/. Acesso em: 4 maio 2025.

DOLBERG, Noa. Material 3: Curiosity, Flexibility, and Expression. Google Design Library. Disponível em: https://design.google/library/material-design-eras. Acesso em: 3 maio 2025.

ESTUDONAUTA CURSOS, A. Kotlin M00A01 - A História da Linguagem Kotlin. 11 out. 2018. Disponível em: https://www.youtube.com/watch?v=nV1FVTcKKYQ&t. Acesso em: 26 abr. 2025.

FLUTTER. User Showcase. 26 abr. 2025. Disponível em: https://flutter.dev/showcase. Acesso em: 26 abr. 2025.

GIBSON, Zach. When Material Made Its Global Debut. Google Design Library 2024. Disponível em: https://design.google/library/material-design-launch-2014. Acesso em: 3 maio 2025.

GITHUB DOCS. Sobre o GitHub e o Git. Disponível em: https://docs.github.com/pt/get-started/start-your-journey/about-github-and-git. Acesso em: 4 maio 2025.

GITHUB. Octoverse: AI leads Python to top language as the number of global *developers* surges. 22 nov. 2024. Disponível em: https://github.blog/news-insights/octoverse/octoverse-2024/. Acesso em: 4 maio 2025.

GSMA. The Mobile Economy 2024. 30 abr. 2024. Disponível em: https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/wp-content/uploads/2024/02/260224-The-Mobile-Economy-2024.pdf. Acesso em: 27 abr. 2025.

HAASE, Chet. *Androids*: The Team That Built the *Android* Operating System. 2021. ISBN 737354810412 p. ISBN1737354810.

HASSAN, Afaf Mirghani. JAVA and DART programming languages: Conceptual comparison. Indonesian Journal of Electrical Engineering and Computer Science. 2020. Disponível em: https://pdfs.semanticscholar.org/c657/21e98abfafd6470d5ac15a057c2bf3c8eeaf.pdf. Acesso em: 26 abr. 2025.

JEMEROV, Dmitry. Hello World. 19 jul. 2011. Disponível em: https://blog.jetbrains.com/kotlin/2011/07/hello-world-2/. Acesso em: 26 abr. 2025.

JEMEROV, Dmitry; ISAKOVA, Svetlana. Kotlin in Action. 2017. Disponível em: https://books.google.com.br/books?hl=pt-BR&lr=&id=OzkzEAAAQBAJ&oi=fnd&pg=PT18&dq=Kotlin&ots=F2YCoApeUf&sig=JMqmV4\_kaKhuu1PAB7itQI4VH6Y#v=onepage&q=Kotlin&f=false. Acesso em: 26 abr. 2025.

JEPSEN, Lynn. Material 2: A Cohesive Collection of Components. Google Design Library. Disponível em: https://design.google/library/material-design-eras. Acesso em: 3 maio 2025.

KAUSHIK, Vikas. The History Of Flutter: A Comprehensive Overview Of The Development *Framework*. 23 nov. 2023. Disponível em: https://www.techaheadcorp.com/knowledge-center/history-of-flutter/. Acesso em: 26 abr. 2025.

KEMP, Simon. Digital 2025: Brazil. 3 mar. 2025. Disponível em: https://datareportal.com/reports/digital-2025-brazil. Acesso em: 27 abr. 2025.

KHOLMATOVA, Alla. Design Systems: A practical guide for creating design languages for digital products. Smashing Magazine, 2017. 288 p. ISBN 978-3-945749-58-6.

KOTLIN Foundation Annual Report’24. Disponível em: https://kotlinfoundation.org/. Acesso em: 26 abr. 2025.

KRAJCI, Iggy; CUMMINGS, Darren. History and Evolution of the *Android* OS: An Introduction to Optimizing for Intel Architecture: Apress, Berkeley, CA, 2013. ISBN 978-1-4302-6130-8.

LOUGHEED, Parker. Dart‘s core libraries. 2025. Disponível em: https://dart.dev/docs. Acesso em: 11 maio 2025.

MOHTA, Bhavicka; HAYAT, Hashim; HAYAT, Daheem. Flutter's Evolution: Insights from CTO Report 2024. 18 jul. 2024. Disponível em: https://www.walturn.com/insights/flutter-s-evolution-insights-from-cto-report-2024. Acesso em: 26 abr. 2025.

MOSKALA, Marcin; WOJDA, Igor. *Android* Development with Kotlin. 2017.

NIEMEYER, Patrick; KNUDSEN, Jonathan. Learning Java. 2005. Disponível em: https://books.google.com.br/books?id=T9abAgAAQBAJ&printsec=frontcover&hl=pt-BR&source=gbs\_ge\_summary\_r&cad=0#v=onepage&q&f=false. Acesso em: 20 maio 2005.  
Fundamentos da programação de computadores: algoritmos, PASCAL, C/C++ e JAVA. By: Ana Fernanda Gomes Ascencio (autor), Edilene Aparecida Veneruchi de Campos

NUKUMIZU, Shota. The History And Rules of Dart Language. 11 ago. 2022. Disponível em: https://medium.com/@author2000.1225/the-history-and-rules-of-dart-language-f25e09a58530. Acesso em: 26 abr. 2025.

OREILLY, A. The history of Kotlin. Disponível em: https://www.oreilly.com/library/view/kotlin-for-enterprise/9781788997270/ea4ec584-db64-4026-89a8-2086301eb9c5.xhtml. Acesso em: 26 abr. 2025.

RESENDE, Kassiano. Kotlin com *Android*: Crie aplicativos de maneira fácil e divertida. 2018.

RICHARDSON, Leland. Understanding *Jetpack Compose* — part 1 of 2: Better UI building with Compose. 27 ago. 2020. Disponível em: https://medium.com/*androiddevelopers*/understanding-jetpack-compose-part-1-of-2-ca316fe39050. Acesso em: 11 maio 2025.

ROBERTSON, Christian. The Physics and Magic Behind the Screen. Google Design Library. 2024. Disponível em: https://design.google/library/material-design-eras. Acesso em: 3 maio 2025.

STATISTA. Cross-platform mobile *framework*s used by software *developers* worldwide from 2019 to 2023. 26 abr. 2025. Disponível em: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. Acesso em: 26 abr. 2025.

THIAGO, A. Java: história e principais conceitos. 2012. Disponível em: https://www.devmedia.com.br/java-historia-e-principais-conceitos/25178. Acesso em: 26 abr. 2025.

THOMSEN, Michael. Celebrating Flutter’s “Production Era”.17 dez. 2024. Disponível em: https://medium.com/flutter/flutter-in-production-f9418261d8e1. Acesso em: 26 abr.2025.

WINDMILL, Eric. Flutter in Action. 2020. Disponível em: https://books.google.com.br/books?id=EzgzEAAAQBAJ&printsec=frontcover&hl=pt-BR&source=gbs\_ge\_summary\_r&cad=0#v=onepage&q&f=false. Acesso em: 26 abr. 2025.