Giovany Frossard Teixeira
Alextian Bartholomeu Liberato
Renan Osório Rios
Igor Carlos Pulini
Raphael Magalhães Gomes Moreira

Fundamentos de Flutter e Dart para Desenvolvimento de Apps Móveis

Edifes

Fundamentos de Flutter e Dart para Desenvolvimento de Apps Móveis

GIOVANY FROSSARD TEIXEIRA
ALEXTIAN BARTHOLOMEU LIBERATO
RENAN OSÓRIO RIOS
IGOR CARLOS PULINI
RAPHAEL MAGALHÃES GOMES MOREIRA

Fundamentos de Flutter e Dart para Desenvolvimento de Apps Móveis

Edifes

Vitória, 2024

Editora do Instituto Federal de Educação,
Ciência e Tecnologia do Espírito Santo
R. Barão de Mauá, nº 30 – Jucutuquara
29040-689 – Vitória – ES
www.edifes.ifes.edu.br | editora@ifes.edu.br

Produção editorial
Projeto Gráfico: Assessoria de Comunicação Social do Ifes
Revisão de texto: Thaís Rosário da Silveira (Edifes)
Diagramação e epub: Higor Ferraço (Edifes)
Capa: Higor Ferraço (Edifes)

Dados Internacionais de Catalogação na Publicação (CIP)

F981 Fundamentos de Flutter e Dart para desenvolvimento de apps móveis [recurso eletrônico] / Giovany Frossard Teixeira... [et al.]. – Vitória, ES : Edifes, 2024.
1 recurso digital : ePub ; 143 p. ; il. col.

Vários autores.
ISBN: 978-85-8263-847-7 (e-book).

1. Flutter (Framework). 2. Dart (Linguagem de programação de computador). 3. Aplicativos móveis – Desenvolvimento. I. Teixeira, Giovany Frossard. II. Liberato, Alextian Bartholomeu. III. Rios, Renan Osório. IV. Pulini, Igor Carlos. V. Moreira, Raphael Magalhães Gomes. VI. Título.

CDD 22 – 005.1

Bibliotecária responsável: Rossanna dos Santos Santana Rubim – CRB6- ES 403

DOI: 10.36524/9788582638477
Este livro foi avaliado e recomendado para publicação por pareceristas *ad hoc.*

SUMÁRIO

1 INTRODUÇÃO

O cenário atual é marcado pelo aumento crescente na popularidade e na diversificação do uso das tecnologias móveis, como smartphones e tablets. Esses dispositivos transformaram-se em ferramentas ubíquas que moldam a maneira como as interações com o mundo ocorrem, abrangendo desde as formas de comunicação até o acesso a informações e entretenimento.

A compreensão dos conceitos e das ferramentas necessárias para desenvolver aplicativos direcionados a dispositivos móveis amplia não somente as fronteiras do conhecimento, mas também abre oportunidades para se engajar ativamente nessa sociedade em constante evolução.

Dessa forma, neste capítulo, serão apresentados o Flutter e a linguagem Dart, além de compará-los com as principais tecnologias concorrentes para programação em dispositivos móveis. O processo de instalação do Flutter e do Dart, bem como do Android Studio, também será descrito. Por fim, será criada a primeira aplicação em Flutter usando o Android Studio, sendo essa aplicação executada num smartphone virtual com o código gerado sendo discutido de forma introdutória, para maior detalhamento em capítulos posteriores.

O que é o Flutter? E o Dart?

O Flutter, desenvolvido pelo Google, é um conjunto de ferramentas de interface do usuário que permite a criação de aplicativos compilados para dispositivos móveis, web e desktop utilizando um único código-fonte. Embora possa ser necessário algum nível de programação específica para cada plataforma (móvel, web ou desktop), o Flutter facilita o compartilhamento eficiente de grande parte do código entre as diferentes plataformas.

O Flutter é um *framework* de programação de aplicações móveis, web e desktop, com foco inicial e principal no suporte a aplicações móveis. Ele é disponibilizado gratuitamente e possui seu código-fonte aberto. Isso significa que os desenvolvedores têm acesso ao código-fonte do Flutter, permitindo a personalização e contribuição para aprimorar o *framework*. Ele oferece uma funcionalidade chamada *hot reload*, que permite atualizar visualmente a aplicação sem precisar recompilá-la completamente. Durante o *hot reload*, os arquivos de código-fonte atualizados são inseridos na Dart *Virtual Machine* (VM) em tempo de execução. A VM, por sua vez, atualiza as classes com as novas versões de atributos e métodos, enquanto a estrutura do Flutter reconstrói o esquema de widgets. Isso possibilita uma visualização mais rápida dos efeitos das alterações implementadas (GOOGLE, 2023h).

É possível imaginar um aplicativo como um conjunto de peças para montagem de brinquedos, em que cada pequeno widget equivale a uma peça individual. Ao final do processo, essas diversas peças se unem para criar um brinquedo completo e funcional.

No contexto do desenvolvimento com Flutter, essa analogia também se aplica. O Flutter é uma estrutura de criação de aplicativos que se baseia na composição de widgets. Assim como as peças para montagem de brinquedos, os widgets do Flutter são os blocos de construção fundamentais. Eles podem ser combinados e organizados de diversas maneiras para formar a interface do usuário e a funcionalidade do aplicativo como um todo. Cada widget no Flutter possui uma

função específica, desde elementos de layout simples até componentes mais complexos, como botões, listas, campos de texto e muito mais. Portanto, ao projetar um aplicativo no Flutter, os desenvolvedores montam uma variedade de widgets para criar a experiência final que os usuários verão e interagirão.

Flutter utiliza a linguagem de programação Dart. A linguagem Dart é uma linguagem C-style, ou seja, é parecida com C, C++, Java, Javascript ou C#.

A linguagem Dart é tipada, mas a definição de tipos é opcional. Dart possui generics, mixins, funções de alta ordem entre outros recursos. Dart pode ser compilada e interpretada.

Dart é uma linguagem lançada em 2011. Seu objetivo inicial era substituir a linguagem Javascript, que é uma linguagem que continua evoluindo mas tem sua origem no século passado. Entretanto naquele momento não obteve êxito. Com o Flutter, o Dart volta como uma linguagem multiparadigma, apesar de seu cerne ser uma linguagem orientada a objetos. Acredita-se que Dart não evoluiu em 2011 por conta da visão da comunidade de desenvolvimento. A ideia de fragmentar plataformas web não foi bem-vista, além disso, a Google havia deixado de investir em alguns projetos anteriores, o que pode ter gerado certa desconfiança.

1.1 Arquitetura Básica

Conforme é possível verificar na documentação do Flutter[1] sua arquitetura é dividida em 3 (três) camadas: Embedder, Engine e Framework.

A camada Embedder é encarregada de adaptar a aplicação em Flutter para se adequar a cada plataforma em particular. Cada sistema operacional possui seu próprio Embedder exclusivo, elaborado usando linguagens como Java e C++ para Android, ObjectiveC/Objective-C++ para iOS e macOS, e C++ para Windows e Linux.

1 https://docs.flutter.dev/resources/architectural-overview

Por sua vez a camada Engine é responsável por renderizar os widgets na tela, escrito prioritariamente na linguagem C++, fornece a implementação de baixo nível da API central do Flutter, incluindo gráficos (Impeller[2] ou Skia[3]), layout de texto, entrada e saída de arquivos e dados de rede, etc.

Já a camada de Framework é normalmente o meio utilizado pelos desenvolvedores para interagir com o Flutter. É nessa camada que se encontram os widgets e os pacotes de componentes visuais Material (Android) e Cupertino (iOS), além de bibliotecas básicas de animação, desenho e controle de gestos.

O framework Flutter é relativamente pequeno; muitos dos recursos principais que os desenvolvedores podem utilizar são implementados como pacotes, incluindo plugins como câmera, webview, http, etc.

1.2 Instalação

A instalação do Flutter é relativamente simples e está disponível no endereço: **https://flutter.dev/docs/get-started/install.**

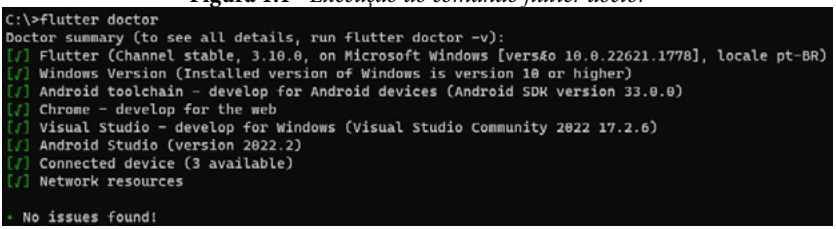
É possível instalar o Flutter em Windows, macOS, Linux ou ChromeOS. Basta seguir o passo a passo do link apresentado acima que será possível instalar e configurar o Flutter.

Uma das vantagens na instalação do Flutter é a utilização do Flutter Doctor. Essa ferramenta faz uma varredura na plataforma de desenvolvimento verificando se há alguma pendência para o desenvolvimento de aplicações em Flutter. Para acionar o Flutter Doctor basta ir ao terminal e executar o comando **flutter doctor**. A Figura 1.1 mostra a execução do comando **flutter doctor** no terminal do Windows 11, notar que o comando foi executado a partir de **C:**, isso não é um problema pois durante o processo de instalação será feito o ajuste de variáveis de sistema, mais especificamente a variável *Path*.

2 https://docs.flutter.dev/perf/impeller

3 https://skia.org/

Figura 1.1 - *Execução do comando flutter doctor*



Fonte: Elaborado pelos autores.

É possível executar o comando **flutter doctor** diretamente no Android Studio. O Android Studio será o IDE (Ambiente de Desenvolvimento Integrado) utilizado nos exemplos apresentados neste livro.

Para instalar o Android Studio basta continuar seguindo os passos disponibilizados em **https://flutter.dev/docs/get-started/install**. Para instalação em macOS será necessário instalar também o Xcode, isso possibilitará testar as aplicações no simulador do iOS dentro do Android Studio.

Importante: Não é possível, atualmente, rodar o simulador do iOS em ambientes Windows ou Linux de forma integrada. Ou seja, no Android é possível criar um smartphone virtual de uma versão específica do Android e rodá-lo para testes, no iOS, em ambientes Windows ou Linux, não há a possibilidade de criação desse dispositivo virtual. Para resolver esse problema existem soluções que utilizam simuladores de iOS de forma remota, mas isso costuma ter custos e a solução mais simples acaba sendo adquirir um computador com macOS instalado e com o Xcode devidamente configurado.

1.3 Tecnologias concorrentes

Neste subtópico, serão apresentados os principais concorrentes do Flutter, destacando suas características, vantagens e desvantagens.

Ionic: é um excelente *framework* para desenvolvimento de aplicações híbridas. Seu foco é "usar a plataforma", ou seja, usar padrões abertos (HTML, CSS, Javascript) da web sempre que possível. O

problema é que, na prática, ele é uma WebView com grande dependência de plugins do Cordova. Como resultado final temos uma aplicação cuja performance é normalmente inferior a tecnologias como Flutter (LIMA, 2020), apesar de que na maioria dos casos essa diferença seja quase imperceptível. Para maiores informações sobre o Ionic basta ir a página do projeto[4].

React Native: desenvolvido pela equipe do Facebook, também é um ótimo *framework*. Aqui tem-se uma melhoria do desempenho, visto que o React Native não usa WebViews. Possui mais tempo de mercado que o Flutter e é bastante utilizado, mas o desempenho é normalmente inferior ao do Flutter por conta da necessidade de uma *bridge* (ponte) entre o aplicativo e os recursos nativos. Para conhecer o React Native de forma mais abrangente o ponto de partida pode ser a página do projeto[5].

Xamarin: foi uma das primeiras tecnologias de desenvolvimento de aplicativos para dispositivos móveis criada para funcionar em várias plataformas. Através dela tornou-se possível desenvolver códigos para Android e iOS com maior aproveitamento de código e parecendo praticamente nativos. Em 2016, a Microsoft compra a empresa Xamarin e essa tecnologia torna-se parte do Visual Studio.

O Xamarin utiliza a linguagem C# e é um *framework* bastante moderno/maduro e com muitos recursos. Possui forte vínculo com o Visual Studio, sendo praticamente a única IDE de desenvolvimento em Xamarin.

O Xamarin.Forms é uma ferramenta do Xamarin que permite o reaproveitamento de componentes visuais para várias arquiteturas (iOS, Android, Windows Phone, etc.).

Os componentes visuais do Flutter são mais ricos em recursos e animações que a maioria dos concorrentes, incluindo Xamarin.

Em termos de desempenho, não é possível afirmar quem é mais rápido. De fato, ambos possuem um desempenho praticamente de

4 https://ionicframework.com/

5 https://reactnative.dev/

código totalmente nativo (Android – Java/Kotlin e iOS – Objective C e Switch).

Atualmente o Xamarin faz parte da plataforma .Net (MICROSOFT, 2023), aumentando assim sua integração com outras ferramentas da Microsoft e possibilitando a evolução dessa tecnologia, com destaque para a .NET MAUI que é a evolução do Xamarin.Forms.

Delphi (Mobile): é mais uma das opções para quem deseja desenvolver *cross-platform* (DUARTE, 2015). O *framework* Firemonkey promete uma experiência de usuário tão boa quanto a de aplicativos nativos.

Normalmente faz essa opção quem já desenvolve em Delphi e deseja navegar pela programação para dispositivos móveis.

A quantidade de material e desenvolvedores é bem menor que a de tecnologias como React Native ou Xamarin.

Enquanto a Microsoft está por trás do Xamarin, o Facebook do React Native e o Google do Flutter, a Embarcadero é responsável pelo Delphi. Sem dúvida nenhuma uma empresa muito menos conhecida que esses três gigantes.

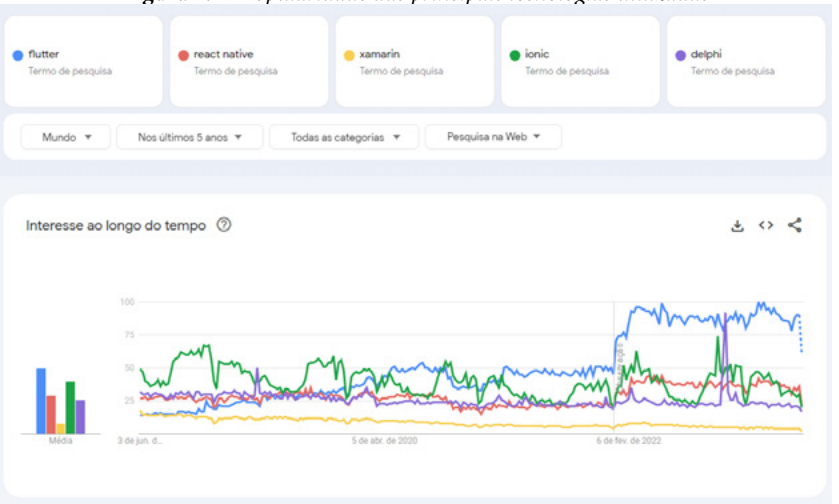
Enfim, se o programador já utiliza Delphi em seu trabalho pode ser interessante usar essa ferramenta para desenvolver mobile, mas é importante ficar atento ao tamanho desse mercado e das possíveis limitações dessa tecnologia.

1.4 Comparando o interesse pelas tecnologias

Uma métrica interessante para sabermos o quão uma tecnologia está avançando é identificar o quão ela tem sido buscada na web, para tanto a Google disponibiliza o serviço Google Trends[6]. Nesse contexto, comparando as buscas por: flutter, react native, xamarin, ionic e delphi nos últimos 5 anos, em 01/06/2023, tem-se o gráfico apresentado na Figura 1.2.

6 https://trends.google.pt/trends/

Figura 1.2 - *Popularidade das principais tecnologias utilizadas*



Fonte: Google Trends (www.google.com/trends)

É importante ressaltar que o Delphi Mobile não foi apresentado na busca pois seus resultados foram praticamente irrelevantes. Por outro lado, pesquisar por Delphi gera resultados não tão verdadeiros pois inclui buscas para desenvolvimento tradicional em Delphi. Mesmo diante desse contexto, é fácil verificar que Flutter, desde março de 2021 (aproximadamente), é a tecnologia mais buscada dentre as selecionadas. Sendo que atualmente, junho de 2023, é a mais buscada com larga margem.

Por fim, os dados da Figura 1.2 servem para demonstrar o quão necessário e diferencial é utilizar a tecnologia Flutter/Dart.

1.5 Visão dos autores

O melhor desempenho possível será sempre o código 100% nativo. Quanto mais próximo disso melhor o desempenho.

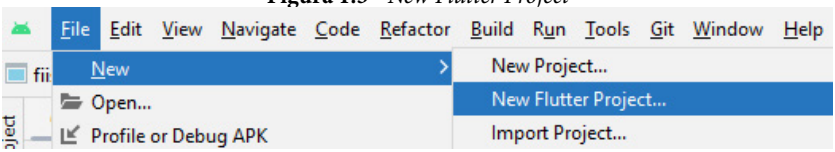
Aprender várias tecnologias ao mesmo tempo é trabalhoso e difícil. Nesse contexto, o desenvolvimento híbrido ou *cross-plataform*[7] possibilitarão maior facilidade de se manter atualizado.

Utilizar ferramentas ligadas a tecnologias web sempre trarão a vantagem de aprender "uma coisa só".

1.6 Primeiro aplicativo no Flutter

Para criar um projeto Flutter, no Android Studio, deve-se ir em: **File→New→New Flutter Project ...**, conforme apresentado na Figura 1.3.

Figura 1.3 - *New Flutter Project*

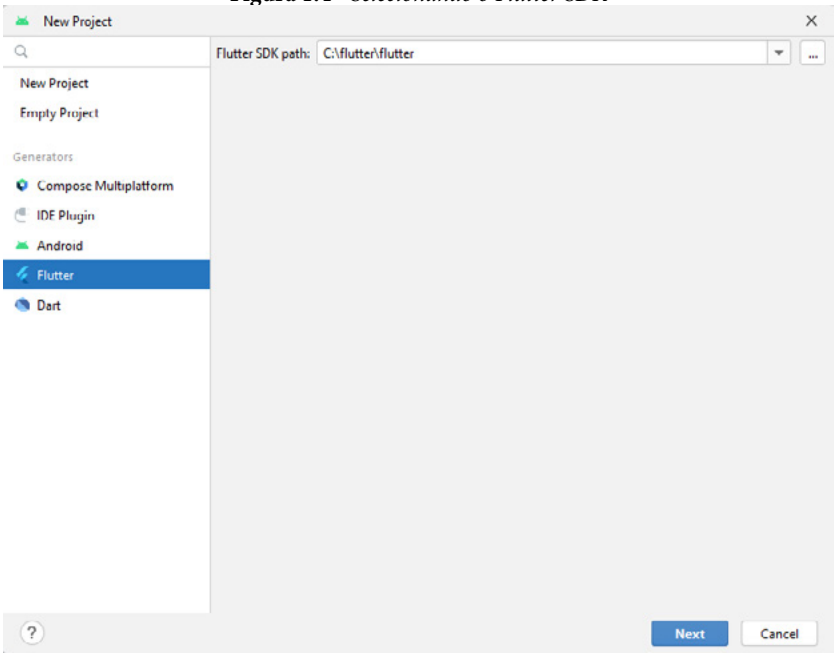


Fonte: Elaborado pelos autores.

Na tela seguinte, Figura 1.4, deve ser selecionado o caminho onde se encontra o Flutter SDK - *Software Development Kit* (onde foi instalado na máquina) e avançar (*Next*).

7 *Cross-platform* refere-se à capacidade de um software, aplicativo ou sistema funcionar em diferentes plataformas ou ambientes sem precisar ser reescrito ou adaptado significativamente para cada uma delas.

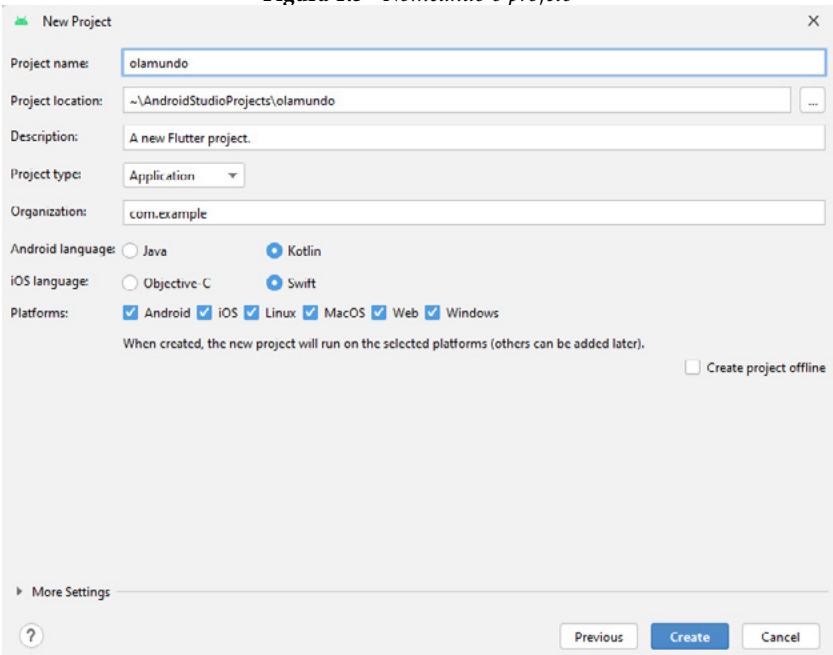
Figura 1.4 - *Selecionando o Flutter SDK*



Fonte: Elaborado pelos autores.

O próximo passo é definir as configurações do projeto, conforme Figura 1.5. Mantenha as opções pré-definidas e defina o nome do projeto. Em seguida, basta clicar em *Create* para que o projeto seja criado.

Figura 1.5 - *Nomeando o projeto*

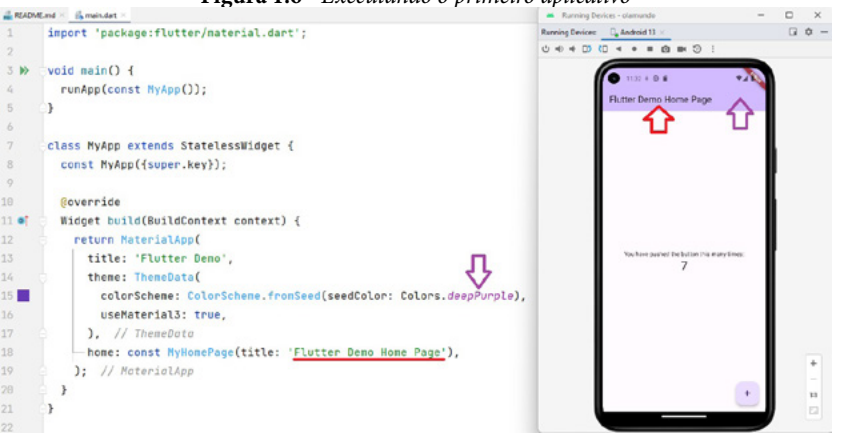


Fonte: Elaborado pelos autores.

Uma vez criado o projeto é possível executá-lo tanto em dispositivos virtuais Android quanto em smartphones ou tablets com esse sistema operacional. Para criar e gerenciar dispositivos virtuais basta seguir o passo a passo descrito por Google Developers (2023a). Já para dispositivos físicos (smartphones ou tablets) é necessário fazer algumas configurações no dispositivo específico antes de utilizá-lo, esse processo de configuração também é descrito por Google Developers (2023b). Nesse livro não será discutido esse processo de configuração pois, além de farto material na web, isso pode variar um pouco de acordo com o fabricante do dispositivo físico utilizado.

Uma vez configurado o dispositivo a ser utilizado, será feita a análise do código fonte gerado pelo Android Studio. Para tanto optou-se por retirar as linhas de comentário geradas automaticamente (isso é fundamental para deixamos o código mais limpo para compreendê-lo de forma mais fácil).

Figura 1.6 - *Executando o primeiro aplicativo*



Fonte: Elaborado pelos autores.

Na Figura 1.6 é possível ver o ponto de partida do aplicativo, a função *main()*. A função *main()* basicamente instancia o objeto da classe **MyApp** que é a classe começa efetivamente a execução do aplicativo. Ela é *StatelessWidget*, ou seja, não mudará de estado (nenhum atributo ou objeto seu alterará seu valor, por exemplo), esse conceito será melhor abordado posteriormente.

A palavra **const** serve para criar "objetos constantes", em outras palavras, ela indica ao compilador que esse objeto nunca será alterado, logo se for criado outro objeto igual, ele pode fazer apenas referência ao criado anteriormente.

O **MaterialApp** é um widget que é criado uma vez por aplicação e serve para setar configurações iniciais e chamar a primeira tela em si.

Nesse exemplo é setado o *colorScheme* para *ColorScheme.fromSeed(seedColor: Colors.deepPurple)*. Experimente mudar de *Colors.deepPurple* para *Colors.green* para ver o efeito na aplicação (não é necessário rodar tudo novamente, basta salvar que o Flutter fará o "*hot reload*").

MyHomePage é a primeira tela criada e recebeu o texto *'Flutter Demo Home Page'* no parâmetro *title*. Experimente mudar o valor de *title* e verifique o efeito na aplicação.

Na Figura 1.7, é possível ver a classe **MyHomePage**. **MyHomePage** é um *StatefulWidget*, isso significa que ele suporta mudança de estado.

Figura 1.7 - *Classe MyHomePage*

```
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});
  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
```

Fonte: Elaborado pelos autores.

Na sequência tem-se a definição do construtor e o atributo *title* é inicializado (*title* é um parâmetro nomeado, e para torná-lo obrigatório utiliza-se a palavra reservada *required*).

Por fim é criado um objeto do tipo **_MyHomePageState**. Esse objeto trabalhará com o **MyHomePage** para tratar a questão da mudança de estado.

Na Figura 1.8 apresenta-se o início do código da classe **_MyHomePageState**. A classe **_MyHomePageState** possui um atributo interno *_counter* que servirá para acumular os cliques no botão flutuante na base da tela. O método *_incrementCounter()* serve para essa finalidade, esse método chama o *setState()* que incrementará o valor do *_counter*. Só é possível utilizar o *setState()* porque a **MyHomePage** é um *StatefulWidget*.

Figura 1.8 - *Classe _MyHomePageState*

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
```

Fonte: Elaborado pelos autores.

Quando o *setState()* é chamado o widget da tela é redesenhado no método *build()* (ver Figura 1.9). É importante reforçar que o atributo interno *_counter* é utilizado no momento do desenho da tela para mostrar o texto da quantidade de cliques.

Figura 1.9 - *Método build*

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: Theme.of(context).colorScheme.inversePrimary,
      title: Text(widget.title),
    ),  // AppBar
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          const Text(
            'You have pushed the button this many times:',
          ),  // Text
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headlineMedium,
          ),  // Text
        ],  // <Widget>[]
      ),  // Column
    ),  // Center
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for build methods.
  );  // Scaffold
}
```

Fonte: Elaborado pelos autores.

O método *build()* é o responsável por desenhar a tela. Logo no início temos um **Scaffold**, esse widget é usado como base para telas no Flutter, ou seja, normalmente uma tela começa de um **Scaffold**.

A **AppBar** é a barra superior que aparece na aplicação. Ela tem como título o valor passado no construtor de **MyHomePage**. Como o método *build()* está em **_MyHomePageState** para acessar o atributo *title* é necessário fazer referência a palavra *widget*.

O *body* é o corpo da tela. Nele é colocado o widget **Center** para centralizar a **Column** (coluna) e dentro da coluna existem dois textos: Um fixo e um que mostra o valor de *_counter*.

Por fim, tem-se o **FloatingActionButton** que quando clicado irá chamar o método *_incrementCounter* (aquele que irá mudar o valor de *_counter*) e acionará novamente o *build()* para reconstruir toda a tela através do acionamento do *setState()*.

É importante notar que quando a aplicação é inicializada e instancia **_MyHomePageState**, o método *build()* é automaticamente chamado (para desenhar o widget pela primeira vez), depois disso o widget só é redesenhado quando *setState()* for acionado e isso só ocorre no *_incrementCounter* em decorrência do clique no botão do tipo **FloatingActionButton**.

É óbvio que diversos conceitos apresentados nesse exemplo carecem de um melhor detalhamento, o objetivo dessa apresentação, nesse momento, foi basicamente introduzir o Flutter/Dart em um exemplo completo mas não muito extenso. No próximo capítulo será apresentada a linguagem Dart, ela é base para compreender os códigos gerados e os que serão futuramente desenvolvidos.

2 A LINGUAGEM DART

Em 2011, o Google lançou o Dart como uma linguagem de scripts com o objetivo de se tornar a principal linguagem para desenvolvimento de scripts em navegadores, substituindo o JavaScript. Dart foi introduzido com o intuito de oferecer uma alternativa poderosa e flexível ao JavaScript, atendendo às demandas crescentes de desenvolvimento web.

Dart recebeu reconhecimento significativo, especialmente devido ao *framework* Flutter, que possibilita a programação de aplicativos nativos para Android e iOS. Além disso, a linguagem também oferece suporte para a criação de aplicações web e desktop.

Assim como C, C++, Java, C# e seguindo o paradigma de orientação a objetos, Dart é uma linguagem fortemente tipada, porém, não é necessário especificar os tipos, uma vez que o Dart consegue inferi-los automaticamente. Além disso, a utilização do caractere underline (_) antes de um atributo ou método o torna privado.

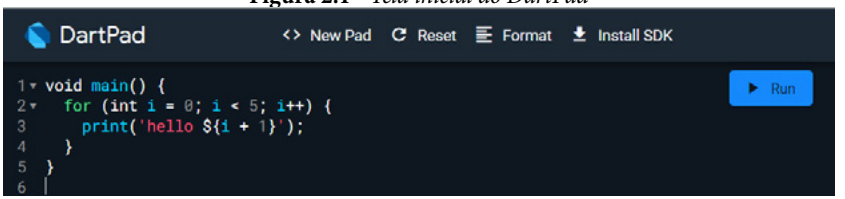
Códigos feitos em Dart podem ser compilados de duas formas: *ahead-of-tim*e (AOT) *e just-in-time* (JIT). No primeiro caso, o código é compilado para uma linguagem nativa (ARM nativo, por exemplo), proporcionando um desempenho semelhante ao de aplicativos nativos. No segundo caso, a compilação ocorre diretamente no dispositivo em que a aplicação está sendo executada, como em um smartphone, permitindo o recurso de *hot reload*.

2.1 DartPad

Para programar em Dart de forma mais fácil e sem a necessidade de um ambiente Flutter configurado é possível usar o site DartPad[8]. O DartPad possui código aberto e pode ser executado em qualquer navegador atual (GOOGLE, 2023c).

A Figura 2.1 apresenta a tela inicial do DartPad. É nesse ambiente que todos os exemplos deste capítulo serão executados.

Figura 2.1 - *Tela inicial do DartPad*



Fonte: Elaborado pelos autores.

2.2 Conceitos iniciais

Em Dart, a programação é estaticamente tipada, o que significa que, em geral, uma vez que atribuímos um valor a uma variável com um tipo específico, não é possível armazenar valores de outros tipos nessa mesma variável (exceto para o tipo *dynamic*). Por exemplo, se uma variável é declarada como uma String, ela não será automaticamente convertida em um número para fins de soma.

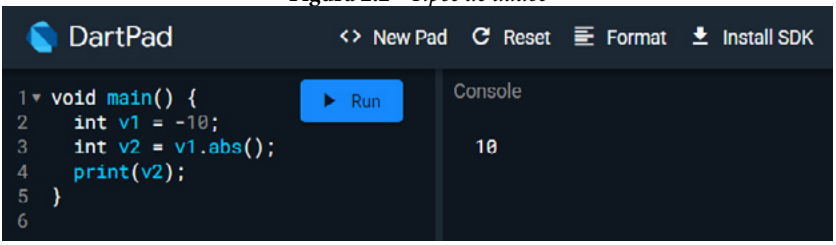
Neste tópico serão discutidas classes que trabalham com números e que em outras linguagens comumente são tratadas como tipos primitivos. Na sequência será tratada a classe String e a classe bool, também será tratada a classe dynamic e as classes Function, List e Map. Por fim serão tratados os argumentos de uma Function e possibilidades relacionadas a esse conceito.

2.2.1 Classes que trabalham com números

8 https://dartpad.dev/

Tipos primitivos, como existem em linguagens como C ou Java não existem em Dart. Em Dart "Tudo é objeto". A Figura 2.2 ilustra uma variável do tipo **int** chamando o método **abs()**.

Figura 2.2 - *Tipos de dados*



Fonte: Elaborado pelos autores.

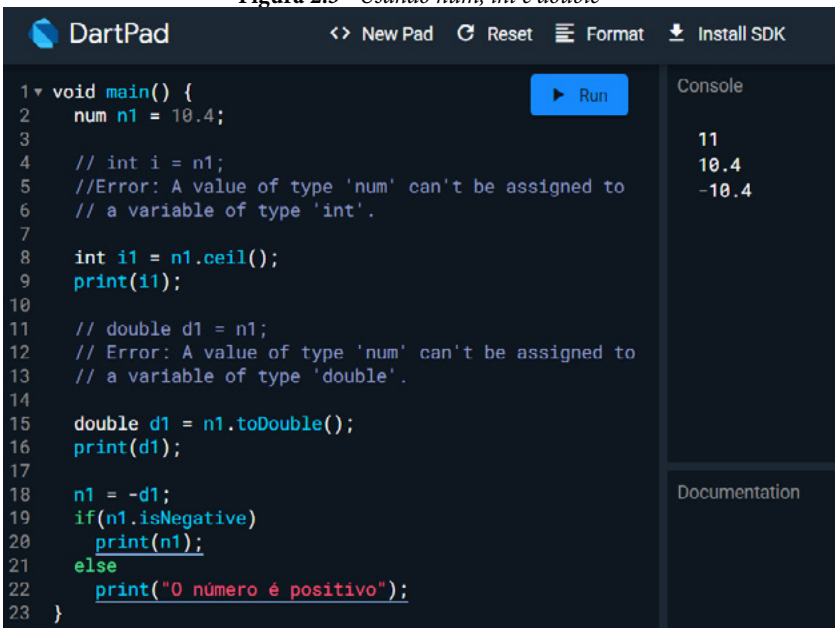
Para trabalhar com números a linguagem Dart disponibiliza o tipo **num** (Number) tendo como subtipos **int** e **double**. O tipo **num** trabalha números de forma mais genérica já os tipos **int** e **double** são os subtipos mais específicos. É importante notar que **num**, **int** e **double** são classes e portanto possuem seus métodos.

A Figura 2.3 apresenta o uso dos tipos **num**, **int** e **double**. Como **int** e **double** herdam de **num** não é possível fazer variáveis do tipo **int** ou **double** receberem um **num**, mas o contrário pode ser feito, ou seja, variáveis do tipo **num** podem receber um **int** ou **double** normalmente.

Para maiores informações sobre classes que trabalham com números recomenda-se a página de referência da Google sobre essa temática[9].

9 https://dart.dev/guides/language/numbers

Figura 2.3 - *Usando num, int e double*



Fonte: Elaborado pelos autores.

2.2.2 Classe String

Em Dart, Strings podem ser representadas por aspas simples ou aspas duplas. Isso facilita a colocação de caracteres aspas simples e aspas duplas no meio de String, ou seja, passa a não ser necessário o uso de um caractere de escape.

Na Figura 2.4 é possível visualizarmos alguns dos principais métodos da classe String:

toUpperCase: retorna a String em "caixa alta", ou seja, toda em maiúscula;

toLowerCase: retorna a String em "caixa baixa", ou seja, toda em minúscula;

length: possui o tamanho (quantidade de caracteres) da String;

substring: retorna a substring que começa no caractere com a posição informada no primeiro parâmetro e termina na posição imediatamente anterior a informada no segundo parâmetro.

Figura 2.4 - *Usando a classe String*

Fonte: Elaborado pelos autores.

A classe String da linguagem Dart possui vários outros métodos como os encontrados em linguagens como Java (FURGERI, 2018a). Para maiores informações sobre métodos e funcionamento da classe String recomenda-se a página de referência da Google sobre essa temática[10].

2.2.3 Classe bool

Essa classe permite armazenar apenas dois valores: ***true*** (verdadeiro) e ***false*** (falso). A Figura 2.5 ilustra o uso da classe bool.

Figura 2.5 - *Usando a classe bool*

Fonte: Elaborado pelos autores.

10 https://api.dart.dev/stable/3.0.4/dart-core/String-class.html

2.2.4 Classe dynamic

Variáveis do tipo ***dynamic*** podem receber valores de todos os outros tipos e é possível mudar esses valores em tempo de execução. A Figura 2.6 mostra a variável 'd' recebendo um valor do tipo **int** (10), depois recebendo uma **String** ("Pedro") e novamente recebendo um valor do tipo **int** (20):

Figura 2.6 - *Usando dynamic*

Fonte: Elaborado pelos autores.

2.2.5 Classe Function

Em Dart é possível criar variáveis e parâmetros do tipo Function (função). Isso significa que comportamentos podem ser parametrizados e/ou armazenados em variáveis, na prática isso possibilita a implementação de comportamentos mais dinâmicos o que facilita e potencializa o processo de programação. Na Figura 2.7 tem-se o uso de classes do tipo Function como variáveis e parâmetros de funções.

Figura 2.7 - *Usando a classe Function*


```
void imprime(String s){
  print('O valor de s é $s');
}

imprimeSoma(int n1, int n2, Function funcaoImpressao){
  int soma = n1 + n2;
  funcaoImpressao(soma.toString());
}

void main() {
  Function funcao = imprime;
  imprimeSoma(4, 5, funcao);

  Function(int, int, Function) f2 = imprimeSoma;
  f2(10, 20, funcao);
}
```

Fonte: Elaborado pelos autores

2.2.6 Classe List

São conjuntos de dados organizados em uma certa ordem (de índices, por exemplo). A Figura 2.8 apresenta diversas formas de criar uma Lista. Abaixo tem-se uma breve explicação sobre cada uma dessas possibilidades:

List lista1 = [];

Nesse caso é criada uma lista vazia e colocada na variável lista1.

List lista2 = [1, 2, 3];

Aqui a lista é criada com valores pré-definidos e fixos.

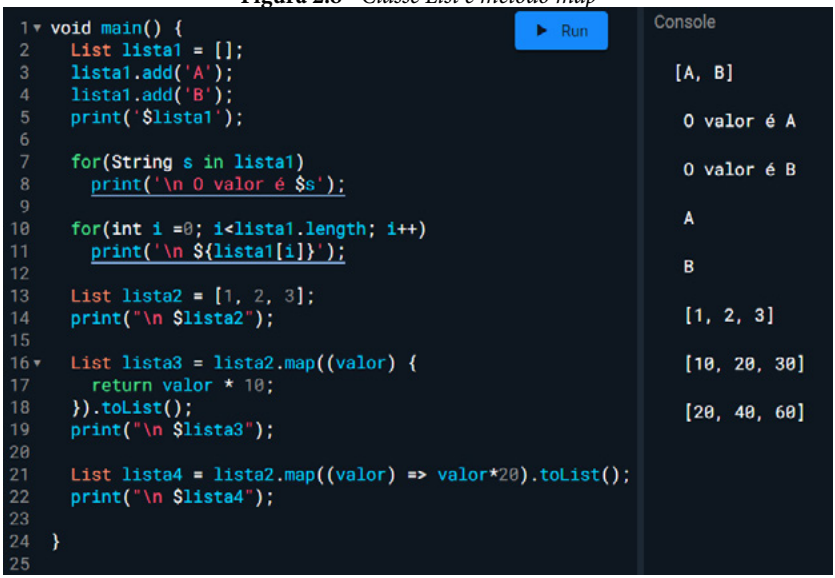
List lista3 = lista2.map((valor) { return valor * 10; }).toList();

O método **map** é aplicado a lista2 e gera um **Iterable** com cada um dos valores de lista2 multiplicados por 10. Na sequência é gerada uma nova lista a partir dos elementos do Iterable e essa nova lista é atribuída à variável lista3.

List lista4 = lista2.map((valor) => valor*20).toList();

Tem-se o uso do método **map** mas aqui de forma mais resumida (usando funções lambda). O efeito prático é o mesmo e ocorre a geração de uma nova lista com seus elementos multiplicados por 20.

Figura 2.8 - *Classe List e método map*




```
void main() {
  List lista1 = [];
  lista1.add('A');
  lista1.add('B');
  print('$lista1');

  for(String s in lista1)
    print('\n O valor é $s');

  for(int i =0; i<lista1.length; i++)
    print('\n ${lista1[i]}');

  List lista2 = [1, 2, 3];
  print("\n $lista2");

  List lista3 = lista2.map((valor) {
    return valor * 10;
  }).toList();
  print("\n $lista3");

  List lista4 = lista2.map((valor) => valor*20).toList();
  print("\n $lista4");

}
```

Fonte: Elaborado pelos autores.

2.2.7 Classe Map

É uma estrutura de dados que organiza os elementos por chave/valor. Através de uma chave é possível recuperar o valor desejado sempre que necessário. A variável **map** da classe **Map**, na Figura 2.9, usa como chaves "Fruta", "Carro" e "Telefone" para os valores "Manga", "Um Carro X" e "Um telefone Y" respectivamente. Posteriormente são criadas as chaves "Computador" e 3 para os valores "Um computador W" e "Número 3".

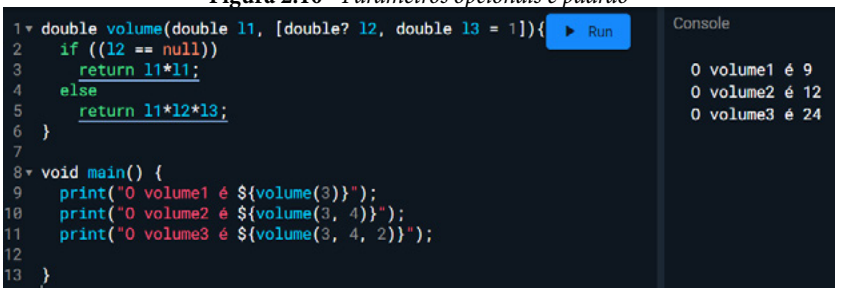
Figura 2.9 - *Classe Map*

Fonte: Elaborado pelos autores

2.2.8 Parâmetros de Funções

Funções, em Dart, podem possuir parâmetros opcionais (delimitados pelos colchetes), ou seja, parâmetros que não precisam, necessariamente, serem informados na chamada da função, a Figura 2.10 ilustra esse conceito no parâmetro l2. Além disso, permite que argumentos opcionais possuam valores padrão, dessa forma quando o argumento opcional não é informado um valor padrão é definido para ele, essa característica pode ser visualizada também na Figura 2.10, no parâmetro l3 (valor 1). Uma outra questão que pode ser vista na Figura 2.10 é o uso do caractere ? junto à classe double para o parâmetro l2, isso significa que l2 pode ser nulo, ou seja, que não necessita ser informado e não precisa de valor padrão. O caractere ? se relaciona com o conceito de *null safety* introduzido no Dart 2.0 (NYSTROM, 2020).

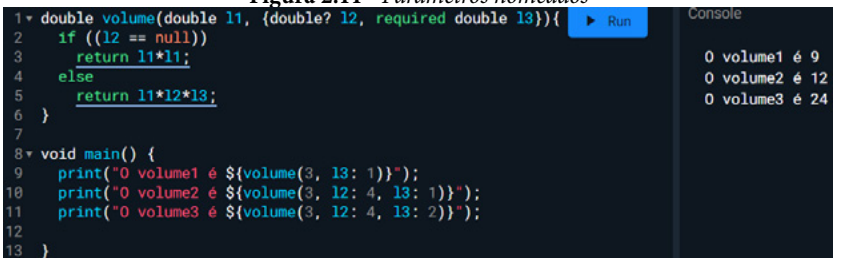
Figura 2.10 - *Parâmetros opcionais e padrão*



Fonte: Elaborado pelos autores.

Uma outra possibilidade, no Dart, é o uso de argumentos nomeados (delimitados pelas chaves), conforme Figura 2.11. O uso de argumentos nomeados facilita a passagem de parâmetros para métodos com muitos parâmetros, visto que ao invés da ordem, utiliza-se o nome do argumento para a passagem de seu valor. Quando um argumento nomeado deve obrigatoriamente ser informado utiliza-se a palavra reservada ***required*** para obrigar o programador a informar esse parâmetro, isso pode ser visto na Figura 2.11.

Figura 2.11 - *Parâmetros nomeados*



Fonte: Elaborado pelos autores

2.3 Orientação a Objetos em Dart

Dart, conforme apresentado anteriormente, é uma linguagem Orientada a Objetos. A Orientação a Objetos é um paradigma de análise (ENGHOLM JR., 2013), projeto (GAMMA et al., 2000) e programação de sistemas (FURGERI, 2018a) e se baseia na ideia de interação entre

unidades de software chamadas objetos. Os objetos são criados a partir de classes que definem seus atributos e funcionalidades.

Para este tópico a orientação a objetos será discutida de forma prática e direcionada à linguagem Dart, ou seja, não serão feitas discussões aprofundadas acerca dos conceitos de Orientação a Objetos e sim de sua aplicação na linguagem Dart.

2.3.1 Declaração de classes e instanciação de objetos

Em Dart a declaração de classes é feita de forma similar a outras linguagens, como Java (FURGERI, 2018a), por exemplo. O uso do comando ***new***, por sua vez, é permitido em Dart mas não obrigatório.

Uma outra questão relacionada a instanciação de objetos é o uso da palavra ***late***. O uso da palavra ***late*** indica que o atributo terá seu valor atribuído de forma tardia, ou seja, fora do construtor. Isso se relaciona com o conceito de *null safety* do Dart, a ideia é forçar o programador a ficar atento a essa questão e evitar a possibilidade de ponteiro nulo.

A Figura 2.12 ilustra a não obrigatoriedade do comando ***new*** e o uso da palavra ***late*** para atribuição tardia de valores aos atributos. É importante notar que a retirada a palavra late, no atributo idade, por exemplo, gera um erro de compilação (*Error: Field 'idade' should be initialized because its type 'int' doesn't allow null.*) pois idade poderia ser nulo, uma vez que não foi definido um construtor para atribuir seu valor, mas Dart não possibilita isso, a não ser que o atributo fosse definido utilizando o operador '?' (***int? idade;***).

Figura 2.12 - *Criação de classes, comando new e palavra reservada late*


```
class Pessoa{
  late String nome;
  late int idade;
}

void main() {
  Pessoa p1 = Pessoa();
  Pessoa p2 = new Pessoa();

  p1.nome = 'Pedro';
  p1.idade = 30;

  p2.nome = 'Ana';
  p2.idade = 60;

  print("A idade de ${p1.nome} é ${p1.idade}");
  print("A idade de ${p2.nome} é ${p2.idade}");

}
```

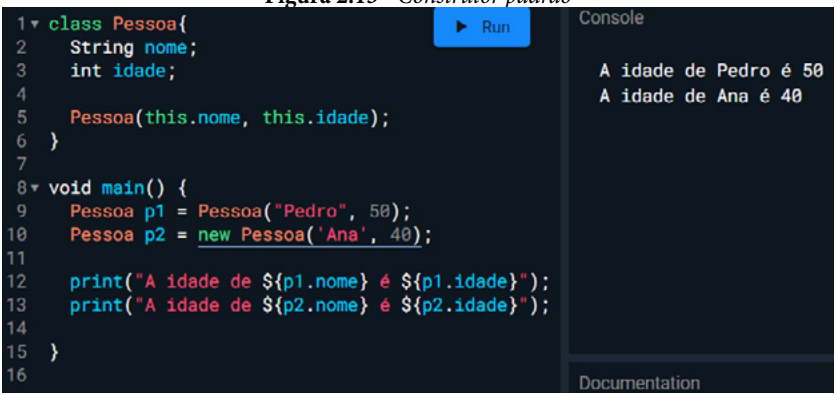
Fonte: Elaborado pelos autores.

2.3.2 Construtor padrão

Os construtores em Dart possuem sintaxe simplificada e resumida, conforme é possível verificar na Figura 2.13. Os construtores em Dart não precisam setar manualmente os valores dos atributos (***this.nome*** = ***nome***, por exemplo). Quando fazemos ***Pessoa(this.nome, this.idade);*** já são atribuídos os valores dos atributos nome e idade. Isso é muito interessante pois deixa o código mais limpo e permite ao programador codificar menos.

Uma outra questão interessante é que, quando tem-se ***Pessoa(this.nome, this.idade);*** os atributos nome e idade serão atribuídos não sendo necessário o uso da palavra ***late***, em sua definição, uma vez que não será feita a atribuição tardia de valores aos atributos. Em outras palavras, quando temos a definição de atributos de uma classe ou eles terão a palavra late (ex: *late String nome;*), ou deverá ser definido um construtor padrão (ex: *Pessoa(this.nome, this.idade);*) para setar seus valores ou deverão ser definidos com o operador ***'?'*** (*String? nome;*), permitindo assim o valor nulo.

Figura 2.13 - *Construtor padrão*



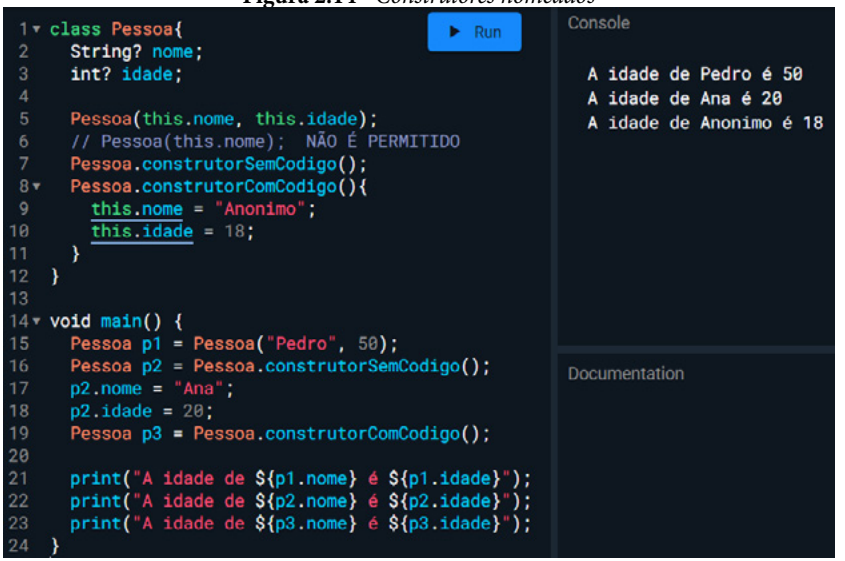

```
class Pessoa{
  String nome;
  int idade;

  Pessoa(this.nome, this.idade);
}

void main() {
  Pessoa p1 = Pessoa("Pedro", 50);
  Pessoa p2 = new Pessoa('Ana', 40);

  print("A idade de ${p1.nome} é ${p1.idade}");
  print("A idade de ${p2.nome} é ${p2.idade}");

}
```

Fonte: Elaborado pelos autores.

2.3.3 Construtores nomeados

Dart permite a criação de construtores nomeados sem código. Na prática um construtor desse tipo apenas vai alocar o espaço necessário para o objeto (seria como um malloc em C, ou seja, não faria nenhum código de inicialização). Na Figura 2.14 é possível ver o construtor ***Pessoa.construtorSemCodigo();*** que não possui código. Também é possível colocar código em construtores nomeados, como é possível ver na Figura 2.14 com o construtor ***Pessoa.construtorComCodigo()***. É importante notar que Dart não trabalha com sobrecarga de construtores não permitindo ***Pessoa(this.nome);*** por exemplo.

Figura 2.14 - *Construtores nomeados*




```dart
class Pessoa{
  String? nome;
  int? idade;

  Pessoa(this.nome, this.idade);
  // Pessoa(this.nome);  NÃO É PERMITIDO
  Pessoa.construtorSemCodigo();
  Pessoa.construtorComCodigo(){
    this.nome = "Anonimo";
    this.idade = 18;
  }
}

void main() {
  Pessoa p1 = Pessoa("Pedro", 50);
  Pessoa p2 = Pessoa.construtorSemCodigo();
  p2.nome = "Ana";
  p2.idade = 20;
  Pessoa p3 = Pessoa.construtorComCodigo();

  print("A idade de ${p1.nome} é ${p1.idade}");
  print("A idade de ${p2.nome} é ${p2.idade}");
  print("A idade de ${p3.nome} é ${p3.idade}");
}
```

Fonte: Elaborado pelos autores

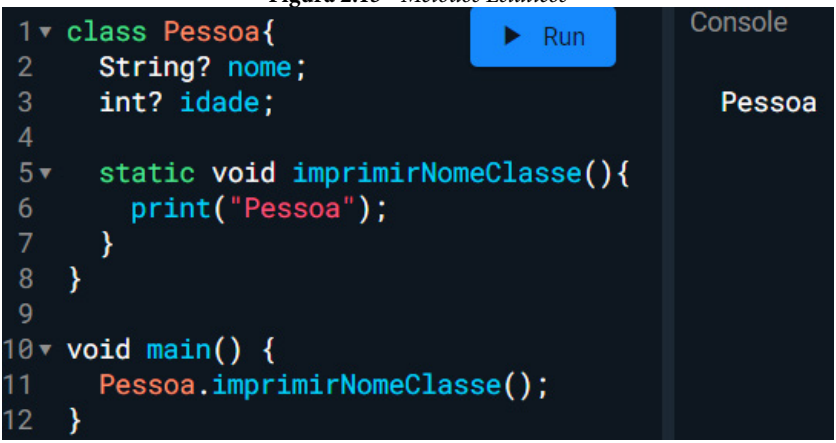
Observação: O DarPad costuma colocar sublinhado abaixo de comandos que considera desnecessários. Na Figura 2.13 ele colocou sublinhado abaixo de new Pessoa('Ana', 40); e isso foi feito por conta do uso da palavra **new** (que não é necessária para a chamada do construtor). Já na Figura 2.14 em this.nome ou this.idade a questão é o uso da palavra **this**, que realmente não é necessária para o código funcionar mas ajuda a ilustrar a alteração de atributos do objeto que está sendo criado (no exemplo **p3**).

2.3.4 Métodos estáticos

Métodos estáticos são aqueles inerentes a uma classe e não a seus objetos, sendo sinalizados pela palavra reservada ***static***. Na prática isso significa que não há sentido ou necessidade da existência de um objeto para acionar um método. Em outras palavras, pode-se visualizar um método estático como um método que ou não "afeta" nenhum objeto de uma dada classe ou "afeta" a todos, ou seja, é um comportamento que

não é de um objeto específico. Na Figura 2.15 tem-se um exemplo de método estático em Dart:

Figura 2.15 - *Métodos Estáticos*



Fonte: Elaborado pelos autores.

2.3.5 Encapsulamento

Dart utiliza um padrão de encapsulamento de atributos, métodos e classes bem simplificado. Para que um atributo, método ou classe seja privado basta colocar '_' (*underline*) na frente do atributo, método ou classe. Em Dart algo ser privado significa que é válido apenas no contexto do arquivo onde foi criado.

Figura 2.16 - *Encapsulamento de atributos, métodos e classes*

a.dart

```
class _A{
  int? _atributo_privado;
  void _metodo_privado(){ }
}
class A{
  int? _atributo_privado;
  void metodo_publico(){}
  void _metodo_privado(){
    _A local_a = _A();
    local_a._metodo_privado();
    local_a._atributo_privado = 10;
    return;
  }
}
```

b.dart

```
import 'a.dart';

class B{
  void usandoAe_A(){
    A a = A();
    a.metodo_publico();
    a._metodo_privado();
    a._atributo_privado = 10;

    _A a2 = A();
  }
}
```

Fonte: Elaborado pelos autores

Na Figura 2.16 tem-se a classe privada _A (que está no arquivo a.dart) pois o nome da classe é precedido por '_'. A classe _A ser privada significa que não poderá ser acessada em outro arquivo, como tentou-se na classe B (que está no arquivo b.dart) e gerou-se o erro (marcação em vermelho abaixo do nome da classe na definição da variável a2). Ainda na classe B tentou-se acessar o _metodo_privado() e o _atributo_privado, ambos da classe pública A (A é pública pois não é precedida por '_'), mas novamente foram obtidos erros decorrentes de ambos serem privados (_metodo_privado() e _atributo_privado).

2.3.6 Métodos gets e sets

Métodos gets e sets são padrões para obter e atribuir valores a atributos, respectivamente, muito utilizados em outras linguagens como Java (FURGERI, 2018a) e gerados por IDEs (Ambientes de Desenvolvimento Integrado) como o NetBeans (APACHE, 2023). O uso de métodos ao invés da atribuição direta permite o acréscimo ou retirada de comportamento quando necessário, uma vez que há o corpo de um método para codificar o que se deseja.

Em Dart existe a flexibilização na criação de gets e sets, ou seja, é possível criar um get e/ou set apenas quando realmente for necessário.

No exemplo da Figura 2.17 vê-se a criação de dois objetos do tipo Pessoa (p1 e p2), o preenchimento de seus atributos (nome e idade) e a impressão deles na tela. E se fosse desejado obter a idade em formato de String ou com a palavra 'anos' adicionada a essa String? Como isso seria feito? Modificar cada uma das linhas de impressão é uma possibilidade, mas isso implica em passar por todos os usos da obtenção do atributo idade e isso, obviamente, é bastante passível de erros.

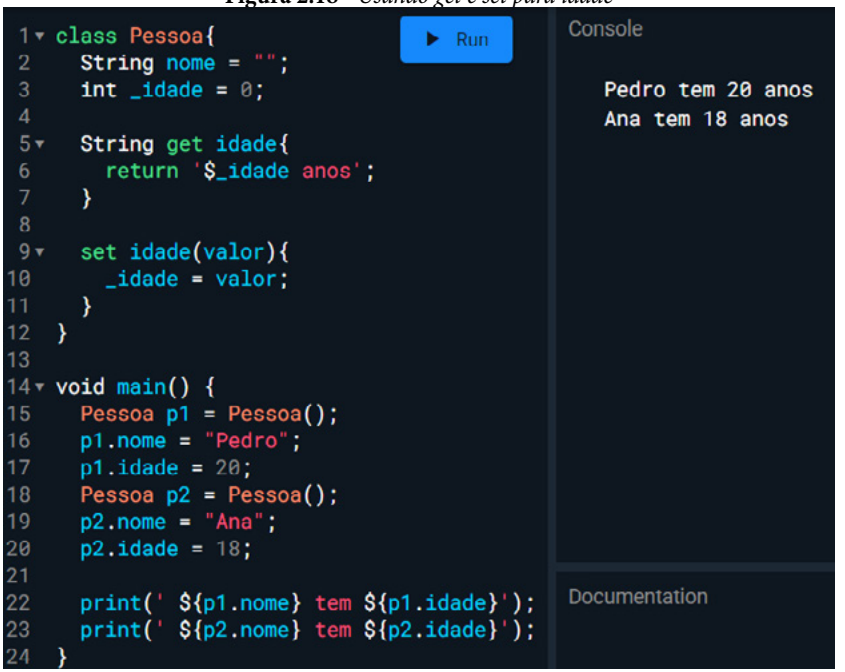
Figura 2.17 - *Obtendo nome e idade sem get*


```
1  class Pessoa{
2    String nome = "";
3    int idade = 0;
4  }
5
6  void main() {
7    Pessoa p1 = Pessoa();
8    p1.nome = "Pedro";
9    p1.idade = 20;
10   Pessoa p2 = Pessoa();
11   p2.nome = "Ana";
12   p2.idade = 18;
13
14   print(' ${p1.nome} tem ${p1.idade}');
15   print(' ${p2.nome} tem ${p2.idade}');
16
17 }
18
```

Fonte: Elaborado pelos autores.

Uma possibilidade para resolver o problema apresentado na Figura 2.17 é criar um **get** específico para **idade**. Com um **get** passa a ser possível acrescentar um comportamento no momento da obtenção do atributo **idade**. A Figura 2.18 ilustra como esse **get** pode ser criado e o respectivo **set** necessário a seu funcionamento.

Figura 2.18 - *Usando get e set para idade*




```dart
class Pessoa{
  String nome = "";
  int _idade = 0;

  String get idade{
    return '$_idade anos';
  }

  set idade(valor){
    _idade = valor;
  }
}

void main() {
  Pessoa p1 = Pessoa();
  p1.nome = "Pedro";
  p1.idade = 20;
  Pessoa p2 = Pessoa();
  p2.nome = "Ana";
  p2.idade = 18;

  print(' ${p1.nome} tem ${p1.idade}');
  print(' ${p2.nome} tem ${p2.idade}');
}
```

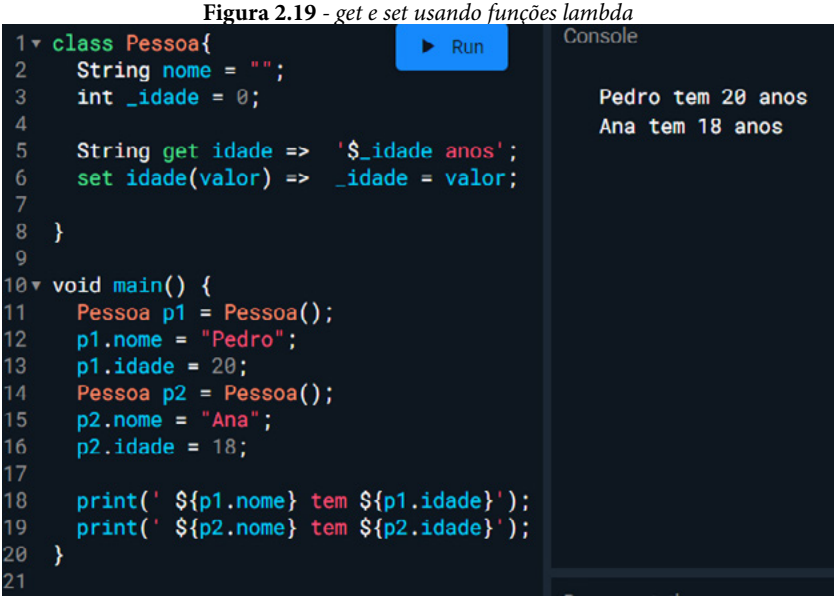
Fonte: Elaborado pelos autores.

É importante notar que Dart possui um padrão para **gets** e **sets**: Quando é feito **p1.idade** o método **String get idade** é acionado e quando é feito **p1.idade = 20** o método **set idade(valor)** é acionado. Dessa forma, **idade** que antes era um atributo real, passa a ser uma espécie de label (um texto fixo) para a chamada dos **get** e **set** para um atributo privado, no caso do exemplo **_idade**.

Essa abordagem gera uma flexibilidade interessante pois caso seja necessário criar um **get** ou **set**, ele será criado e caso não seja, é possível codificar diretamente usando o valor do atributo.

Por fim é possível escrever os métodos get e set usando funções lambda. A Figura 2.19 ilustra essa possibilidade:

Figura 2.19 - *get e set usando funções lambda*



Fonte: Elaborado pelos autores.

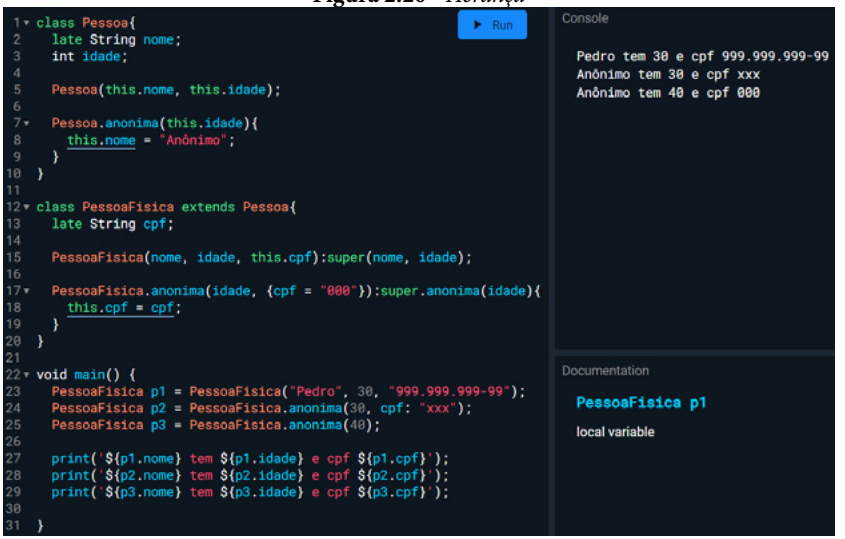
2.3.7 Herança

A herança é um dos principais conceitos da Programação Orientada a Objetos (FURGERI, 2018b). Basicamente ela consiste em aproveitar comportamentos definidos numa classe em suas descendentes, ou seja, classes que herdam dessa classe original. Na prática, com a herança, não é necessário ficar redefinindo comportamentos já previstos e definidos numa classe ancestral (classe da qual outra herda).

Por exemplo, numa classe Ave poderia ser definida uma funcionalidade botarOvo. Nesse contexto, numa classe Galinha (que herdaria de Ave) não seria necessário implementar o comportamento de botarOvo, visto que já teria sido definido num ancestral. Assim a herança se insere como um conceito que nos auxilia a evitar replicar código em diversas classes dificultando sua manutenibilidade, ou seja, uma vez aplicando a herança temos um único ponto de código acerca de um dado comportamento, assim se esse comportamento muda a necessidade de ajuste é feita em apenas um local, diminuindo assim a possibilidade de erros.

A linguagem Dart obviamente possui suporte ao conceito de herança. Na Figura 2.20 é possível vermos a aplicação desse conceito na linguagem Dart:

Figura 2.20 - *Herança*




```dart
class Pessoa{
  late String nome;
  int idade;

  Pessoa(this.nome, this.idade);

  Pessoa.anonima(this.idade){
    this.nome = "Anônimo";
  }
}

class PessoaFisica extends Pessoa{
  late String cpf;

  PessoaFisica(nome, idade, this.cpf):super(nome, idade);

  PessoaFisica.anonima(idade, {cpf = "000"}):super.anonima(idade){
    this.cpf = cpf;
  }
}

void main() {
  PessoaFisica p1 = PessoaFisica("Pedro", 30, "999.999.999-99");
  PessoaFisica p2 = PessoaFisica.anonima(30, cpf: "xxx");
  PessoaFisica p3 = PessoaFisica.anonima(40);

  print('${p1.nome} tem ${p1.idade} e cpf ${p1.cpf}');
  print('${p2.nome} tem ${p2.idade} e cpf ${p2.cpf}');
  print('${p3.nome} tem ${p3.idade} e cpf ${p3.cpf}');

}
```

Fonte: Elaborado pelos autores.

Na Figura 2.20 tem-se o uso da palavra reservada **super**, essa palavra serve para chamar o construtor correspondente na classe ancestral, dessa forma, quando uma PessoaFisica é criada seu construtor codifica o que se relaciona com seus atributos (no caso de PessoaFisica apenas cpf), deixando o resto do "trabalho" para o construtor da classe ancestral (e esse preenche o nome e a idade). Essa abordagem é interessante pois evita a replicação de código em diversas classes o que prejudicaria a manutenibilidade.

2.3.8 Sobrescrita (aplicação no método toString)

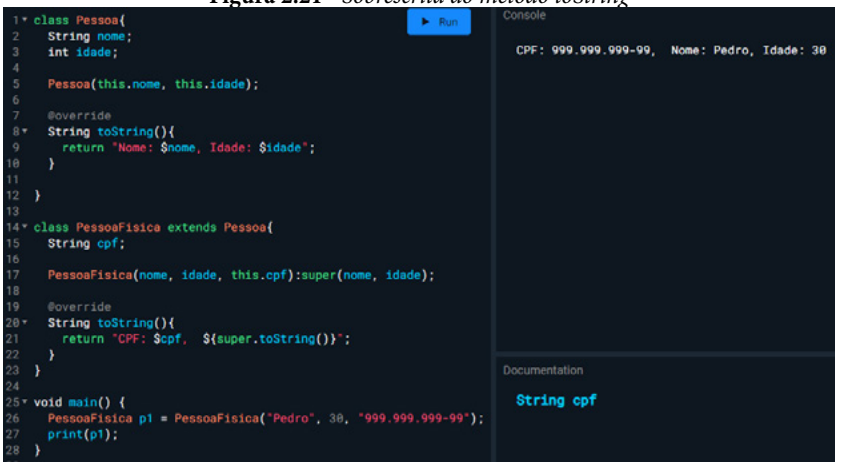
A sobrescrita é um tipo de polimorfismo que consiste basicamente em redefinir o comportamento definido em um ancestral. Uma classe Cao, por exemplo, poderia ter uma funcionalidade *latir*, com um comportamento padrão de latido, entretanto pode ser desejado rede-

finir esse comportamento para algumas raças específicas que poderiam latir diferente (emitir o som do latido de forma diferente). Dessa forma, todo Cao teria a funcionalidade de *latir*, mas um Pitbull, por exemplo, poderia ter essa funcionalidade modificada, ou seja, redefinida para ser feita de acordo com a forma que um Pitbull late.

Dart possui suporte à sobrescrita de métodos e à classe Object (GOOGLE, 2023d) como ancestral padrão de outras classes. Na classe Object são definidos métodos que podem ser sobrescritos em seus descendentes, um dos mais conhecidos é o método *toString()*. A ideia do método *toString()* é possibilitar a conversão de um objeto para o formato de String, isso é muito interessante quando busca-se, por exemplo, imprimir os dados de um objeto na tela.

Na Figura 2.21 a classe Pessoa, que herda de Object (quando uma classe não herda de alguém de forma explícita, ela herda de Object), sobrescreve o método *toString()*. Por sua vez, a classe PessoaFisica (que herda de Pessoa) sobrescreve novamente o método toString() mas através da palavra *super* aproveita o comportamento implementado na classe Pessoa. Em outras palavras, para a funcionalidade *toString()* uma PessoaFisica faz o que uma Pessoa faz e "mais alguma coisa", por isso chama o *toString()* de Pessoa mas também acrescenta algum comportamento próprio.

Figura 2.21 - *Sobrescrita do método toString*

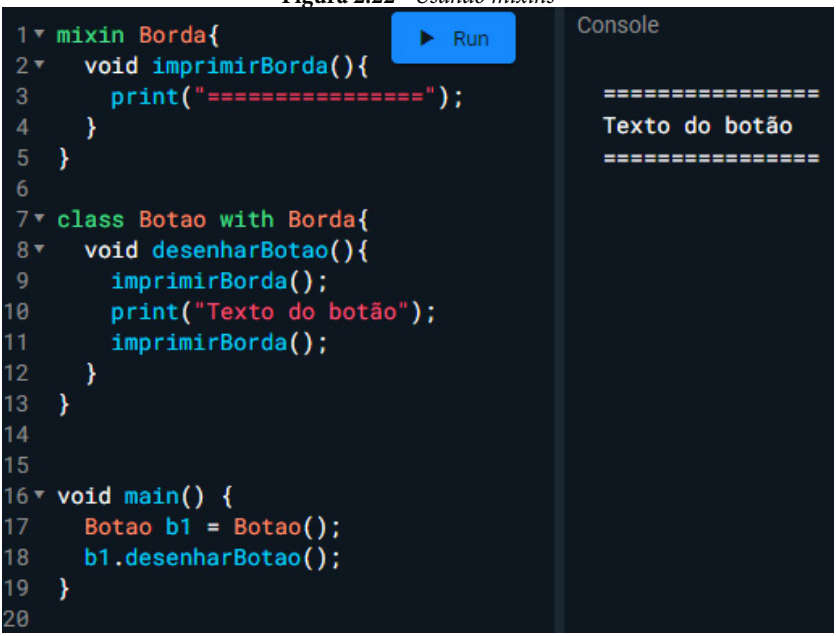


Fonte: Elaborado pelos autores.

2.3.9 Mixins

Dart não possui suporte a herança múltipla, algo encontrado em linguagens como C++ (STROUSTRUP, 2013). Para resolver essa necessidade Dart utiliza Mixins. Mixins permitem adicionar métodos sem usar herança. A Figura 2.22 ilustra o conceito de Mixins:

Figura 2.22 - *Usando mixins*



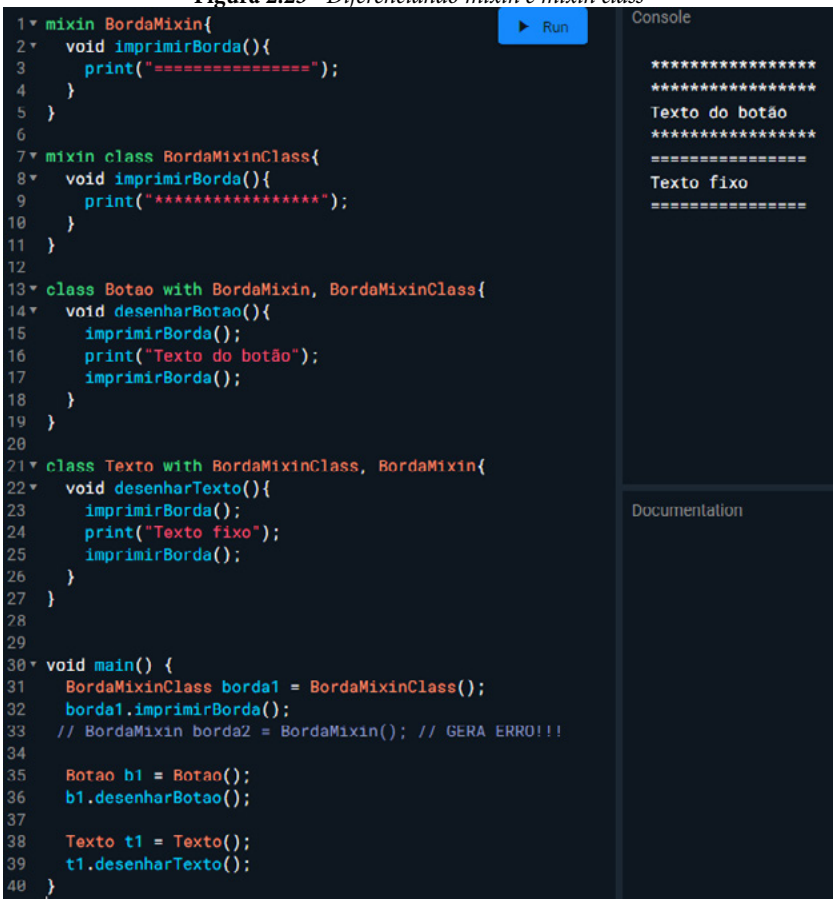

```
mixin Borda{
  void imprimirBorda(){
    print("==========");
  }
}

class Botao with Borda{
  void desenharBotao(){
    imprimirBorda();
    print("Texto do botão");
    imprimirBorda();
  }
}


void main() {
  Botao b1 = Botao();
  b1.desenharBotao();
}

```

Fonte: Elaborado pelos autores.

Para definir um Mixin, na versão atual do Dart, existem duas formas: usando a palavra reservada *mixin* ou usando *mixin class*. Ao usar apenas *mixin* não pode-se utilizar o que foi definido como uma classe regular (na Figura 2.23 não é permitido instanciar um *mixin* BordaMixin na variável *borda2*), mas quando *mixin class* é utilizado tem-se a possibilidade de uso tanto como classe regular como quanto mixin (GOOGLE, 2023e).

Uma outra questão importante é que, quando dois mixins são utilizados em uma classe e há um conflito, o último mixin colocado é o que "fica valendo". Na Figura 2.23 a classe Botao tem como mixins BordaMixin, BordaMixinClass (nessa ordem) e ambos possuem o método imprimirBorda(), nesse caso a classe Botao vai usar o *imprimirBorda()* de BordaMixinClass. Já para a classe Texto a ordem de definição dos mixins é BordaMixinClass, BordaMixin, dessa forma, para Texto, será utilizado o *imprimirBorda()* de BordaMixin, uma vez que ele foi o último a ser definido para a classe Texto.

Figura 2.23 - *Diferenciando mixin e mixin class*




```
mixin BordaMixin{
  void imprimirBorda(){
    print("==========");
  }
}

mixin class BordaMixinClass{
  void imprimirBorda(){
    print("**********");
  }
}

class Botao with BordaMixin, BordaMixinClass{
  void desenharBotao(){
    imprimirBorda();
    print("Texto do botão");
    imprimirBorda();
  }
}

class Texto with BordaMixinClass, BordaMixin{
  void desenharTexto(){
    imprimirBorda();
    print("Texto fixo");
    imprimirBorda();
  }
}


void main() {
  BordaMixinClass borda1 = BordaMixinClass();
  borda1.imprimirBorda();
  // BordaMixin borda2 = BordaMixin(); // GERA ERRO!!!

  Botao b1 = Botao();
  b1.desenharBotao();

  Texto t1 = Texto();
  t1.desenharTexto();
}
```

Fonte: Elaborado pelos autores.

2.3.10 Classes e métodos abstratos

Uma classe abstrata é uma classe que não permite a instanciação de objetos a partir dela. É utilizada para reaproveitamento de alguns comportamentos para classes descendentes.

Métodos abstratos são métodos que obrigam sua implementação em classes descendentes concretas (não abstratas), exceto que já tiveram esse comportamento implementado em algum ancestral da sua hierar-

quia. Essa obrigação ou "contrato" auxilia o programador a compreender comportamentos obrigatórios numa dada hierarquia.

Dart possui suporte a ambos os conceitos. Para classes abstratas basta utilizar a palavra abstract antes da definição da classe. Para métodos abstratos basta definir um método sem código (somente a assinatura) que ele será assumido como abstrato. A Figura 2.24 ilustra o uso desses conceitos:

Figura 2.24 - *Classes e métodos abstratos*

```
abstract class Mamifero{
  void mamar();
}

abstract class Cetaceo extends Mamifero{
  void nadar(){
    print("Estou nadando ");
  }
}

class Baleia extends Cetaceo{
  @override
  void mamar(){
    print("Vou mamar como uma Baleia");
  }
}

class Golfinho extends Cetaceo{
  @override
  void mamar(){
    print("Vou mamar como uma Golfinho");
  }
  @override
  void nadar(){
    print("Golfinhos nadam de forma especial");
  }
}

void main() {
  // Mamifero m1 = Mamifero(); ERRO a classe Mamifero não pode ser instanciada
  // Cetaceo c1 = Cetaceo();   ERRO a classe Cetaceo não pode ser instanciada
  Baleia b1 = Baleia();
  b1.mamar();
  b1.nadar();
  Golfinho g1 = Golfinho();
  g1.mamar();
  g1.nadar();
}
```

Run

Console

```
Vou mamar como uma Baleia
Estou nadando
Vou mamar como uma Golfinho
Golfinhos nadam de forma especial
```

Documentation

```
Golfinho Golfinho()
```

Fonte: Elaborado pelos autores.

No exemplo da Figura 2.24, temos as classes abstratas **Mamifero** e **Cetaceo**, além do método abstrato *mamar()*. **Cetaceo** não é obrigado a implementar *mamar()* por ser abstrato, mas vai repassar essa obrigação para seus descendentes. **Cetaceo** define o método *nadar()* mas ele não é abstrato, dessa forma só precisa redefini-lo a classe que realmente precisar. Tanto as classes **Baleia** como **Golfinho** precisaram implementar *mamar()* pois **Cetaceo** não o fez e, como ambas são classes concretas, terão de fazer isso obrigatoriamente. Em **Golfinho**

precisou-se acrescentar um comportamento específico para a funcionalidade *nadar()*, pois para fins desse exemplo, Golfinhos nadam de forma especial e diferenciada necessitando, portanto, de programação específica. Já as Baleias mantiveram a forma de *nadar()* definida em **Cetaceo**.

2.4 Palavras e símbolos complementares

Algumas palavras e símbolos possuem grande impacto na programação em linguagem Dart. As palavras const e final possuem similaridades e diferenças, os símbolos '?' e '!' são utilizados em contexto similares mas possuem efeitos diferentes que precisam ser discutidos e compreendidos adequadamente. Este tópico se propõe a tratar dessas temáticas.

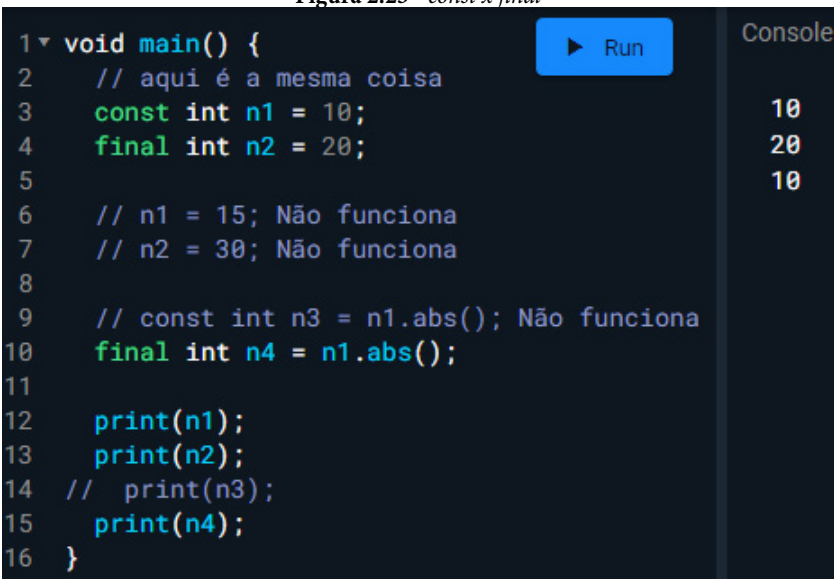
2.4.1 const x final

A palavra const não permite a alteração da variável e o valor é atribuído em tempo de compilação, ou seja, deve ser um constante (não pode ser advindo de uma função que precisa ser calculada).

A palavra final não permite a alteração de uma variável e o valor é colocado em tempo de execução, ou seja, pode ser calculado por uma função mas uma vez atribuído não poderá mais ser modificado.

A Figura 2.25 mostra que as duas palavras se equivalem quando atribuímos valores fixos a uma variável, como é possível ver em n1 e n2. Mas mostra também que quando busca-se fazer uma variável com a palavra const receber o resultado de uma função a ser calculada, isso não é aceito, como é possível ver na tentativa de se definir n3:

Figura 2.25 - *const x final*




```dart
void main() {
  // aqui é a mesma coisa
  const int n1 = 10;
  final int n2 = 20;

  // n1 = 15; Não funciona
  // n2 = 30; Não funciona

  // const int n3 = n1.abs(); Não funciona
  final int n4 = n1.abs();

  print(n1);
  print(n2);
//  print(n3);
  print(n4);
}
```

Fonte: Elaborado pelos autores.

2.4.2 Construtor const

Construtores **const** são construtores cujas classes são compostas apenas por atributos final. Os objetos na prática são "constantes em tempo de compilação". Isso é especialmente útil em Flutter para evitar recriar objetos que não mudam o estado (utiliza-se o mesmo objeto já que ele é um constante). No exemplo da Figura 2.26 é possível verificar que **p1** é igual a **p2** pois na prática são o mesmo objeto, isso porque o Dart, quando identifica a tentativa de nova criação de um "objeto constante" já definido anteriormente, simplesmente faz o apontamento para o objeto já criado anteriormente.

Figura 2.26 - *Construtores const*


```dart
class Pessoa{
  final String nome;
  final int idade;

  const Pessoa(this.nome, this.idade);
}

void main() {
  Pessoa p1 = const Pessoa("Pedro", 10);
  Pessoa p2 = const Pessoa("Pedro", 10);
  Pessoa p3 = const Pessoa("Pedro", 20);

  Pessoa p4 = Pessoa("Pedro", 10);
  Pessoa p5 = Pessoa("Pedro", 10);
  Pessoa p6 = Pessoa("Pedro", 20);

  if(p1 == p2){
    print("p1 e p2 são o mesmo objeto");
  }
  if(p1 == p4){
    print("p1 e p4 são o mesmo objeto");
  }
  if(p4 == p5){
    print("p4 e p5 são o mesmo objeto");
  }
  if(p3 == p6){
    print("p3 e p6 são o mesmo objeto");
  }
}
```

Fonte: Elaborado pelos autores.

2.4.3 Operadores '!' e '?'

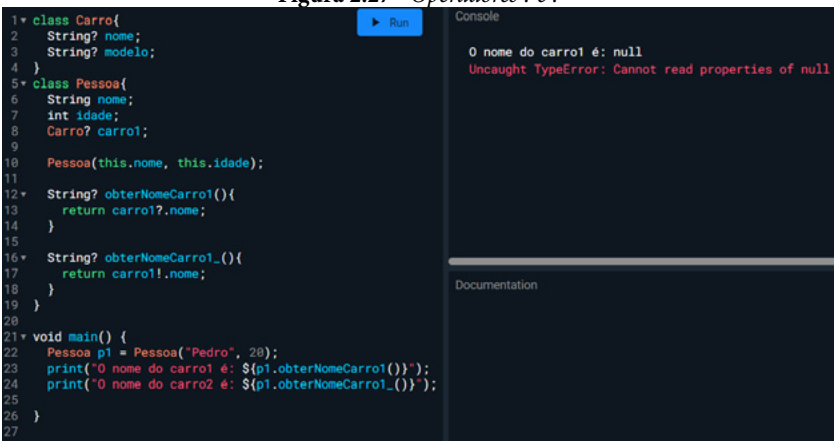
Quando objetos são utilizados, um deles pode acessar um campo ou método de um outro objeto interno. Entretanto o que deve acontecer se esse objeto interno for nulo? É isso que os operadores '!' e '?' tratam.

- operador '!' verificará se há nulo, se houver, irá disparar uma exceção.
- operador '?' irá retornar nulo (null).

A Figura 2.27 mostra o efeito do uso dos operadores '!' e '?' para a tentativa de acesso ao atributo *carro1<operador>.nome*. No exemplo, *carro1* é nulo, ou seja, não há como haver o retorno de um *nome* para o *carro1*. No método *obterNomeCarro1()* o operador é '?', nesse caso, como *carro1* é nulo, ocorre simplesmente o retorno do valor *null* (que é impresso

na tela). Já o método *obterNomeCarro1()* utiliza o operador '!', nesse contexto como *carro1* é nulo ele dispara uma exceção.

Figura 2.27 - *Operadores ? e !*




```
class Carro{
  String? nome;
  String? modelo;
}
class Pessoa{
  String nome;
  int idade;
  Carro? carro1;

  Pessoa(this.nome, this.idade);

  String? obterNomeCarro1(){
    return carro1?.nome;
  }

  String? obterNomeCarro1_(){
    return carro1!.nome;
  }
}

void main() {
  Pessoa p1 = Pessoa("Pedro", 20);
  print("O nome do carro1 é: ${p1.obterNomeCarro1()}");
  print("O nome do carro2 é: ${p1.obterNomeCarro1_()}");

}
```

Fonte: Elaborado pelos autores.

3 COMPONENTES VISUAIS BÁSICOS

A partir deste tópico serão trabalhados os componentes visuais do Flutter. Como serão muitas classes tem-se a divisão em três partes e este tópico tratará da primeira parte. Em Flutter os componentes visuais se chamam widgets (GOOGLE, 2023f).

Os widgets são blocos de construção reutilizáveis que representam partes da interface do usuário, como botões, caixas de texto e imagens. Eles podem ser combinados e aninhados uns dentro dos outros para criar interfaces mais elaboradas.

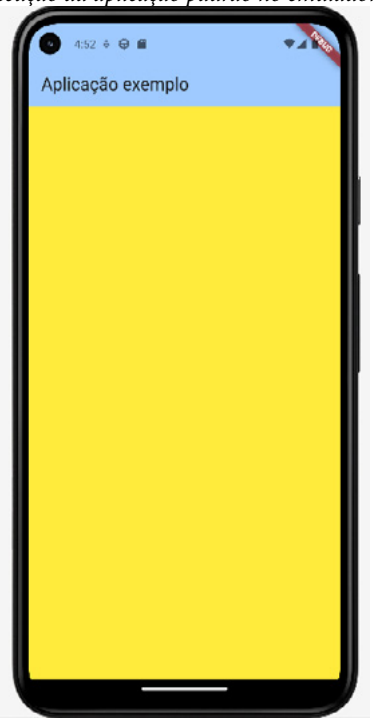
Para simplificar a compreensão do funcionamento de cada um dos widgets que serão apresentados, será utilizada a aplicação padrão gerada pelo Android Studio, com modificações para torná-la mais clara e concisa. Dessa forma, em todos os exemplos deste e dos próximos tópicos que abordam widgets, o foco será no conteúdo do método *body()*. Isso permitirá concentrar a atenção nas partes relevantes do código e entender melhor como cada widget é utilizado e influencia a interface do usuário. A Figura 3.1 mostra a aplicação padrão gerada pelo Android Studio com modificações e a Figura 3.2 o resultado dessa aplicação no emulador do Android 13.

Figura 3.1 - *Aplicação padrão modificada*

```dart
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.blue),
        useMaterial3: true,
      ),  // ThemeData
      home: const MyHomePage(),
    );  // MaterialApp
  }
}
class MyHomePage extends StatefulWidget {
  const MyHomePage();
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text("Aplicação exemplo"),
      ),  // AppBar
      body: body(), // This trailing comma makes auto-formatting nicer
    );  // Scaffold
  }

  body() { return Container(color: Colors.yellow); }
}
```

Fonte: Elaborado pelos autores.

Figura 3.2 - *Execução da aplicação padrão no emulador do Android 13*



Fonte: Elaborado pelos autores.

No conteúdo do método *body()* (ver Figura 3.1) tem-se a criação de um widget Container. O Container é uma das classes mais importantes do Flutter. É parecido com o JPainel do Java Swing (FURGERI, 2018a) ou os Fragments do Android Java/Kotlin (LECHETA, 2017). Consiste num "espaço de tela" onde é possível colocar outro widget. Nesse espaço é possível definir configurações como preenchimento, bordas, margens, paddings, etc. A Figura 3.3 apresenta as definições de altura (*height* para 100) e largura (*width* para 200), do Container, no método *body()*. Já a Figura 3.4 mostra o resultado dessa codificação na tela.

Figura 3.3 - *Definindo altura e largura de um Container*

```
body() {
  return Container(
    color: Colors.yellow,
    height: 100,
    width: 200,
  );
}
```

Fonte: Elaborado pelos autores.

Figura 3.4 - *Emulador Android 13 usando Container*

Fonte: Elaborado pelos autores.

Para a classe Container existem diversos parâmetros opcionais que podem ser passados como parâmetro em seu construtor. Além da altura (*height*) e largura (*width*) é possível definir, por exemplo, os parâmetros

margin e *padding*. A Figura 3.5 apresenta um exemplo de uso desses parâmetros e seu resultado no emulador Android 13.

Figura 3.5 - *Usando margin e padding*

Fonte: Elaborado pelos autores.

O atributo *margin* é a margem externa ao widget, já o atributo *padding* é a margem interna ao widget (para colocação de um widget dentro dele). O atributo *child* define o widget "filho" que será colocado dentro do widget.

A classe **EdgeInsets** permite a definição de cada uma das margens individualmente (usando *only*) ou de todas de uma vez (usando *all*). É importante notar que, na Figura 3.5, para *margin*, não foi definida a margem inferior (*bottom*).

3.1 BoxDecoration

O BoxDecoration é um widget que possibilita a criação de uma borda "estilizada" no widget onde é aplicado, mas seu uso conflita com o uso do parâmetro *color*.

A Figura 3.6 mostra uma tentativa de se atribuir a cor verde (*Colors. green*) ao parâmetro *color*, do Container interno (o que foi definido em *child*), e ao mesmo tempo atribuir um BoxDecoration ao parâmetro *decoration*. Essa operação gera conflito no esquema de widgets do Flutter.

Para colocar uma cor no Container interno é necessário definir essa cor no BoxDecoration e não no Container, conforme é possível verificar na Figura 3.7.

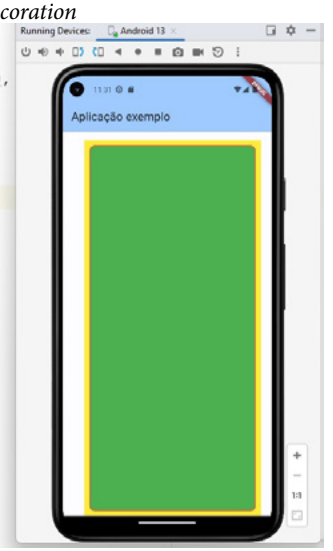
Figura 3.6 - *Conflito BoxDecoration x color*

```
body() {
  return Container(
    margin: EdgeInsets.only(top: 16, left: 40, right:  20),
    padding: EdgeInsets.all(10),
    color: Colors.yellow,
    child: Container(
      color:Colors.green,
      decoration: BoxDecoration(
        borderRadius: BorderRadius.circular(10.0),
        border: Border.all(
          color: Colors.red,
          style: BorderStyle.solid,
          width: 1.50
        )  // Border.all
      ),  // BoxDecoration
    )  // Container
  );  // Container
  }
}
```

Fonte: Elaborado pelos autores.

Figura 3.7 - *Usando BoxDecoration*

```
body() {
  return Container(
    margin: EdgeInsets.only(top: 16, left: 40, right:  20),
    padding: EdgeInsets.all(10),
    color: Colors.yellow,
    child: Container(
      decoration: BoxDecoration(
        color:Colors.green,
        borderRadius: BorderRadius.circular(10.0),
        border: Border.all(
          color: Colors.red,
          style: BorderStyle.solid,
          width: 1.50
        )  // Border.all
      ),  // BoxDecoration
    )  // Container
  );  // Container
  }
}
```



Fonte: Elaborado pelos autores.

Além do atributo *color* o **BoxDecoration** possui outros parâmetros. O *borderRadius* possibilita, entre outras possibilidades, que seja definida uma borda arredondada no widget que ela decora (nesse caso o **Container** interno). O valor 10 para *BorderRadius.circular* torna os cantos do **Container** interno arredondados de acordo com esse valor, ou seja, se esse valor for diminuído os cantos ficam menos arredondados, caso seja aumentado o arredondamento da borda também aumenta.

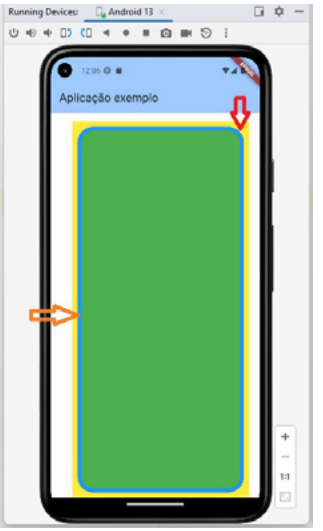
Já o parâmetro *border* define as características da borda em si, como a cor (color: *Colors.red*), o estilo (*style:BorderStyle.solid*) ou a espessura (*width:1.5*).

No exemplo da Figura 3.8, o *BorderRadius.circular* é mudado de 10 para 30. A cor da borda também é modificada para *blue* (antes era *red*) e a espessura mudada para 7.5 (antes era 1.5).

Além das customizações apresentadas, é possível estilizar cada uma das bordas (esquerda, direita, topo e piso) de forma separada (GOOGLE, 2023i).

Figura 3.8 - *Mudando parâmetros do BoxDecoration*

```
body() {
  return Container(
    margin: EdgeInsets.only(top: 16, left: 40, right:  20),
    padding: EdgeInsets.all(10),
    color: Colors.yellow,
    child: Container(
      decoration: BoxDecoration(
        color:Colors.green,
        borderRadius: BorderRadius.circular(30.0),
        border: Border.all(
          color: Colors.blue,
          style: BorderStyle.solid,
          width: 7.50
        ) // Border.all
      ), // BoxDecoration
    ) // Container
  ); // Container
 }
}
```



Fonte: Elaborado pelos autores.

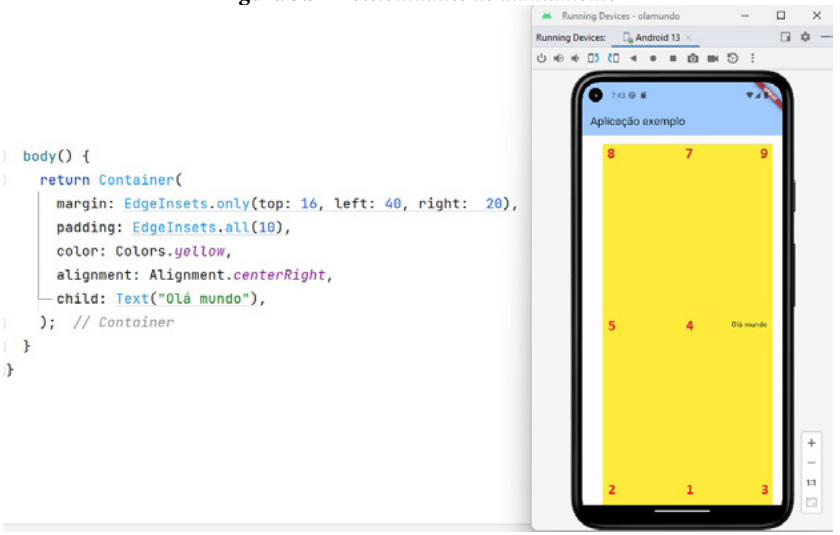
Conforme já descrito anteriormente, a leitura da API do Flutter (GOOGLE, 2023j) é fundamental e traz o detalhamento necessário a contextos mais específicos. Ou seja, esse livro não se propõe esgotar a temática dos widgets do Flutter, nem detalhá-los à exaustão, seu foco é apresentá-los e direcionar o leitor a estudos complementares quando houver necessidade.

3.2 Alinhamento (alignment)

O *alignment* é a propriedade que alinha um widget interno dentro de seu widget externo. Existem 9 possibilidades de alinhamento:

1. *bottomCenter*: parte inferior do widget externo na vertical e centralizado na horizontal;
2. *bottomLeft*: parte inferior do widget externo na vertical e alinhado à esquerda na horizontal;
3. *bottomRight*: parte inferior do widget externo na vertical e alinhado à direita na horizontal;
4. *center*: centralizado na vertical e na horizontal;
5. *centerLeft*: centralizado no widget externo, na vertical, e alinhado à esquerda na horizontal;
6. *centerRight*: centralizado no widget externo, na vertical, e alinhado à direita na horizontal;
7. *topCenter*: parte superior do widget externo na vertical e centralizado na horizontal;
8. *topLeft*: parte superior do widget externo na vertical e alinhado à esquerda na horizontal;
9. *topRight*: parte superior do widget externo na vertical e alinhado à direita na horizontal.

Figura 3.9 - *Possibilidades de alinhamento*



Fonte: Elaborado pelos autores.

A Figura 3.9 apresenta as possibilidades de alinhamento no Container (widget externo) e o alinhamento do widget Text("Olá mundo") criado (widget interno), nesse caso *Alignment.centerRight*.

Para facilitar a visualização foi colocado o número que identifica cada uma das possibilidades de alinhamento. Obviamente a opção 6 não existe na Figura 3.9 pois é exatamente o alinhamento *Alignment.centerRight* (se o número fosse colocado iria dificultar a visualização do widget Text("Olá mundo")).

3.3 Text

O widget Text (WINDMILL, 2020) é responsável por exibir uma sequência de texto com um estilo único. O primeiro parâmetro (obrigatório) é a String que será apresentada no Text. Os demais parâmetros são opcionais e nomeados.

Na Figura 3.10 tem-se o uso do widget Text e alguns dos principais parâmetros que podem ser passados em sua construção:

textAlign: apresenta o alinhamento do texto em relação ao widget externo (que contém o Text). *TextAlign.justify* alinha o texto de forma justificada.

overflow: determina o que será feito quando não houver espaço suficiente para o texto. *TextOverflow.elipsis* coloca "..." quando não há espaço para todo o texto.

maxLines: determina o número de linhas máximo para exibir o texto.

style: determina o estilo do texto (tamanho da fonte, cor, opacidade, etc.).

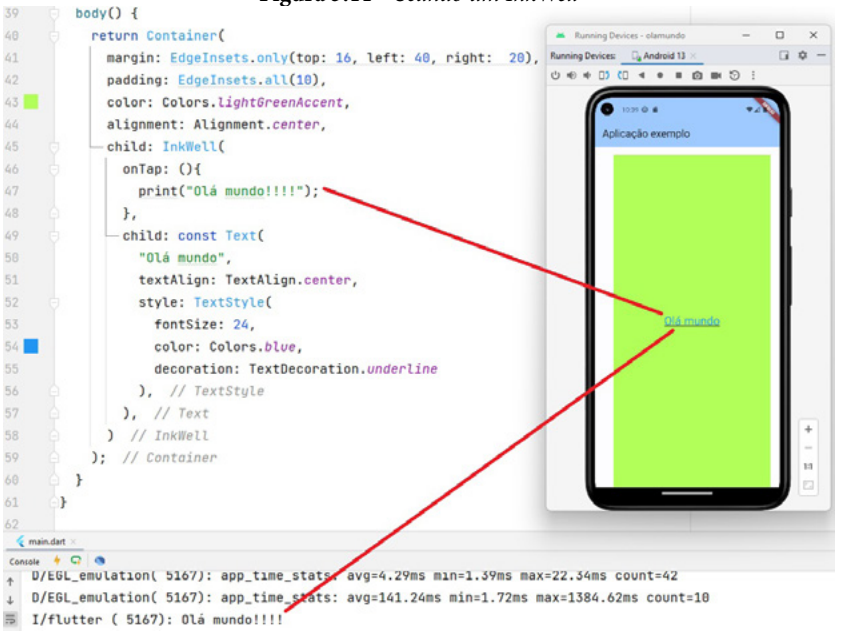
Figura 3.10 - *Usando o widget Text*

Fonte: Elaborado pelos autores.

3.4 InkWell

O widget **InkWell** possibilita deixar um **Text** "clicável" e executar uma ação quando clicado. A Figura 3.11 ilustra um exemplo de uso do **InkWell**:

Figura 3.11 - *Usando um InkWell*



Fonte: Elaborado pelos autores.

A função que determina o comportamento executado no clique sobre o **InkWell** é a função definida no parâmetro *onTap*. No caso do exemplo da Figura 3.11 foi definida uma função que apenas imprime *"Olá mundo!!!!"* no prompt. Ainda no mesmo exemplo é possível verificar que o **InkWell** utiliza um **Text** como child, ou seja, o **InkWell** utiliza como widget interno o **Text**. Por fim, para aplicar o efeito normalmente utilizado em links de páginas na web definiu-se a cor do **Text** para azul (*color: Colors.blue*) e a decoração para sublinhado (*decoration: TextDecoration.underline*).

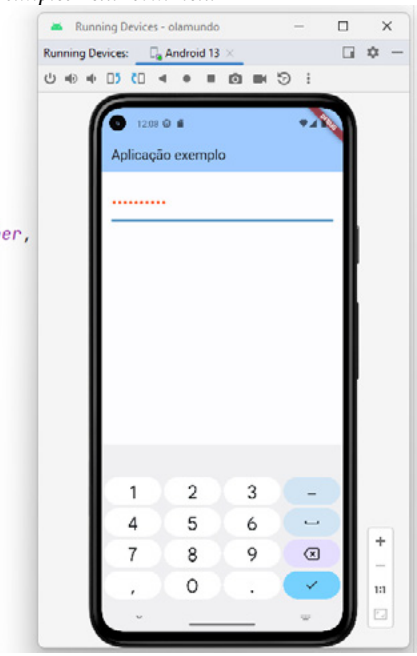
3.5 TextFormField

O widget **TextFormField** é um campo de edição de texto usado em formulários. Possui diversos parâmetros opcionais definidos em seu construtor. O *obscureText* é um deles e permite definir se os caracteres

do texto devem ficar visíveis (*true*) ou não (*false*), isso é muito utilizado para campos de senha. Um outro parâmetro também bastante utilizado é o *keyboardType*, esse parâmetro permite definir o tipo do teclado virtual que será utilizado (texto, número, telefone, endereço de email, etc.). A Figura 3.12 apresenta um exemplo simples de uso do TextFormField:

Figura 3.12 - *Exemplo simples TextFormField*

```
body() {
  return Container(
    margin: EdgeInsets.all(16),
    child: TextFormField(
      // senha
      obscureText: true,
      // tipo do teclado
      keyboardType: TextInputType.number,
      style: const TextStyle(
        fontSize: 25,
        color: Colors.deepOrange
      ),  // TextStyle
    )  // TextFormField
  );  // Container
}
}
```



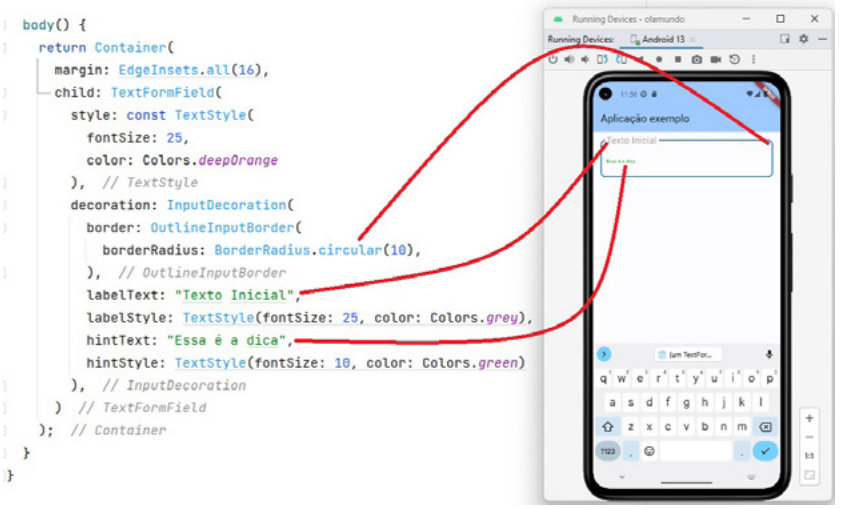
Fonte: Elaborado pelos autores.

Um outro parâmetro opcional bastante importante é o *decoration*. Com o *decoration* é possível definir os aspectos visuais do **TextFormField** através de um widget **InputDecoration** ou outro similar. Um **InputDecoration** é responsável por "decorar"/"estilizar" o widget decorado (um **TextFormField**, por exemplo).

O **InputDecoration** possui diversos parâmetros opcionais definidos em seu construtor. O *border* permite definir a forma da borda do widget decorado (um **TextFormField**, por exemplo). O *labelText*

permite definir um texto que descreve o widget decorado e *labelStyle* irá permitir estilizar esse *labelText*. O *hintText* fornece a possibilidade de definir uma "dica" do que deve ser preenchido no widget decorado e *hintStyle* é a possibilidade de customização visual dessa dica, da mesma forma que *labelStyle* customiza *labelText*. A Figura 3.13 mostra o uso do **InputDecoration** para estilizar um **TextFormField**:

Figura 3.13 - *InputDecoration no TextFormField*

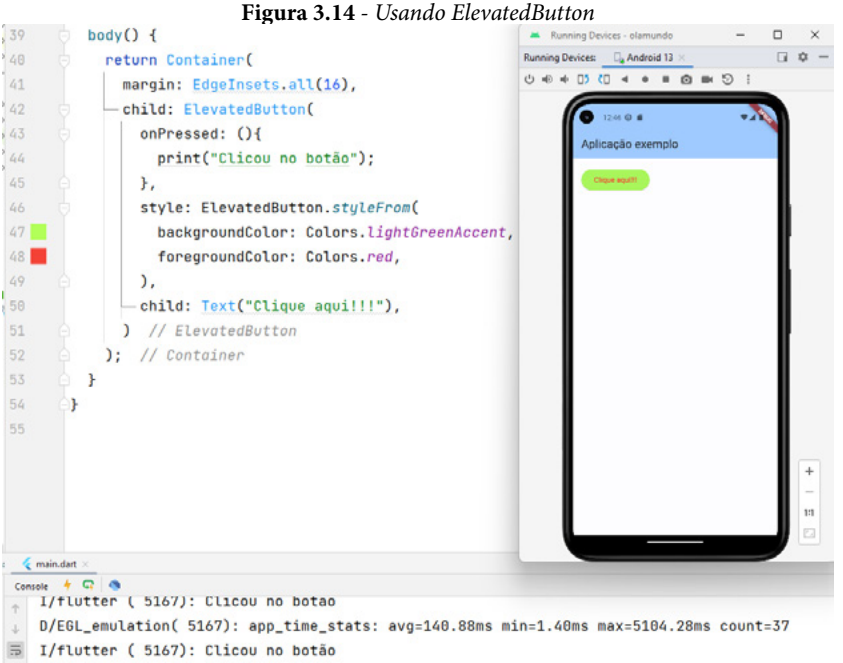


Fonte: Elaborado pelos autores.

3.6 ElevatedButton

O widget ElevatedButton é uma das classes de botões do Flutter. No parâmetro *onPressed* tem-se a definição da função a ser executada no clique do botão. Para definir a cor da letra (*foregroundColor*) do botão bem como a cor de fundo (*backgroundColor*) do botão pode ser utilizado o parâmetro *style*. Já o texto do botão pode utilizar como *child* um widget Text. A Figura 3.14 apresenta um exemplo simples do uso desses parâmetros, bem como apresenta a classe ElevatedButton:

Figura 3.14 - *Usando ElevatedButton*



Fonte: Elaborado pelos autores.

3.7 Center e Icon

O widget **Center** é uma forma fácil de centralizar um widget em relação a um widget externo. Já o widget **Icon** é uma forma simples para se trabalhar com ícones específicos.

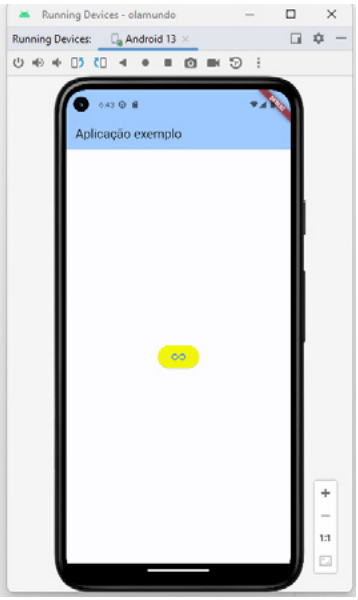
Para utilizar o widget **Center** não há segredo, basta colocar o widget **Center** como *child* do widget externo e colocar em seu *child* o widget interno, assim o widget interno ficará centralizado em relação ao widget externo.

Para uso do widget **Icon** é necessário saber que ícone se deseja exibir e esse será o primeiro parâmetro na construção do **Icon** (no exemplo da Figura 3.15, *Icons.all_inclusive*). Depois do ícone é possível definir outros parâmetros opcionais como cor (*color*), tamanho (*size*), etc.

Na Figura 3.15 tem-se o uso dos widgets **Center** e **Icon**, bem como do widget **ElevatedButton**:

Figura 3.15 - *Usando Center e Icon*

```
body() {
  return Container(
    margin: EdgeInsets.all(16),
    child: Center(
      child: ElevatedButton(
        onPressed: (){
          print("Clicou no botão");
        },
        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.yellowAccent,
        ),
        child: const Icon(
          Icons.all_inclusive,
          color: Colors.blueAccent,
        ),  // Icon
      )  // ElevatedButton
    )  // Center
  );  // Container
 }
}
```



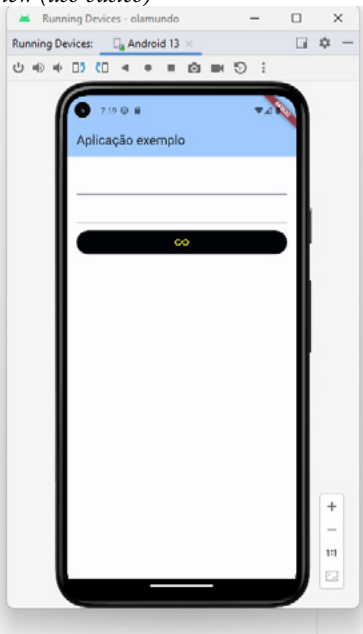
Fonte: Elaborado pelos autores.

3.8 ListView

O ListView é um widget que pode receber uma lista de outros widgets e os exibe em formato de listagem (MIOLA, 2020). Dessa forma ao invés de ter um parâmetro *child* possui um parâmetro *children* que é um array de widgets a serem exibidos no formato de listagem. A Figura 3.16 ilustra o uso básico do ListView:

Figura 3.16 - *Usando ListView (uso básico)*

```
body() {
  return Container(
    margin: EdgeInsets.all(16),
    child: ListView(
      children: [
        TextFormField(),
        TextFormField(),
        SizedBox(height: 10),
        ElevatedButton(
            onPressed: (){
              print("Clicou no botão");
            },
            style: ElevatedButton.styleFrom(
              foregroundColor: Colors.yellow,
              backgroundColor: Colors.black,
            ),
            child: Icon(Icons.all_inclusive)
          ) // ElevatedButton
      ],
    ) // ListView
  ); // Container
 }
}
```



Fonte: Elaborado pelos autores.

O widget **ListView** possibilita listagem dinâmicas de widgets através de *ListView.builder*. Isso é fundamental quando são utilizadas listas de objetos também definidos de forma dinâmica (vindos de banco de dados, por exemplo).

O *ListView.builder* possui um parâmetro *itemCount* que deve informar quantos widgets serão "desenhados". Já o parâmetro *itemBuilder* define a função que deve construir cada um desses widgets sendo *index* o índice do item corrente. Ou seja, é como se existisse um *"for"* de 0 a menor que *itemCount* chamando a função definida em *itemBuilder* para construir cada um dos widgets que a **ListView** deve exibir.

A Figura 3.17 apresenta um exemplo de uso do *ListView.builder*. Nesse exemplo é criado um array com 3 (três) nomes (Pedro, Maria e João), dessa forma esse array (palavras) possui *length* igual a 3 (três). Em *itemCount* o valor atribuído é 3 (três), ou seja, para *ListView.builder*

devem ser construídos 3 (três) widgets. A função definida em *itemBuilder* é então acionada 3 (três) vezes e, cada uma dessas vezes, cria um widget **Text** com o conteúdo de um dos nomes definidos em *palavras* de acordo com seu índice (*index*).

Figura 3.17 - *Usando ListView.builder*

```
body() {
  List<String> palavras = ["Pedro", "Maria", "João"];
  return Container(
    margin: EdgeInsets.all(16),
    child: ListView.builder(
      itemCount: palavras.length,
      itemBuilder: (context,index){
        return Text(
          "${palavras[index]}",
          style: const TextStyle(
            fontSize: 30,
            fontWeight: FontWeight.bold,
          ),  // TextStyle
        );  // Text
      },
    ),  // ListView.builder
  );  // Container
 }
}
```

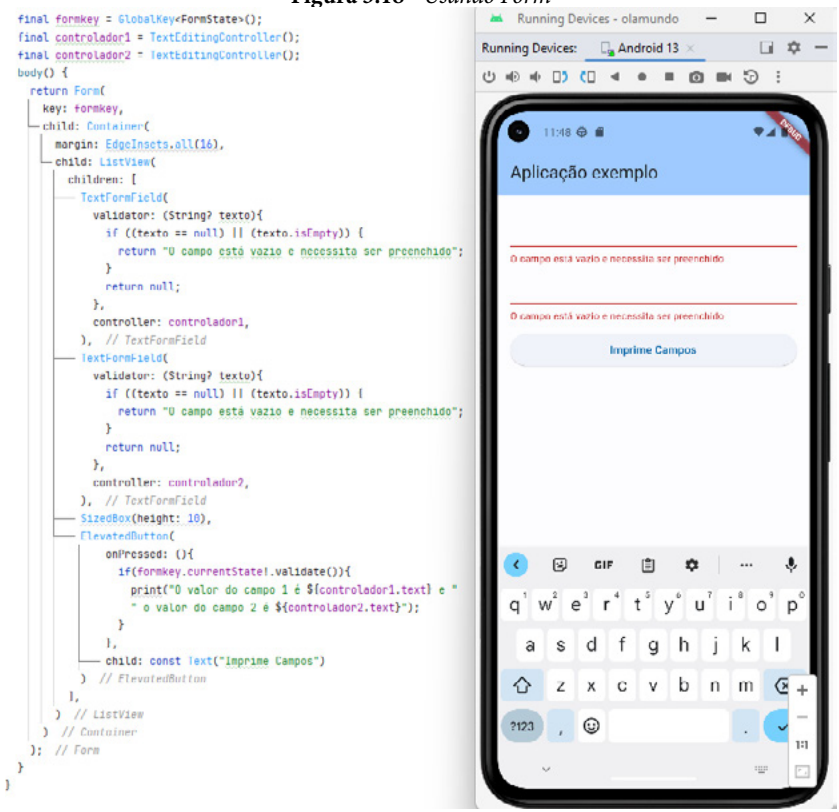
Fonte: Elaborado pelos autores.

3.9 Form

O widget **Form** é um widget de formulários do Flutter. Nele é possível validar os **TextFormField** internos através do método *validate()*. Cada **TextFormField** precisará definir uma função de validação para o parâmetro *validator*. O *validate()* passará pelos **TextFormField** do formulário, acionando a função definida no parâmetro *validator* de cada um deles.

Outra questão importante é a recuperação dos valores digitados nos **TextFormField**. Isso pode ser feito através dos controladores colocados no atributo *controller* de cada um dos **TextFormField** do **Form**. A Figura 3.18 apresenta o uso do **Form**:

Figura 3.18 - *Usando Form*



Fonte: Elaborado pelos autores.

No exemplo da Figura 3.18 ocorre a definição de 3 (três) atributos da classe **_MyHomePageState** (essa é a classe possuidora do método *body()* – ver Figura 3.1). São eles:

formkey: é uma chave que identifica de forma única o **Form**. É ela que vai permitir a validação dos **TextFormField** do **Form**. A chamada do *formkey.currentState!.validate()* vai varrer as funções atribuídas ao parâmetro *validator* de cada um dos **TextFormField** dentro do formulário. Se houver problema em algum *validator*, o **TextFormField** respectivo irá exibir a mensagem gerada pela função definida. Quando um *validator* é chamado e não há problema ele simplesmente retorna *null*. Se todos os *validator* retornarem *null* então *formkey.currentState!.*

validate() retorna *true* e, nesse caso, o print mostrando os conteúdos de *controlador1.text* e *controlador2.text* será executado.

controlador1: é o controlador associado ao primeiro **TextFormField** do **Form**. Esse tipo de controlador é necessário tanto para capturar o valor digitado pelo usuário no **TextFormField** quanto para atribuir um valor ao respectivo **TextFormField**. A estrutura hierárquica dos códigos feitos em Flutter direciona o programador a ter objetos de controle (não visuais) associados a classes visuais (widgets), isso colabora para uma melhor delegação de responsabilidades.

controlador2: é o controlador associado ao segundo **TextFormField** do **Form**.

Uma outra questão importante é a definição do parâmetro associado a função de validação que deve ser atribuída a um *validator*. Um *validator* espera/obriga uma função que passe como parâmetro uma **String? texto**. Isso significa que **texto** pode ser uma String ou nulo (*null*). Entretanto se *texto* for *null* ocorreria um erro pois **texto.isEmpty** seria inválido, uma vez que *null* não possui um atributo *isEmpty*. Para questões como essa o Dart/Flutter possui o *null safety*, ou seja, essas tecnologias buscam evitar que situações como essa aconteçam alertando o usuário e ajudando-o na proteção de seu código. Especificamente no exemplo tratado ocorre a validação *texto == null* antes de *texto.isEmpty*, dessa forma se *texto* for nulo ele já executa o conteúdo do *if* sem fazer a segunda comparação (isso porque os dois comandos estão ligados por um ||, operador *ou*).

Ainda tratando da questão de *null safety* tem-se a '!' em *formkey.currentState!.validate()*. Isso ocorre porque *currentState* não pode ser nulo para a chamada do *validate()*.

3.10 TextFormField Customizado

Uma das características interessantes que existe no Flutter/Dart é a possibilidade de criação de widgets customizados feitos a partir dos widgets disponibilizados na tecnologia.

No exemplo da Figura 3.18 foi possível verificar que a função de validação de ambos TextFormField eram iguais, o que tem total sentido, pois para ambos os casos a validação apenas verificava se algo havia sido digitado no respectivo TextFormField. Esse comportamento mais genérico poderia já estar padronizado em um widget customizado, assim o programador só precisaria implementar a validação quando realmente fosse necessário, ou seja, quando fosse algo a mais que verificar apenas se o TextFormField possui ou não um valor.

Na Figura 3.19 tem-se o widget CampoEdicao que é basicamente uma customização de um TextFormField. É importante notar que CampoEdicao não herda diretamente de TextFormField, ele herda de StatelessWidget, isso facilita a implementação e gera maior flexibilidade pois a dependência de TextFormField ocorre, de fato, apenas no método *build()* que efetivamente "desenha" o widget.

O primeiro passo ao definir um widget customizado é definir o que será disponibilizado para ser passado como parâmetro, o que ficará fixo no código e não poderá ser alterado e o que possui um comportamento padrão mas em caso de necessidade pode ser redefinido.

Para CampoEdicao a função de validação (do parâmetro *validator* do TextFormField e *validador* em CampoEdicao), caso não tenha sido definida, terá o comportamento de verificar se o campo está ou não preenchido.

CampoEdicao também irá passar o texto de exibição do TextFormField (*texto_label* = *labelText*), o texto da dica (*texto_dica* = *hintText*) e se o campo deve ou não ocultar os caracteres (*password* = *obscureText*), além do controlador que deve ser associado ao TextFormField (*controlador* = *controller*) para obter ou atribuir valores ao TextFormField. O teclado será definido como o padrão para texto.

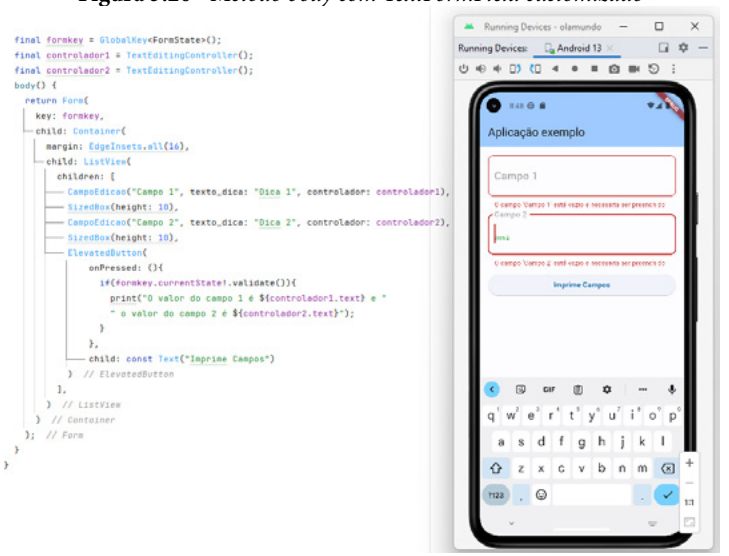
Figura 3.19 - *TextFormField Customizado*

```
import 'package:flutter/material.dart';

class CampoEdicao extends StatelessWidget{
  String texto_label;  String texto_dica; bool password;
  TextEditingController? controlador;
  FormFieldValidator<String>? validador;
  TextInputType teclado;

  String? validarCampoEdicao(String? text){
    if((text == null) || (text.isEmpty)){
      return "O campo '$texto_label' está vazio e necessita ser preenchido";
    }
    return null;
  }

  CampoEdicao(this.texto_label,
    { // Argumentos nomeados e opcionais
     this.texto_dica = "",    this.password = false,
     this.controlador, this.validador,
     this.teclado = TextInputType.text,
    }){ // Corpo do construtor
    if(validador == null){
      validador = validarCampoEdicao;
    }
  }

  Widget build(BuildContext context){
    return TextFormField(
      validator: validador,     obscureText: password,
      controller: controlador,  keyboardType: teclado,
      style: const TextStyle(fontSize: 25, color: Colors.black),
      decoration: InputDecoration(
        border: OutlineInputBorder(borderRadius: BorderRadius.circular(10)),
        labelText: texto_label,
        labelStyle: const TextStyle(fontSize: 20, color: Colors.grey),
        hintText: texto_dica,
        hintStyle: const TextStyle(fontSize: 10, color: Colors.green),
      ),  // InputDecoration
    );  // TextFormField
  }
}
```

Fonte: Elaborado pelos autores.

Utilizando o widget **CampoEdicao** ao invés do widget **TextFormField** padrão tem-se um código mais simplificado que pode ser visualizado na Figura 3.20:

Figura 3.20 - *Método body com TextFormField customizado*



Fonte: Elaborado pelos autores.

O uso de um widget **TextFormField** customizado encapsulou comportamentos e configurações de design no **CampoEdicao** permitindo maior reaproveitamento dessa codificação encapsulada.

Um widget bem simples que não havia sido discutido até o momento é o **SizedBox**. Basicamente ele é um widget usado para "dar um espaçamento" entre outros 2 (dois) widgets. Ele é muito usado em listagens para não deixar os widgets muito próximos.

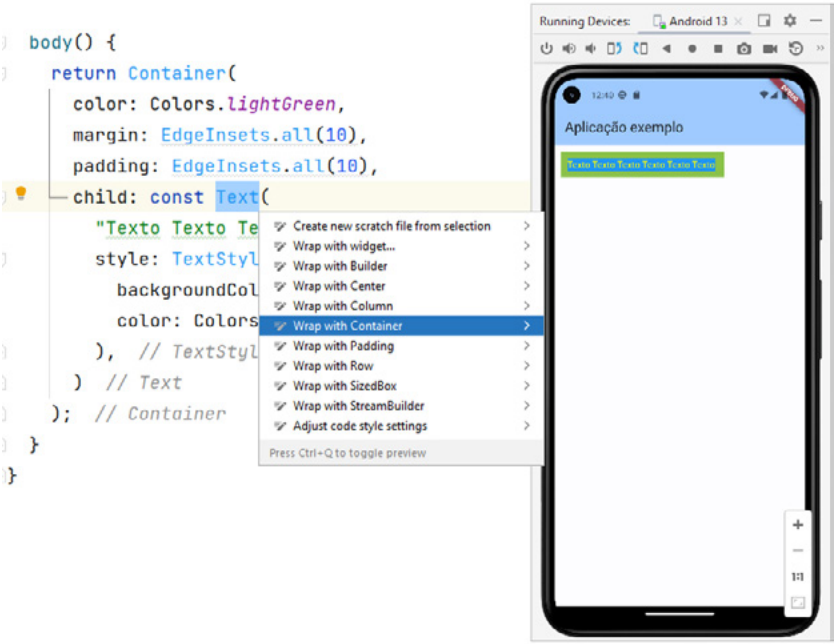
A customização de widgets possui uma outra vantagem: O encapsulamento de widgets padrão que eventualmente podem ficar obsoletos. Por exemplo: O **RaisedButton** era uma das principais classes de botão do Flutter/Dart mas foi substituído pelo **ElevatedButton**. Se o programador tivesse encapsulado, previamente, o **RaisedButton** num widget customizado, poderia modificar apenas o método *build()* desse widget customizado, trocando o **RaisedButton** pelo **ElevatedButton**, e deixar todo código que depende desse botão customizado intacto.

4 COMPONENTES VISUAIS DE AGRUPAMENTO

Neste tópico serão discutidos widgets que possuem a função de organizar outros widgets visualmente na tela, seja na forma de uma linha (widget Row), na forma de uma coluna (widget Column), empilhando-os (widget Stack) ou de forma rolável (widget PageView). Para tanto a estratégia de continuar editando o método *body()* (ver Figura 3.1) se mantém, ou seja, todos os códigos a serem apresentados continuarão na estratégia de mostrar o conteúdo do método *body()*, para cada um dos exemplos, e o resultado da codificação no emulador do Android (versão 13).

O uso de widgets de padrão de disposição na tela possibilita a implementação de aplicativos mais complexos. Um atalho muito importante do Android Studio, nesse contexto, é o *Alt+Enter*. Com *Alt+Enter* é possível encapsular um widget com outro, que será seu widget externo, e isso pode ser feito para qualquer widget da hierarquia de widgets do aplicativo. A Figura 4.1 mostra a chamada o uso do atalho *Alt+Enter* para um widget Text:

Figura 4.1 - *Usando o atalho Alt+Enter*



Fonte: Elaborado pelos autores.

Após a escolha do widget Container como widget "pai" do widget Text, na Figura 4.1, é possível estilizá-lo normalmente. É importante notar que a hierarquia *Container child Text* vai ser modificada para *Container child Container child Text*. A Figura 4.2 apresenta o novo Container "inserido" entre o Container e o Text do exemplo original da Figura 4.1, nesse exemplo o ajuste da hierarquia de widgets foi feito de forma automática, bastando ao programador acrescentar as linhas e edição do novo Container, nesse caso as linhas que definem a cor (*Colors.deepOrange*), a margem (*EdgeInserts.all(10)*) e o padding (*EdgeInserts.all(10)*) desse Container.

Figura 4.2 - *Inserindo Container entre outro Container e Text*

```
body() {
  return Container(
    color: Colors.lightGreen,
    margin: EdgeInsets.all(10),
    padding: EdgeInsets.all(10),
    child: Container(
      color: Colors.deepOrange,
      margin: EdgeInsets.all(10),
      padding: EdgeInsets.all(10),
      child: const Text(
        "Texto Texto Texto Texto Texto Texto ",
        style: TextStyle(
          backgroundColor: Colors.blue,
          color: Colors.yellowAccent
        ),  // TextStyle
      ),  // Text
    )  // Container
  );  // Container
}
}
```

Fonte: Elaborado pelos autores.

4.1 Row

O primeiro widget aqui apresentado é o Row. O widget Row exibe seus filhos (*children*) de forma horizontal, ou seja, um após o outro no eixo horizontal. Da mesma forma que o ListView, ao invés de ter o parâmetro *child* possui o parâmetro *children*, pois pode possuir vários filhos.

O parâmetro *mainAxisAlignment* define o alinhamento horizontal (que é o eixo principal de uma linha). Já o parâmetro *crossAxisAlignment* define o alinhamento vertical na linha (que é o eixo auxiliar ou complementar de uma linha).

O *mainAxisAlignment* pode ser definido como: no início (*start*), no fim (*end*), centralizado (*center*), com espaçamento uniforme entre os filhos (*spaceBetween*), com espaçamento em volta dos filhos uniformemente distribuídos (*spaceAround*) e com espaçamento uniforme entre os filhos e as bordas (*spaceEvenly*). No *spaceBetween* o primeiro e último

filhos estão "colados" nas bordas da linha, o *spaceAround* busca centralizar os filhos e espaçá-los de forma uniforme, já o *spaceEvenly* busca garantir o mesmo espaçamento entre cada um dos filhos e as bordas da linha.

O *crossAxisAlignment* pode ser definido como: no início (*start*), no fim (*end*), centralizado (*center*), esticado (*stretch*) ou em linha de base (*baseline*).

As três figuras seguintes, Figura 4.3, Figura 4.4 e Figura 4.5, mostram o uso de possibilidades de alinhamento de uma linha:

Figura 4.3 - *spaceBetween e end*

```
body() {
  return Container(
    color: Colors.yellow,
    height: 50,
    child: const Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      crossAxisAlignment: CrossAxisAlignment.end,
      children: [
        Text("Texto 1"),
        Text("Texto 2"),
        Text("Texto 3"),
      ],
    ),  // Row
  );  // Container
}
}
```

Fonte: Elaborado pelos autores.

Figura 4.4 - *spaceAround e center*

```
body() {
  return Container(
    color: Colors.yellow,
    height: 50,
    child: const Row(
      mainAxisAlignment: MainAxisAlignment.spaceAround,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: [
        Text("Texto 1"),
        Text("Texto 2"),
        Text("Texto 3"),
      ],
    ),  // Row
  );  // Container
 }
}
```

Fonte: Elaborado pelos autores.

Figura 4.5 - *spaceEvenly e start*

```
body() {
  return Container(
    color: Colors.yellow,
    height: 50,
    child: const Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        Text("Texto 1"),
        Text("Texto 2"),
        Text("Texto 3"),
      ],
    ),  // Row
  );  // Container
 }
}
```

Fonte: Elaborado pelos autores.

4.2 SingleChildScrollView

Caso sejam colocados mais widgets do que a Row é capaz de suportar o Flutter mostrará um erro. A Figura 4.6 mostra os Text de uma Row "estourarem" o espaço disponível gerando o erro (faixa em preto e amarelo):

Figura 4.6 - *Erro de invasão do espaço disponível*

```
body() {
  return Container(
    color: Colors.yellow,
    height: 50,
    child: const Row(
      mainAxisAlignment: MainAxisAlignment.end,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: [
        Text("Texto1 Texto11 Texto111 Texto1111"),
        Text("Texto2 Texto22 Texto222 Texto2222"),
        Text("Texto3 Texto33 Texto333 Texto3333"),
      ],
    ),  // Row
  );  // Container
 }
}
```

Fonte: Elaborado pelos autores.

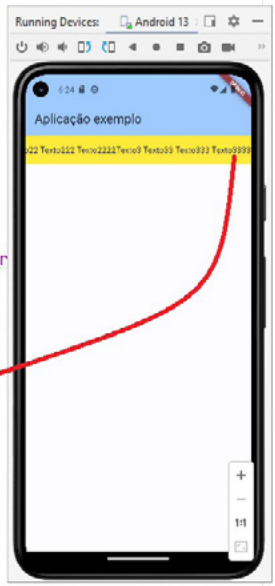
Para resolver o problema de invasão do espaço disponível pode-se utilizar um **SingleChildScrollView**. O widget **SingleChildScrollView** pode colocar um scroll vertical ou horizontal para permitir que seja possível fazer a navegação pelos widgets.

Para acrescentar esse widget deve-se selecionar o widget **Row** e o atalho *Alt+Enter*, na sequência a opção *"Wrap with widget..."* deve ser selecionada, o nome do widget (**SingleChildScrollView**) deve ser então digitado e, por fim, o parâmetro *scrollDirection* deve ser definido para horizontal ou vertical. A Figura 4.7 mostra o conteúdo do último

Text (o conteúdo inicial do primeiro **Text** já não aparece pois foi feito *scroll* nele).

Figura 4.7 - *Usando SingleChildScrollView*

```
body() {
  return Container(
    color: Colors.yellow,
    height: 50,
    child: SingleChildScrollView(
      scrollDirection: Axis.horizontal,
      child: const Row(
        mainAxisAlignment: MainAxisAlignment.end,
        crossAxisAlignment: CrossAxisAlignment.center
        children: [
          Text("Texto1 Texto11 Texto111 Texto1111"),
          Text("Texto2 Texto22 Texto222 Texto2222"),
          Text("Texto3 Texto33 Texto333 Texto3333"),
        ],
      ),  // Row
    ),  // SingleChildScrollView
  );  // Container
 }
}
```

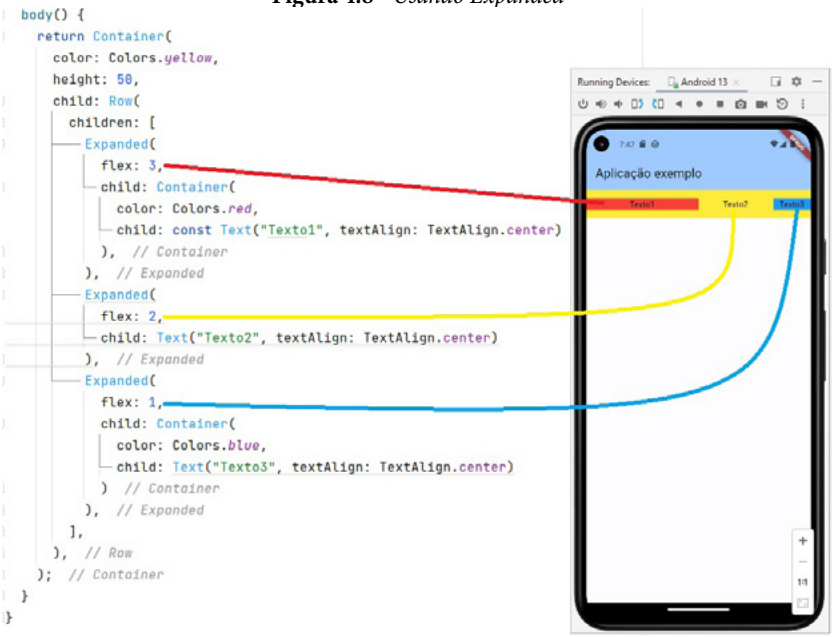


Fonte: Elaborado pelos autores.

4.3 Expanded

Para dividir o espaço disponível na **Row** é possível utilizar o widget **Expanded**. Com esse widget os espaços são divididos de forma proporcional aos pesos (parâmetro *flex*). Esse esquema de dividir o espaço disponível por pesos já é bem tradicional no Android Nativo e foi trazido para o Flutter/Dart. Lecheta (2016, p. 124) apresenta o conceito de pesos (*android:layout_weight*) para um **LinearLayout** que é algo similar a uma **Row**/**Column** no Flutter/Dart. Na Figura 4.8 tem-se o uso de pesos:

Figura 4.8 - *Usando Expanded*



Fonte: Elaborado pelos autores.

O widget **Expanded** também pode ser utilizado para obter espaço que não necessita ser utilizado por outros widgets. Na Figura 4.9, caso não fosse utilizado o widget **Expanded** ocorreria "estouro" do espaço disponível, gerando erro (faixa em preto e amarelo):

Figura 4.9 - *Expanded para evitar erro*

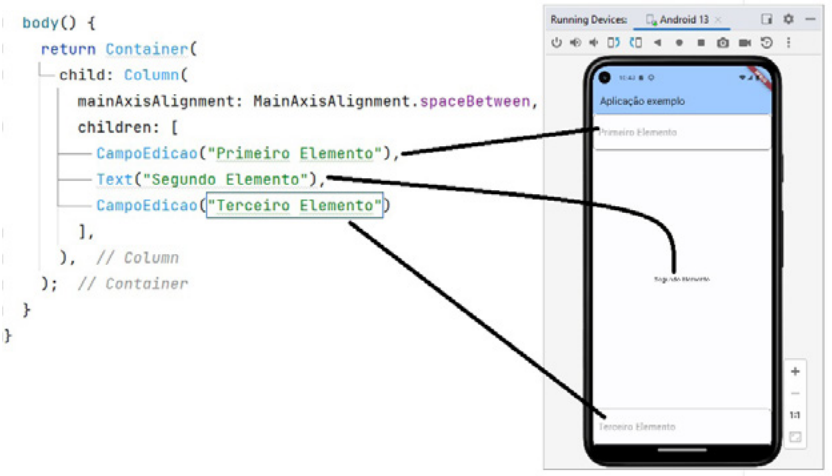
```
body() {
  return Container(
    color: Colors.yellow,
    height: 50,
    child: Row(
      children: [
        Container(
          color: Colors.red,
          child: const Text("Texto1", textAlign: TextAlign.center)
        ),  // Container
        Expanded(
          child: Text("TextoA TextoB TextoC TextoD "
                      "TextoE TextoF TextoG TextoH "
                      "TextoI TextoJ TextoK TextoL ",
                      textAlign: TextAlign.center)  // Text
        ),  // Expanded
        Container(
          color: Colors.blue,
          child: Text("Texto3", textAlign: TextAlign.center)
        ),  // Container
      ],
    ),  // Row
  );  // Container
  }
}
```

Fonte: Elaborado pelos autores.

4.4 Column

É um widget que exibe seus filhos em uma matriz vertical. *mainAxisAlignment*, para Column, define o alinhamento vertical. Já o *crossAxisAlignment* define o alinhamento horizontal (notar que é ao contrário do widget Row).

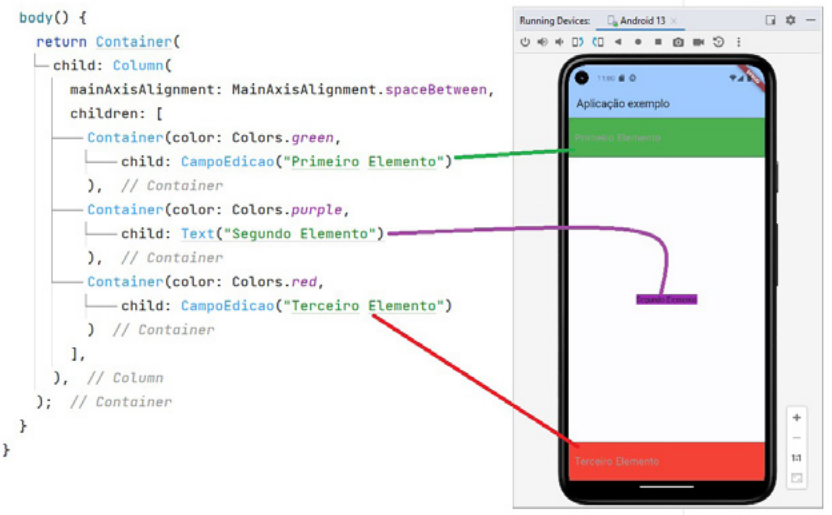
Figura 4.10 - *Usando o widget Column*



Fonte: Elaborado pelos autores.

Na Figura 4.10 tem-se a sensação que o widget **Text**("Segundo Elemento") ocupou todo o espaço disponível, mas o exemplo da Figura 4.11 vai demonstrar que essa sensação não é um fato.

Figura 4.11 - *Column mostrando o espaço ocupado pelos widgets internos*

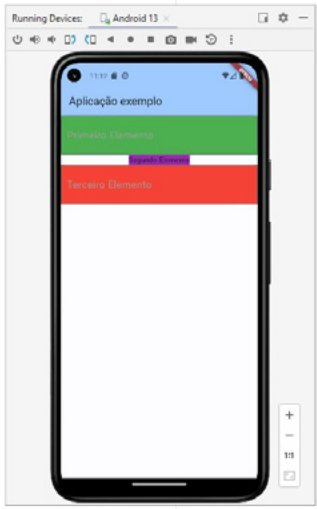


Fonte: Elaborado pelos autores.

No exemplo da Figura 4.11 é possível verificar que o **Text** ocupa apenas o espaço necessário para a sua exibição, isso ocorre porque foi usado *spaceBetween* no Column e isso distribuiu seus elementos na vertical, colocando o primeiro e o último nos extremos e o segundo elemento no meio. Trocando o *spaceBetween* por *start* fica ainda mais claro que os widgets ocupam apenas o espaço necessário para a sua exibição, a Figura 4.12 ilustra essa mudança:

Figura 4.12 - *Trocando spaceBetween por start*

```
body() {
  return Container(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.start,
      children: [
        Container(color: Colors.green,
          child: CampoEdicao("Primeiro Elemento")
        ),  // Container
        Container(color: Colors.purple,
          child: Text("Segundo Elemento")
        ),  // Container
        Container(color: Colors.red,
          child: CampoEdicao("Terceiro Elemento")
        )  // Container
      ],
    ),  // Column
  );  // Container
 }
}
```



Fonte: Elaborado pelos autores.

4.5 Column, Row e Expanded juntos

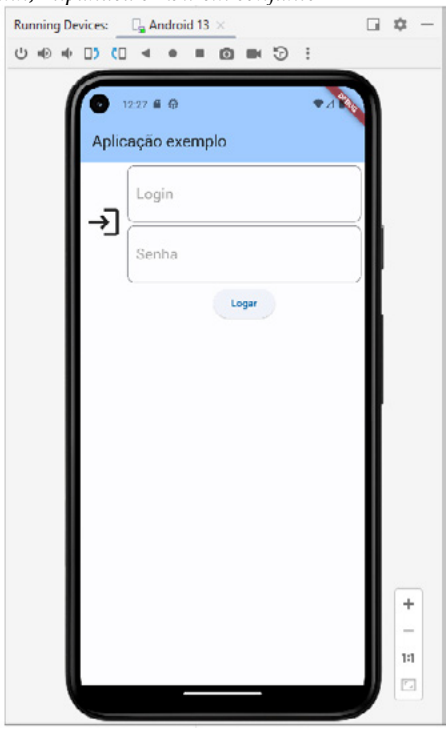
Elaborar layouts complexos em aplicativos não é uma tarefa trivial, isso exige que sejam utilizados widgets que organizem visualmente outros widgets, como é o caso do Column, do Row e do Expanded.

O uso do widget Expanded ajuda a deixar os layouts responsivos pois possibilita que os widgets possam ser distribuídos na tela de acordo com os pesos previamente estabelecidos (parâmetro *flex*). Nesse contexto, é possível prototipar uma tela e buscar organizá-la e implementá-la baseado em widgets que organizam visualmente outros

widgets. Na Figura 4.13 tem-se o uso dos widgets Row, Column e Expanded em conjunto para "desenhar" uma tela de login com um ícone:

Figura 4.13 - *Usando Column, Expanded e Row em conjunto*

```
body() {
  return Row(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Expanded(
        flex: 1,
        child: Container(
          margin: const EdgeInsets.only(top: 50),
          alignment: Alignment.topCenter,
          child: const Icon(Icons.login, size: 50),
        ), // Container
      ), // Expanded
      Expanded(
        flex: 5,
        child: Column(
          children: [
            const SizedBox(height: 5),
            CampoEdicao("Login"),
            const SizedBox(height: 5),
            CampoEdicao("Senha"),
            const SizedBox(height: 5),
            ElevatedButton(
                onPressed: (){
                  print("Logando...");
                },
                child: const Text("Logar")
            ) // ElevatedButton
          ],
        ), // Column
      ), // Expanded
      const SizedBox(width: 5)
    ],
  ); // Row
}
```



Fonte: Elaborado pelos autores.

No exemplo da Figura 4.13 o widget mais externo é um **Row**, ou seja, o espaço da tela será dividido em elementos internos da **Row**.

O parâmetro *crossAxisAlignment*, da **Row**, é responsável por fazer o alinhamento vertical. Como ele foi definido para no início (*start*), os widgets internos da **Row** serão alinhados verticalmente no início da **Row** (na prática, no início/topo da tela).

Por sua vez o parâmetro *children* define os filhos da **Row** que, nesse caso, serão 2 (dois) **Expanded** que dividirão o espaço da Row na proporção de 1 (um) para 5(cinco) (na prática, dividirão o espaço da tela

nessa proporção, visto que a **Row** ocupa todo o espaço da tela). O parâmetro *flex* de cada um dos dois **Expanded** é quem define a proporção de 1 (um) para 5 (cinco).

No primeiro **Expanded**, que ocupa 1 (um) de 6 (seis) espaços/pesos, tem-se um **Container** que foi definido para ajustar a margem e o alinhamento para o **Icon** que será visualizado. Já o parâmetro *size*, do **Icon**, define seu tamanho e pode ser modificado de acordo com a necessidade do programador.

No segundo **Expanded**, que ocupa 5 (cinco) de 6 (seis) espaços/pesos, tem-se um **Column** que organiza 2 (dois) **CampoEdicao** (widget customizado apresentado anteriormente na Figura 3.19) e 1 (um) **ElevatedButton**. O padrão visual desses widgets poderia ter sido mais elaborado, mas para não deixar o exemplo excessivamente grande optou-se por uma apresentação mais simples.

Os **SizedBox** utilizados em todo o exemplo servem apenas para colocar um pequeno espaço entre widgets e bordas da tela.

O Flutter solicita, sempre que possível, o uso da palavra *const* para otimizar a criação de widgets "constantes", isso deixa a reconstrução visual da tela mais rápida pois evita que o Flutter reconstrua objetos desnecessariamente.

4.6 Stack

O widget Stack possibilita colocar widgets um sobre o outro. Esse widget é útil para sobrepor vários filhos de uma maneira simples, por exemplo, um texto sobre uma imagem.

Para trabalhar com imagens uma das formas é utilizando a pasta *assets* (essa é uma pasta padrão para armazenamento e obtenção de imagens num aplicativo). Para utilizar a pasta *assets* é necessário primeiramente criá-la no projeto. Para tanto basta clicar com o botão direito do mouse, no projeto criado no Android Studio, em *New->Directory* e definir o nome do diretório como *assets*. Em seguida basta colocar a imagem que se deseja utilizar na pasta *assets* e habilitar a pasta *assets* no arquivo *pubspec.yaml*.

A Figura 4.14 apresenta o código antes da habilitação da pasta *assets* no arquivo *pubspec.ya*ml e a Figura 4.15 apresenta o código depois da habilitação da pasta *assets* no arquivo *pubspec.ya*ml.

Figura 4.14 - *Antes de configurar a pasta assets*

```
# The following section is specific to Flutter packages.
flutter:

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true

  # To add assets to your application, add an assets section, like this:
  # assets:
  #   - images/a_dot_burr.jpeg
  #   - images/a_dot_ham.jpeg
```

Fonte: Elaborado pelos autores.

Figura 4.15 - *Depois de configurar a pasta assets*

```
# The following section is specific to Flutter packages.
flutter:

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true

  # To add assets to your application, add an assets section, like this:
  assets:
    - assets/
```

Fonte: Elaborado pelos autores.

Uma vez que a pasta *assets* já está habilitada e a imagem que será utilizada já está nessa pasta, basta utilizar o widget **Image** e importar a imagem da pasta *assets*. A Figura 4.16 apresenta um **Stack** utilizando o parâmetro *fit* com o valor *StackFit.expand* e um **Image** utilizando o parâmetro *fit* com o valor *BoxFit.fill*. *StackFit.expand* fará a **Stack**

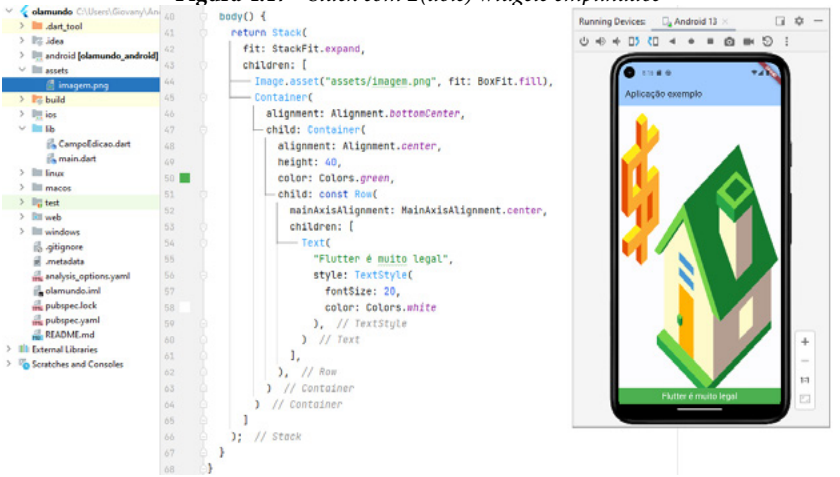
ocupar todo o espaço disponível e *BoxFit.fill* fará a **Image** ocupar todo o espaço disponibilizado pelo **Stack** mesmo que isso deforme visualmente a imagem. O primeiro parâmetro de *Image.asset* é o caminho da imagem que se deseja utilizar no **Image**.

Figura 4.16 - *Usando Image.asset*

Fonte: Elaborado pelos autores.

O próximo passo é efetivamente utilizar o potencial do widget **Stack** que é, conforme descrito anteriormente, empilhar um widget sobre outro. Nesse contexto será inserido um **Container** como um novo filho para a **Stack** (em *children*), esse **Container** ficará sobre a imagem na parte inferior da tela, possibilitando o design diferenciado. A Figura 4.17 apresenta esse código modificado e o efeito visual no emulador do Android 13:

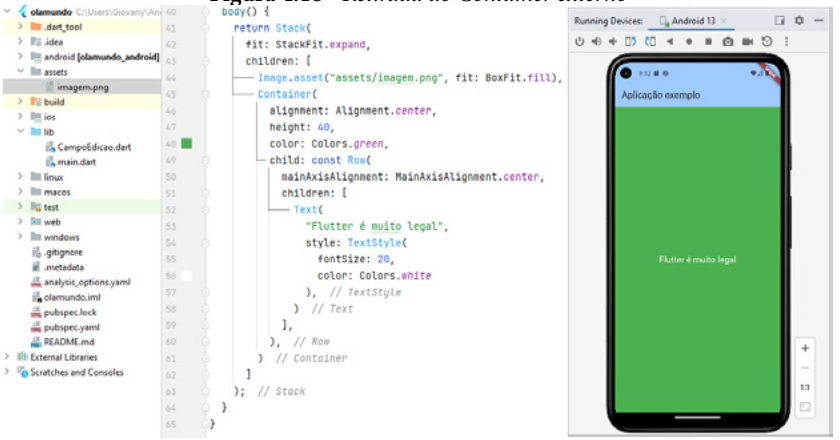
Figura 4.17 - *Stack com 2(dois) widgets empilhados*



Fonte: Elaborado pelos autores.

Uma questão importante, no exemplo da Figura 4.17, está relacionada com o **Container** mais externo (que possui *alignment: Alignment.bottomCenter*). Esse **Container** externo não possui cor (é transparente) e por isso permite a exibição da imagem que está sob ele. Se esse **Container** externo não existir, apesar Container interno ter altura limitada a 40, sua cor irá se propagar e a imagem não será exibida (o *StackFit.expand* irá expandi-lo). A Figura 4.18 mostra o efeito da retirada do **Container** externo:

Figura 4.18 - *Retirada do Container externo*



Fonte: Elaborado pelos autores.

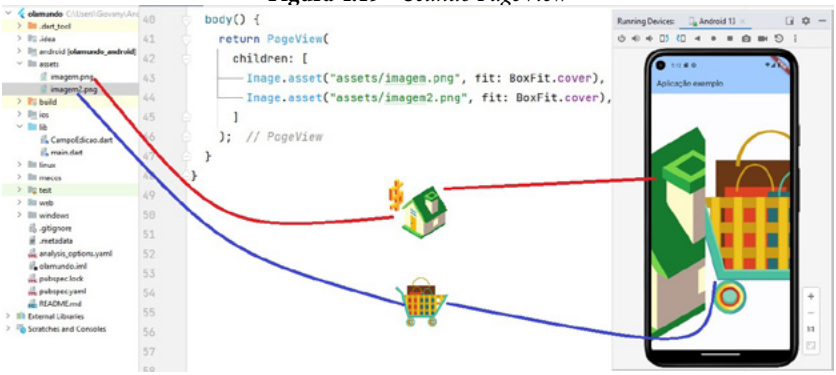
4.7 PageView

O PageView é um widget que gera páginas roláveis. O comportamento desse widget é similar a um "carrossel" e muito utilizado para exibição de imagens.

Para apresentação das imagens será utilizada a opção *BoxFit.cover*, no parâmetro *fit* das imagens (Image), para não deformá-las (essa opção expande um recorte da imagem mantendo as proporções).

A Figura 4.19 ilustra o uso do PageView para 2 (duas) imagens (*imagem.png* e *imagem2.png*). Caso fosse desejado apresentar mais imagens para rolagem bastaria inserir novos filhos no parâmetro *children* do PageView.

Figura 4.19 - *Usando PageView*



Fonte: Elaborado pelos autores.

5 EXEMPLO PRÁTICO DE DESENVOLVIMENTO DE UM APP

Neste tópico serão trabalhados mais alguns widgets mas eles serão tratados com um foco um pouco diferente. Nos tópicos de Componentes Visuais anteriores o foco estava em conhecer cada widget apresentado e os exemplos eram bem simples, até por isso a ideia de um código geral e a mudança apenas do método *body()* para a apresentação de cada widget. Neste tópico o ponto de partida é uma aplicação mais completa e os widgets serão apresentados nessa aplicação.

A aplicação completa tratada neste tópico foi desenvolvida em 2 (duas) abas com funcionalidades diferentes. A primeira aba é uma calculadora de IMC, ou seja, o usuário digita a altura e o peso e a calculadora retorna o IMC e a situação do indivíduo com os peso e altura informados, sendo que a classe que ficará responsável por essa aba é o widget customizado **TabIMC**. A segunda aba é uma calculadora de linha para operações básicas, com **TabSimplesCalc** responsável por tratar seus comportamentos.

Os códigos fontes desta aplicação podem ser encontrados em https://github.com/GiovanyFT/calculadora-imc.

O ponto de partida de uma aplicação Flutter/Dart é a chamada ao método main: *void main() => runApp(Calculadora());* e **Calculadora** é a classe principal e ponto de partida da aplicação que será discutida em todo este tópico. Na Figura 5.1 é possível ver o código da classe **Calculadora** e as importações necessárias ao seu funcionamento:

Figura 5.1 - *Classe Calculadora*

```
import 'package:calculadoraimcflutter/tabs/tab_imc.dart';
import 'package:calculadoraimcflutter/tabs/tab_simples_calc.dart';
import 'package:flutter/material.dart';

void main() => runApp(Calculadora());

class Calculadora extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primaryColor: Colors.blue,
      ),  // ThemeData
      home: HomePage(),
    );  // MaterialApp
  }
}
```

Fonte: Elaborado pelos autores.

Na classe **Calculadora** tem-se a definição de um **MaterialApp** que é o widget que organizará a aplicação e utilizará o padrão Material Design. O Material Design (MEW, 2016) é um padrão para interfaces gráficas de aplicativos definido pela Google, através dele tem-se aplicativos mais bonitos, mais harmônicos pois a combinação de cores, luz, etc. é feita de forma planejada, sistematizada.

Um dos parâmetros de **MaterialApp** é *debugShowCheckedModeBanner* que, por padrão, é definido como *true*. Esse atributo serve para mostrar/ocultar a faixa vermelha escrita *DEBUG* apresentada em aplicativos anteriores (essa faixa pode ser vista, por exemplo, na Figura 4.19 no canto superior direito). Como nessa aplicação essa opção foi definida para *false* a faixa não mais irá aparecer.

Um outro parâmetro definido para **MaterialApp** é o parâmetro *theme*. O parâmetro *theme* aplicará um tema na aplicação. Esse tema define, por exemplo, cores padrões para os widgets usados na aplicação.

Por fim o parâmetro *home* fica responsável por definir a "tela principal" da aplicação, ou seja, o widget que é ponto de partida e "primeira tela" ou "página inicial" para a aplicação. A Figura 5.2 apresenta o código do widget **HomePage**:

Figura 5.2 - *Classe HomePage*

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
      length: 2,
      child: Scaffold(
        appBar: AppBar(
          centerTitle: true,
          title: Text("Calculadora"),
          bottom: TabBar(
            tabs: <Widget>[
              Tab(
                text: "IMC",
              ),  // Tab
              Tab(text: "Simples Calc"),
            ],  // <Widget>[]
          ),  // TabBar
        ),  // AppBar
        body: TabBarView(
          children: <Widget>[
            TabIMC(),
            TabSimplesCalc(),
          ],  // <Widget>[]
        ),  // TabBarView
      ),  // Scaffold
    );  // DefaultTabController
  }
}
```

Fonte: Elaborado pelos autores.

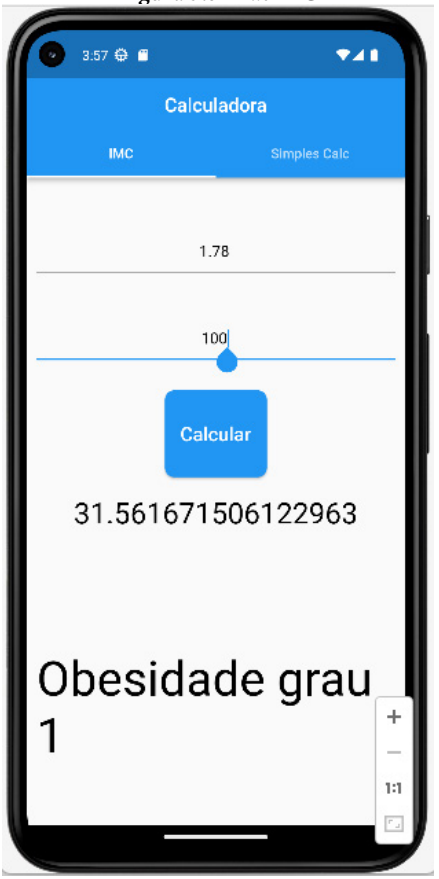
Na construção (método *build*), do widget **HomePage**, tem-se o uso do widget **DefaultTabController**. O **DefaultTabController** é uma das possibilidades de widget para se trabalhar com abas (ou tabs). Um parâmetro fundamental de um **DefaultTabController** é *length*. Esse parâmetro define quantas abas serão criadas. No *child* do **DefaultTabController** tem-se um **Scaffold** que basicamente implementa a estrutura básica de layout visual do Material Design para uma tela/janela.

No **Scaffold** verifica-se o parâmetro *appBar* que é utilizado para definir a **AppBar** da tela/janela criada a partir do **Scaffold**. Ainda no **Scaffold** tem-se o parâmetro *child* que efetivamente apresentará o conteúdo da tela/janela, nesse caso utilizou-se o **TabBarView** e cada um de seus *children* apresentará o conteúdo de uma das abas.

Para alternar entre as abas é definido no **AppBar** o **TabBar**. No caso do exemplo, cada aba será representada por um texto, a primeira "IMC" e a segunda "Simples Calc". É fundamental notar que o valor de *length*, do **DefaultTabController**, deve ser igual a quantidade de **Tab** no parâmetro *tabs* da **TabBar** que deve ser igual a quantidade de *children* no **TabBarView**. Além disso, a ordem de definição das *tabs* da **TabBar** deve ser a mesma dos *children* no **TabBarView**, ou seja, a *Tab(text: "IMC",)* se relaciona com *TabIMC()* e *Tab(text: "Simples Calc")* se relaciona com *TabSimplesCalc()*.

A Figura 5.3 mostra a primeira aba, com texto "IMC" e representada por **TabIMC**:

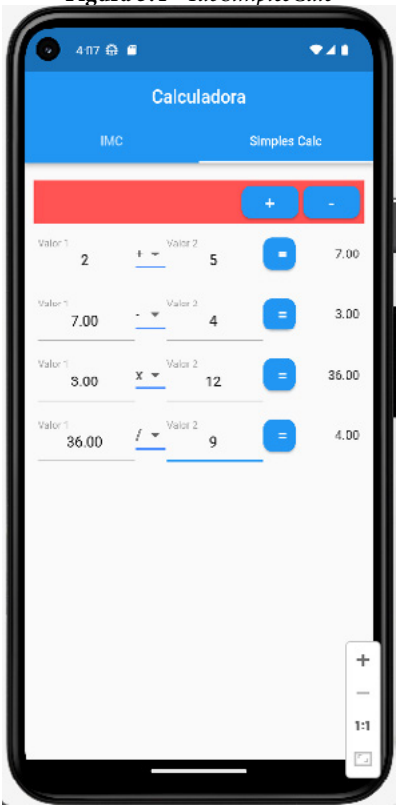
Figura 5.3 - *TabIMC*



Fonte: Elaborado pelos autores.

A Figura 5.4 mostra a segunda aba, com texto “Simples Calc” e representada por **TabSimplesCalc**:

Figura 5.4 - *TabSimplesCalc*

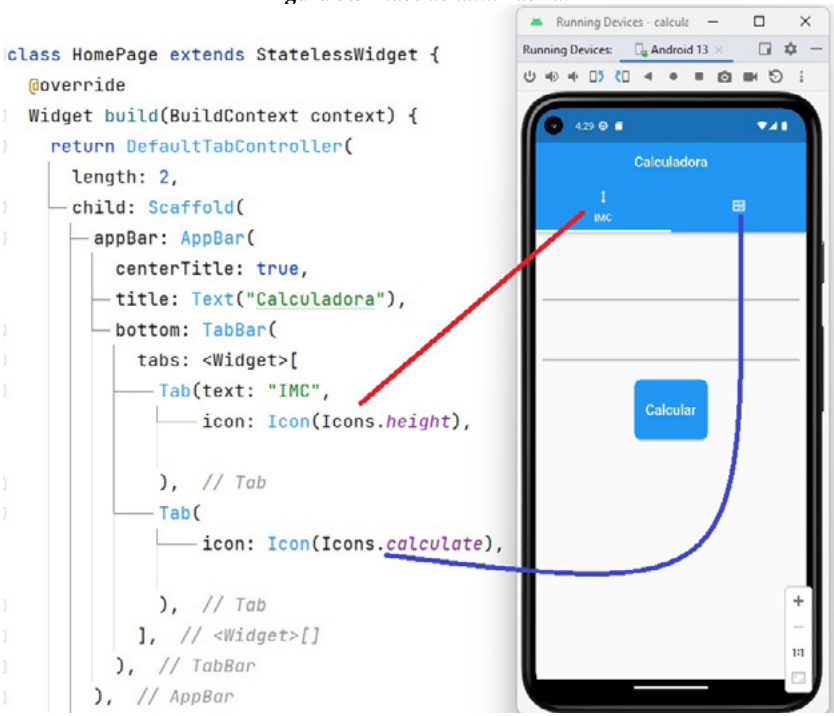


Fonte: Elaborado pelos autores.

5.1 TabBar

TabBar é um widget utilizado quando se deseja colocar Tabs (abas) nos aplicativos. No exemplo da Figura 5.2 tem-se 2 (duas) abas com navegação por menus com textos (“IMC” e “Simples Calc”), mas nada impede que a navegação utilize ícones ou textos e ícones. A Figura 5.5 apresenta o menu de “IMC” usando texto e ícone e o menu de “Simples Calc” usando apenas 1 (um) ícone:

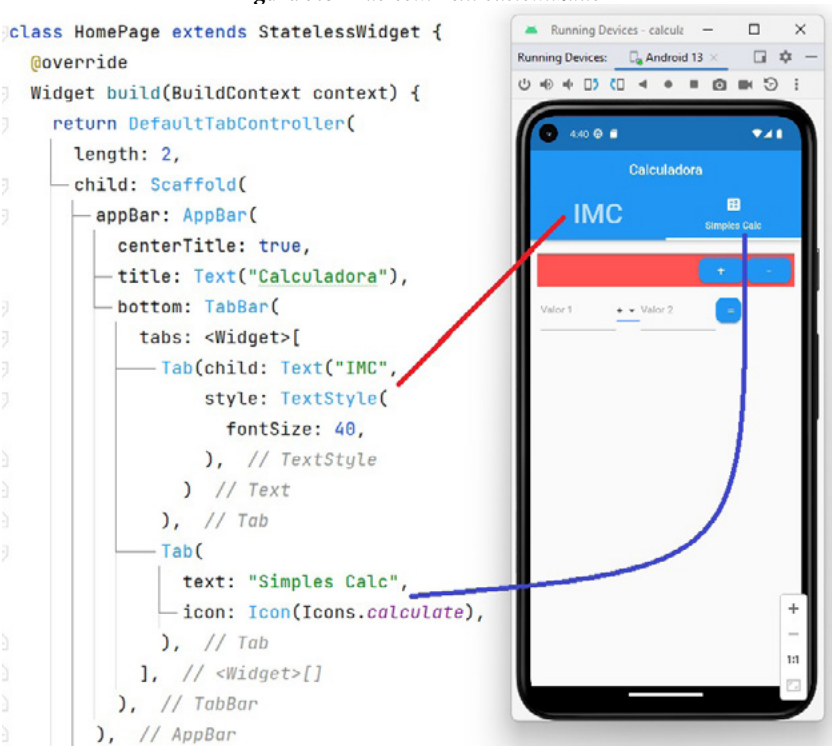
Figura 5.5 - *tabs de uma TabBar*



Fonte: Elaborado pelos autores.

Também é possível utilizar-se um widget **Tab** customizado a partir do parâmetro *child*. A Figura 5.6 apresenta o menu de "IMC" com um **Text** customizado:

Figura 5.6 - *Tab com Text customizado*



Fonte: Elaborado pelos autores.

Para trabalhar com Tabs precisamos executar 3 (três) passos (GOOGLE, 2023g):

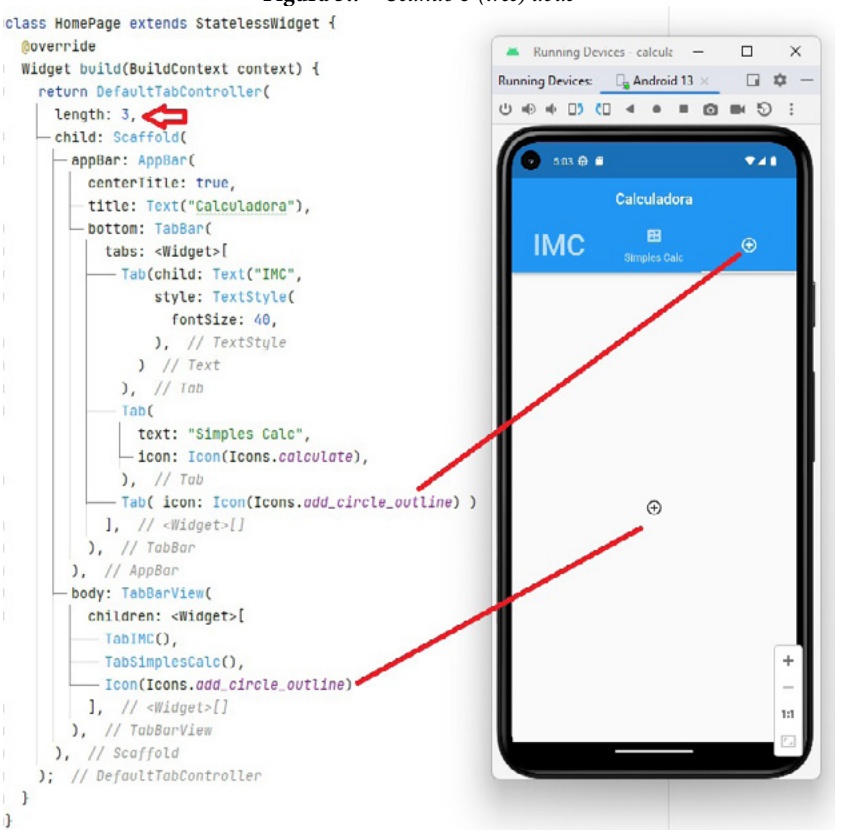
1. Criar um **TabController**: é possível criar um **TabController** personalizado mas a solução mais simples é utilizar um **DefaultTabController**;
2. Criar as tabs no **TabBar**;
3. Criar o conteúdo de cada aba.

Dessa forma se fosse necessário acrescentar uma nova aba, no exemplo apresentado, bastaria:

1. Modificar o parâmetro *length* de 2 (dois) para 3 (três);
2. Acrescentar uma nova **Tab** nas *tabs* da **TabBar**;
3. Acrescentar mais um filho no *children* do **TabBarView**.

A Figura 5.7 ilustra o exemplo apresentado refletindo essa possibilidade de mudança:

Figura 5.7 - *Usando 3 (três) abas*



Fonte: Elaborado pelos autores.

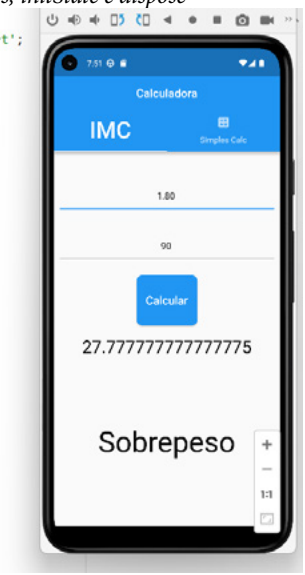
5.2 TabIMC

TabIMC é um widget customizado que representa o conteúdo da primeira aba do aplicativo. Precisa ser um StatefullWidget, isso significa que haverá (ou melhor, poderá haver) mudança de estado. Na prática isso significa que será possível chamar o método *setState*, esse método permite alteração de elementos na tela e após sua chamada a tela será redesenhada, ou seja, chamará o método *build* novamente. Na

Figura 5.8 tem-se a definição dos atributos do widget TabIMC e as definições dos métodos *initState* e *dispose*. Os atributos *controladorPeso* e *controladorAltura* serão usados para acesso aos valores digitados nos campos de edição de peso e altura, respectivamente, por sua vez o atributo *valor_imc* é o texto com o resultado da conta, referente ao peso e altura informados (valor do IMC em si), para ser exibido na tela, já o atributo *situacao_imc* será utilizado para exibir a situação referente ao valor do IMC apresentado ("Sobrepeso", por exemplo). O *initState* é um método chamado quando o widget é criado, já o método *dispose* é acionado no momento da destruição/desalocação do widget, no caso específico desse widget não foi feito qualquer comportamento adicional ao implementado em seu ancestral.

Figura 5.8 - *TabIMC declaração de atributos, initState e dispose*

```dart
import 'package:calculadoraimcflutter/widgets/botao_azul.dart';
import 'package:calculadoraimcflutter/widgets/campo_edicao_double.dart';
import 'package:flutter/material.dart';

class TabIMC extends StatefulWidget {

  @override
  _TabIMCState createState() => _TabIMCState();
}

class _TabIMCState extends State<TabIMC> {
  final controladorPeso = TextEditingController();
  final controladorAltura = TextEditingController();
  String valor_imc = "";
  String situacao_imc = "";

  @override
  void initState() {
    // TODO: implement initState
    super.initState();
  }

  @override
  void dispose() {
    // TODO: implement dispose
    super.dispose();
  }
```



Fonte: Elaborado pelos autores.

botao_azul.dart é o arquivo que contém o widget customizado **BotaoAzul**, já o arquivo *campo_edicao_double.dart* contém o widget customizado **CampoEdicaoDouble**.

O widget **BotaoAzul** utiliza um **ElevatedButton** para sua representação visual deixando fixos atributos como a cor do botão (azul) e o formato (com bordas arredondadas). Esse widget ainda atribui a cor branca para a fonte do texto do botão, apesar de permitir ao programador a customização dessa opção, bem como o tamanho da fonte do botão que por padrão vem no valor 20 (vinte) mas que também pode ser modificado.

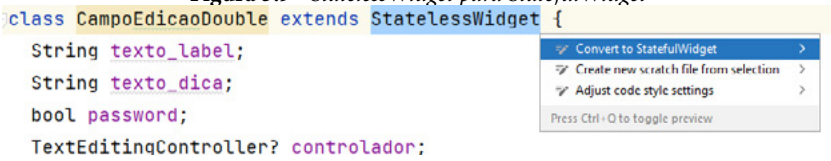
O widget **CampoEdicaoDouble** utiliza um **TextFormField** como base, sendo similar ao **CampoEdicao** apresentado na Figura 3.19. A diferença, basicamente, é que **CampoEdicaoDouble** possui uma função de validação para formulários (*validator* do **TextFormField**) que dispara uma exceção (**FormatException**) caso haja um problema no formato do número informado, ou seja, caso não seja possível transformar o que foi informado em um número real (double). Para ver o código desse widget como dos demais widgets customizados basta ir ao repositório do projeto[11].

Tanto **BotaoAzul** como **CampoEdicaoDouble** são **StatelessWidget**. Isso significa que não haverá mudança de estado.

Sempre que possível é preferível o uso de **StatelessWidget** a **StatefullWidget** pois são mais simples e possuem um custo computacional menor, uma vez que não precisam fazer o gerenciamento de estados. Entretanto nem sempre é possível utilizar um **StatelessWidget** e nesses casos (uso do *setState*, por exemplo) será necessário fazer a conversão de um widget inicialmente definido como **StatelessWidget** para que ele passe a ser um **StatefullWidget**. Essa conversão pode ser codificada manualmente, mas é um tanto quanto trabalhosa. Uma forma mais automatizada de fazer isso, no Android Studio, é através da seleção da palavra **StatelessWidget** junto da combinação de teclas *Alt+Enter*. A Figura 5.9 mostra como o widget **CampoEdicaoDouble** poderia ser transformado num **StatefullWidget** através desse procedimento:

11 https://github.com/GiovanyFT/calculadora-imc

Figura 5.9 - *StatelessWidget para StatefulWidget*



Fonte: Elaborado pelos autores.

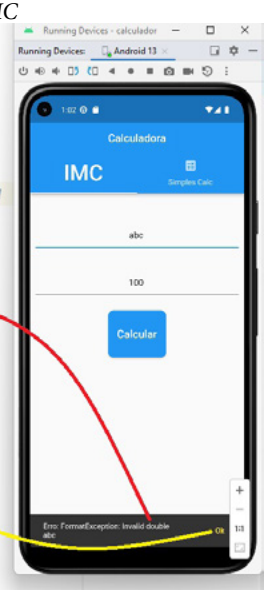
Voltando para o widget **TabIMC** em si o próximo passo é conhecer alguns métodos privados que serão utilizados no método *build* desenhar **TabIMC**.

É importante lembrar que métodos privados em Flutter/Dart são métodos que começam com o caractere '_' (*underline*). Em caso de dúvida sobre o conceito de métodos privados é recomendável a revisão do tópico sobre Encapsulamento (capítulo 2).

O primeiro método que será apresentado é o método *_calcularIMC()*. Esse método captura o peso e altura dos campos de edição e promove o cálculo do IMC (Índice de Massa Corpórea) retornando-o. Em caso de erro (o texto informado em algum dos campos de edição não ser um número, por exemplo) uma exceção é gerada (**FormatException**) e um **SnackBar** é criado para apresentar esse erro. A Figura 5.10 mostra uma entrada de dados inválida para a altura e um **SnackBar** apresentando a mensagem de erro.

Figura 5.10 - *Método _calcularIMC*

```
double? _calcularIMC() {
  try {
    double altura = double.parse(controladorAltura.text);
    double peso = double.parse(controladorPeso.text);
    double imc = peso / (altura * altura);
    return imc;
  } on FormatException catch (erro) {
    // O ScnackBar precisa de um context definido dentro do Scaffold
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text("Erro: $erro"),
        duration: Duration(seconds: 5),
        action: SnackBarAction(
          textColor: Colors.yellow,
          label: "Ok",
          onPressed: () {
            setState(() {
              _reinicializarCampos();
            });
          },
        ),  // SnackBarAction
      ),  // SnackBar
    );
  }
}
```



Fonte: Elaborado pelos autores.

O **SnackBar** é uma barra que aparece na parte inferior da tela e normalmente é utilizada para a apresentação de erros de edição. É um widget presente no Flutter de forma nativa (não é um plugin externo[12]). No **SnackBar** é possível customizar o conteúdo do erro que será apresentado (*content*), o tempo de abertura do **SnackBar** (*duration*), o texto (*label*) e cor (*textColor*) da sua **SnackBarAction** (o "botão de fechamento/ação da SnackBar") bem como a ação que será executada quando clicado (conteúdo do *onPressed*).

Especificamente no exemplo da Figura 5.10 quando o **SnackBar** é clicado (na **SnackBarAction**) os campos são reinicializados e a **TabIMC** é redesenhada (acionamento do método *build* de **TabIMC**) para refletir essa alteração. Isso só é possível por conta da chamada do método *setState* que faz com que a **TabIMC** seja redesenhada (acionamento do método *build* de **TabIMC**) para refletir uma mudança de estado, nesse caso, para mostrar que os campos de edição devem estar

12 https://pub.dev/

vazios. A Figura 5.11 mostra o método privado *_reinicializarCampos* que possui o papel de fazer a reinicialização dos campos de edição da TabIMC:

Figura 5.11 - *Método _reinicializarCampos*

```
void _reinicializarCampos() {
  controladorPeso.text = "";
  controladorAltura.text = "";
  valor_imc = "";
  situacao_imc = "";
}
}
```

Fonte: Elaborado pelos autores.

Por último, mas não menos importante, tem-se o método privado *_determinarSituacaoIMC* esse método retorna, a partir de um IMC informado, a situação do indivíduo. Esse é um método bem simples mas fundamental para a apresentação da situação do indivíduo no que se refere ao IMC. A Figura 5.12 apresenta o código referente ao método privado *_determinarSituacaoIMC*:

Figura 5.12 - *Método privado _determinarSituacaoIMC*

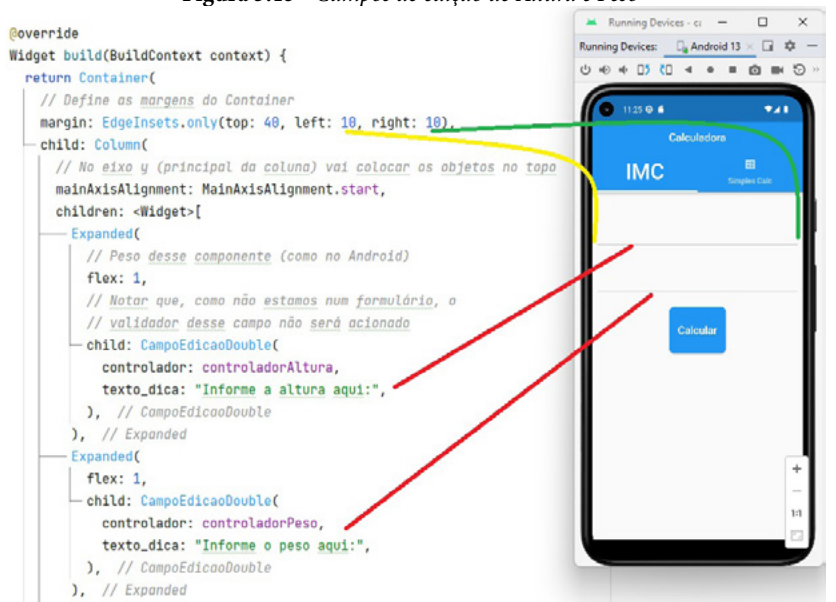
```
String _determinarSituacaoIMC(double imc) {
  if (imc < 18.5)
    return "Abaixo do peso";
  else if (imc < 24.9)
    return "Peso normal";
  else if (imc < 29.9)
    return "Sobrepeso";
  else if (imc < 34.9)
    return"Obesidade grau 1";
  else if (imc < 39.9)
    return "Obesidade grau 2";
  else
    return "Obesidade grau 3";
}
```

Fonte: Elaborado pelos autores.

O IMC (Índice de Massa Corpórea) é um parâmetro utilizado em estudos da área de saúde tendo uma relação direta com a questão do peso adequado e da melhoria na qualidade de vida (CARVALHO; S, 2014) apesar de não ser o único nem necessariamente o melhor (MELÃO JR, 2021).

Após o conhecimento dos atributos de **TabIMC**, dos métodos *initState* e *dispose* e dos métodos privados utilizados para funcionamento da classe, o passo óbvio e necessário é conhecer o método *build* do widget **TabIMC**. Para facilitar a compreensão, o conteúdo do método *build* será dividido em três figuras (Figura 5.13, Figura 5.14 e Figura 5.15).

Figura 5.13 - *Campos de edição de Altura e Peso*



Fonte: Elaborado pelos autores.

Na Figura 5.13 tem-se inicialmente a definição de um **Container** e suas margens (40 para o topo, 10 a esquerda e 10 a direita). Esse **Container** irá garantir que nenhum dos filhos do **Column** interno a ele fique sem espaçamento na borda do **TabIMC**.

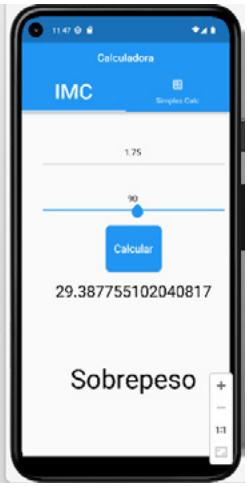
O **Column** organiza seus filhos a partir do topo (*MainAxixAlignment.start*) e os distribui de acordo com seus pesos (*flex* dos **Expanded**). Os 2 (dois) campos de edição para altura e peso possuem peso 1 (um), o botão "Calcular" também possui peso 1 (um), já os 2 (dois) **Text** que mostram o valor do IMC e a situação referente a esse IMC possuem peso 2 (dois). Na prática isso significa que o espaço do **Column** foi dividido em 7 (sete) partes; 2 (duas) partes para os campos de edição (uma para cada um), 1 (uma) parte para o botão "Calcular", 2 (duas) partes para a apresentação do IMC e 2 (duas) partes para a apresentação da situação do indivíduo referente ao IMC calculado.

A Figura 5.14 mostra o código referente ao botão "Calcular" e o resultado da operação, no emulador do Android 13, para cálculo de um IMC para um indivíduo com 1.75 metros e 90 quilos. O ponto chave desse código é a chamada do método *setState* no parâmetro *ao_clicar* do **BotaoAzul**, essa chamada irá executar o código método anônimo passado como parâmetro e posteriormente irá acionar novamente o método *build* redesenhando o **TabIMC**.

No que se refere ao conteúdo do método anônimo passado como parâmetro para o método *setState* tem-se a chamada do método *_calcularIMC* apresentado na Figura 5.10. O método *_calcularIMC* irá retornar o valor do IMC a partir dos controladores associados aos campos de edição de altura (*controladorAltura*) e peso (*controladorPeso*). Esse valor será colocado na variável *imc* e, caso seja válido, irá modificar as variáveis que fornecem os conteúdos dos **Text** para a exibição do imc (*valor_imc*) e da situação referente ao imc (*situacao_imc*). Se *imc* for nulo, ou seja, se não foi possível fazer o cálculo (um número em formato inválido, por exemplo), os valores do imc (*valor_imc*) e da situação referente ao imc (*situacao_imc*) ficam com o padrão para Strings vazias (""). É importante notar que *imc* só pôde ser nula porque sua declaração permitiu isso através do símbolo '?', ou seja, se fosse declarada com apenas *double* ao invés de *double?* a variável *imc* não aceitaria receber valor nulo.

Figura 5.14 - *BotaoAzul "Calcular"*

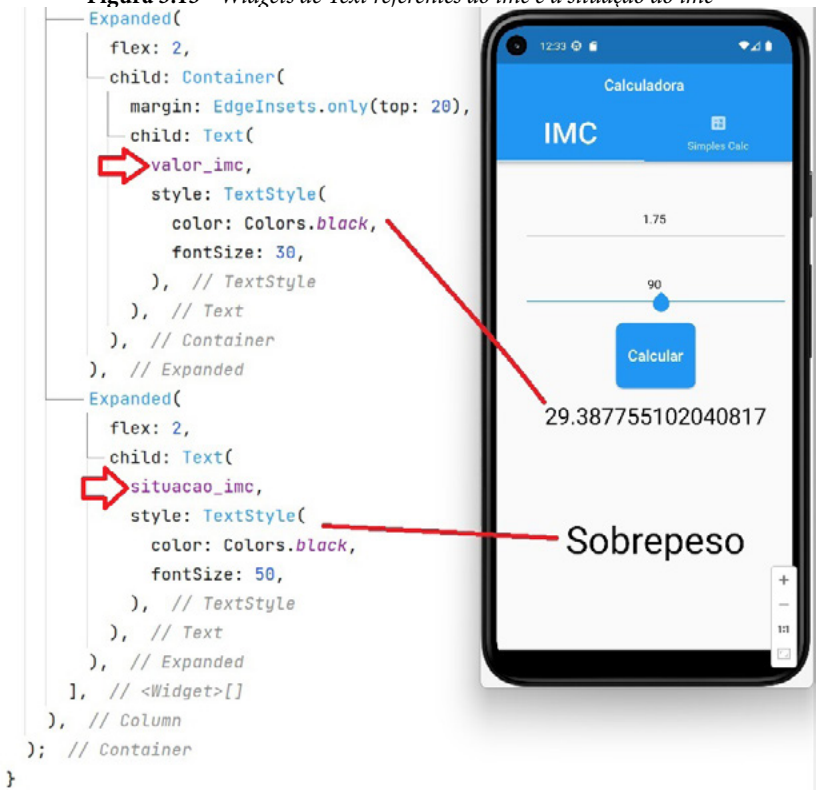
```
Expanded(
  flex: 1,
  child: BotaoAzul(
    texto: "Calcular",
    ao_clicar: () {
      setState(() {
        double? imc = _calcularIMC();
        if(imc == null) {
          valor_imc = "";
          situacao_imc = "";
        }else {
          valor_imc = imc.toString();
          situacao_imc = _determinarSituacaoIMC(imc);
        }
      });
    },
  ),  // BotaoAzul
),  // Expanded
```



Fonte: Elaborado pelos autores.

Por fim, na Figura 5.15, é possível conhecer os **Text** que apresentam o imc e a situação referente ao imc. O ponto chave desses Text é verificar que o conteúdo apresentado por eles é o imc calculado anteriormente (no método anônimo passado como parâmetro para o *setState*) e atribuído para a variável *valor_imc* e a situação referente ao imc calculado anteriormente e atribuído para a variável *situacao_imc*.

Figura 5.15 - *Widgets de Text referentes ao imc e a situação do imc*



Fonte: Elaborado pelos autores.

5.3 Toast (plugin externo)

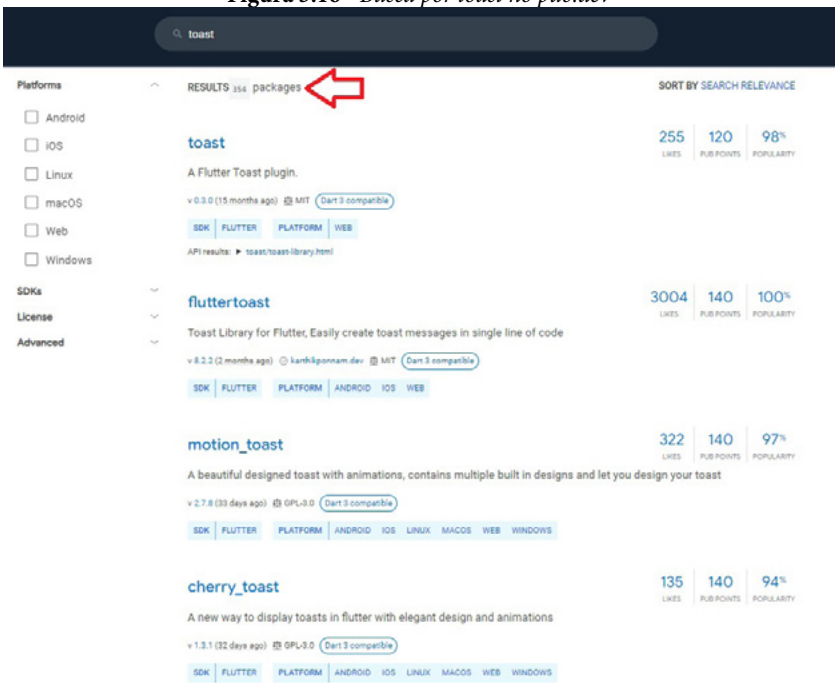
O Toast é uma mensagem que é exibida de forma temporária numa pequena janela flutuante. Normalmente é utilizado para notificar erros ou alertas ao usuário. O Toast é muito utilizado por programadores Java para Android Nativo, sendo praticamente uma das formas padrão para notificações de erros e alertas ao usuário.

No Flutter o Toast não existe de forma nativa, mas pode ser adicionado a qualquer aplicação na forma de um plugin externo. A Figura 5.16 ilustra a busca por Toast no site Pub.dev[13] (é importante

13 https://pub.dev/

notar que na busca foram encontrados 354 pacotes, mas só foram mostrados os 4 primeiros). Esse site contempla os plugins externos do Flutter e deve sempre servir como apoio no momento de se buscar por bibliotecas complementares, evitando assim reprogramação desnecessária, ou seja, não há por que "reinventar a roda" se ela já existe e pode ser utilizada.

Figura 5.16 - *Busca por toast no pub.dev*



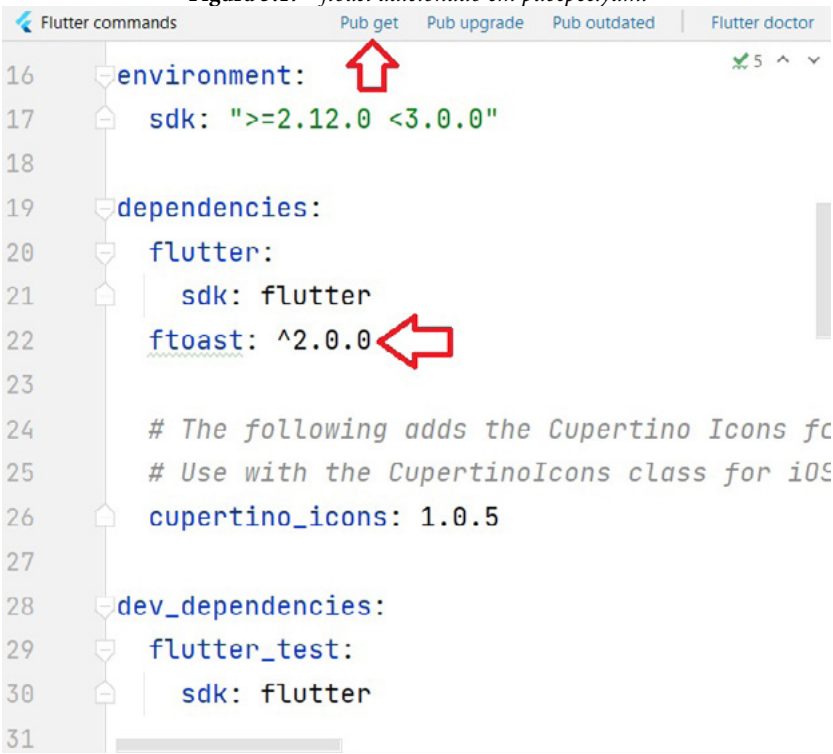
Fonte: Elaborado pelos autores.

No widget **TabSimplesCalc** foi utilizado o plugin externo *ftoast*[14]. O primeiro passo para uso desse plugin é acrescentar a dependência do *ftoast* no arquivo pubspec.yaml (esse e os próximos passos vão valer para praticamente todos os plugins que forem utilizados em projetos Flutter). O segundo passo é usar *flutter pub get* no prompt do Android

14 https://pub.dev/packages/ftoast

Studio ou, de forma mais prática, clicar no menu *Pub get* que aparece no uso do arquivo pubspec.yaml. A Figura 5.17 mostra o acréscimo do *ftoast* (versão ^2.0.0) no arquivo pubspec.yaml, além do menu *Pub get*.

Figura 5.17 - *ftoast adicionado em pubspec.yaml*



Fonte: Elaborado pelos autores.

Após adicionar o plugin do *ftoast* através do comando *flutter pub get* já é possível utilizar essa biblioteca. Nesse contexto se faz necessário estudar detalhadamente a documentação do plugin/biblioteca para compreender seu funcionamento e como pode ser utilizado.

Especificamente no widget personalizado **TabSimplesCalc** o *ftoast* foi utilizado para alertar quando deseja-se acrescentar uma nova linha de operações, sem a linha anterior ter sido terminada. No **TabSimplesCalc** cada conta é executada numa linha que contém 2 (dois) valores, 1 (uma) operação e 1 (um) resultado, quando é solicitado acres-

centar uma nova linha (*Botão +*) utiliza-se o resultado da linha anterior como o primeiro dos valores da linha seguinte, mas obviamente esse valor de resultado só existe quando a linha efetivamente foi executada, ou seja, quando o *Botão =* (da linha atual) for clicado. O Toast então deve avisar o usuário que deve terminar a operação de uma linha para acrescentar uma nova linha. A Figura 5.18 apresenta a tentativa do usuário criar uma nova linha sem terminar a operação da linha anterior e a execução do Toast que trata essa situação:

Figura 5.18 - *Usando FToast.toast*

```
    } else {
      FToast.toast(
        context,
        msg: "Você deve terminar a operação anterior ( clicar em '=' ) ",
        color: Colors.red ,
        duration: 5000,
        msgStyle: TextStyle(
            color: Colors.white,
            fontSize: 16.0,
        ), // TextStyle
      );
    }
  });
 },
), // BotaoAzul
```

Fonte: Elaborado pelos autores.

O uso do plugin *ftoast* no aplicativo foi bastante simples, basicamente foi acionado o método estático *FToast.toast*, conforme é possível ver na Figura 5.18. O primeiro parâmetro (*context)* é um método (*BuildContext get context*) definido na classe **State** do Flutter (*abstract class State<T extends StatefulWidget> with Diagnosticable*) e está relacionado com o contexto do widget, na prática ele vai permitir a ligação entre o Toast que será criado (*FToast.toast*) e o **TabSimplesCalc**. O parâmetro *msg* determina a mensagem que será exibida no Toast. O parâmetro *color* possibilita informar a cor de fundo do Toast. Já o parâmetro *duration* determina o tempo de exibição em milisegundos. Por último *msgStyle* possibilita customizar fonte exibida no Toast.

Para uso do *FToast.toast*, no **TabSimplesCalc**, foi necessário fazer a importação da classe[15] no início do arquivo que contém o código de **TabSimplesCalc**.

Por fim, o *FToast.toast* possui vários outros parâmetros opcionais que podem adicionar subtítulo, ícone entre outras opções. Para maiores informações sobre as diversas possibilidades basta estudar a página do projeto[16].

5.4 SeletorOpcoes (um DropdownButton customizado)

O DropdownButton é um widget de seleção de um item numa lista de itens no estilo do JComboBox do JavaSwing (ZUKOWSKI, 2005). Para facilitar sua compreensão ele será analisado a partir do widget customizado SeletorOpcoes.

TabSimplesCalc é um widget que utiliza outro widget customizado, o widget LinhaConta. Uma LinhaConta é um Form que contém uma Row e essa possui vários Expanded como seus filhos, esses Expanded, por sua vez, distribuem o espaço para cada um dos widgets internos a LinhaConta.

LinhaConta é um widget utilizado para tratar uma operação de conta e portanto possui 2 (dois) campos para fornecimento dos valores para a conta; 1 (um) campo para a apresentação do resultado, 1 (um) botão para acionar a conta em si ("botão de igual" representado pelo texto "=") e um SeletorOpcoes contendo a operação que deverá ser executada. Esse SeletorOpcoes é o foco deste tópico.

A Figura 5.19 mostra o que é uma LinhaConta (quadro vermelho) e um SeletorOpcoes (setas verdes):

15 import 'package:ftoast/ftoast.dart';

16 https://pub.dev/packages/ftoast

Figura 5.19 - *Usando o widget customizado SeletorOpcoes*

```
Expanded(
  // Peso desse componente (como no Android)
  flex: 1,
  child: Center(
    child: SeletorOpcoes(
      opcoes: _operacoes,
      valor_selecionado: _operacao_selecionada,
      ao_mudar_opcao: (String novoItemSelecionado){
        _operacao_selecionada = novoItemSelecionado;
        _resultado = "";
      },
    ), // SeletorOpcoes
  ), // Center
), // Expanded
Expanded(...), // Expanded
Expanded(...), // Expanded
Expanded(...), // Expanded
], // <Widget>[]
), // Row
), // Container
```

Fonte: Elaborado pelos autores.

O widget customizado SeletorOpcoes possui o parâmetro *opcoes* que contém a lista das Strings (*var _operacoes = [“+”, “-”, “x”, “/”];*) que serão exibidas quando o SeletorOpcoes for clicado. O parâmetro *valor_selecionado* (*var _operacao_selecionada = “+”;* inicialmente) mostra o valor atualmente apresentado no SeletorOpcoes. Já o parâmetro *ao_mudar_opcao* define a função que deve ser executada na troca de uma seleção por outra no SeletorOpcoes.

Como é possível verificar, o uso do SeletorOpcoes é bem simples porém mais limitado que do DropdownButton. Nesse contexto, o próximo passo é conhecer o código do SeletorOpcoes e verificar como ele utiliza o DropdownButton.

A Figura 5.20 apresenta o código do widget SeletorOpcoes. Além dos parâmetros anteriormente descritos, merece atenção também o parâmetro/atributo *icone*, esse parâmetro possibilita escolher o ícone que será utilizado no SeletorOpcoes. Entretanto a passagem do ícone a ser utilizado é opcional e caso o programador não escolha que ícone deseja utilizar em seu SeletorOpcoes, o ícone *Icons.arrow_drop_down* será utilizado.

A Figura 5.21 contempla o método *build* relacionado ao widget SeletorOpcoes. É importante notar que o SeletorOpcoes é um Drop-

downButton<String>, isso significa que ele ficará restrito ao tratamento de strings, o que facilita seu uso mas, até certo ponto, o limita.

Figura 5.20 - *Código do widget SeletorOpcoes*

```
import 'package:flutter/material.dart';

class SeletorOpcoes extends StatefulWidget {
  List<String>? opcoes;
  Icon? icone ;
  String? valor_selecionado;
  Function? ao_mudar_opcao;

  SeletorOpcoes({
    Key? key,
    this.opcoes,
    this.icone,
    this.valor_selecionado,
    this.ao_mudar_opcao
  }): super(key: key){
    if(this.opcoes == null) this.opcoes = <String>[];
    if(this.icone == null) this.icone = Icon(Icons.arrow_drop_down);

  }
  _SeletorOpcoesState createState() => _SeletorOpcoesState();
}
```

Fonte: Elaborado pelos autores.

Figura 5.21 - *Método build do widget SeletorOpcoes*

```
class _SeletorOpcoesState extends State<SeletorOpcoes> {
  @override
  Widget build(BuildContext context) {
    return DropdownButton<String>(
      value: widget.valor_selecionado,
      icon: widget.icone,
      iconSize: 24,
      elevation: 16,
      underline: Container(
        height: 2,
        color: Colors.blueAccent,
      ),  // Container
      items : widget.opcoes!.map((String dropDownStringItem) {
        return DropdownMenuItem<String>(
          value: dropDownStringItem,
          child: Text(dropDownStringItem),
        );  // DropdownMenuItem
      }).toList(),
      onChanged: (String? novoItemSelecionado) {
        setState(() {
          widget.valor_selecionado = novoItemSelecionado;
          if(widget.ao_mudar_opcao != null)
            widget.ao_mudar_opcao!(novoItemSelecionado);
        });
      },
    );  // DropdownButton
  }
}
```

Fonte: Elaborado pelos autores.

No **DropdownButton** da Figura 5.21 diversos parâmetros são atribuídos, nesse contexto serão discutidos, em tópicos, cada um deles:

value: contém o valor que deve ser apresentado no **DropdownButton**;

icon: ícone utilizado no **DropdownButton**;

iconSize: tamanho do ícone utilizado (no **SeletorOpcoes** optou-se por deixar esse valor fixo em 24);

elevation: coordenada z (para sensação de profundidade) em que o menu aparecerá quando aberto. As elevações 1, 2, 3, 4, 6, 8, 9, 12, 16 e 24 possuem sombras definidas. A elevação padrão é 8 (no **SeletorOpcoes** o padrão e valor fixo é 16);

underline: permite um sublinhado no **DropdownButton** (no **SeletorOpcoes** essa linha é da cor azul com espessura 2 (dois));

itens: é uma lista de **DropdownMenuItem** (*List<DropdownMenuItem<T>>*). Um **DropdownMenuItem** é composto por um valor (*value*) que é do tipo T (Generics) e um *child* que é usado para a exibição desse objeto no **DropdownButton**. Ou seja, *value* possui o valor do **DropdownMenuItem** e *child* possui a representação visual para o **DropdownButton**. Especificamente para o **SeletorOpcoes** optou-se por gerar a lista de **DropdownMenuItem** a partir da lista de **String**. Cada String passou a ser o *value* de um **DropdownMenuItem** e o conteúdo de cada **String** foi utilizado para gerar **Text** para a exibição no **DropdownButton**;

onChanged: define o comportamento que deve ser implementado na troca de seleção no **DropdownButton**. É importante notar que a tipagem do parâmetro *novoItemSelecionado* não é obrigatória, ou seja, se desejado pode-se retirar o texto *String?*. Especificamente para **SeletorOpcoes** quando a funcionalidade *ao_mudar_opcao* é definida esse código garante que ela será executada (*widget.ao_mudar_opcao!(novoItemSelecionado)*). Por fim também é importante notar que o *setState* é acionado e portanto ocorrerá a atualização visual do widget ao fim de sua execução.

5.5 LinhaConta

LinhaConta é um widget customizado responsável por gerenciar uma linha de execução de operação. Na Figura 5.19 é possível visualizar, no quadro vermelho, uma LinhaConta.

A TabSimplesCalc utiliza uma listagem de LinhaConta para seu funcionamento. Na prática LinhaConta fica responsável por tratar as questões mais locais referentes a uma operação em si e TabSimplesCalc trata do conjunto das operações, ajustando a saída de uma operação de uma LinhaConta para outra. Para que esse mecanismo funcione é fundamental o uso de um objeto não visual para garantir a gestão dos valores anteriormente calculados, ou seja, é preciso armazenar os resultados de cada conta, de cada uma das LinhaConta, para que caso sejam excluídas (Botão -) esses dados possam ser recuperados. Para tanto foi criada a classe ControladorLinhaConta. A Figura 5.22 apresenta o código do ControladorLinhaConta:

Figura 5.22 - *Código do ControladorLinhaConta*

```
class ControladorLinhaConta{
  late List<String> resultados;

  ControladorLinhaConta(){
    resultados = <String>[];
  }

  void adicionarNovoResultado(String resultado){
    resultados.add(resultado);
  }

  int obterQuantidadeResultados(){
    return resultados.length;
  }

  String obterUltimoResultado(){
    return resultados[resultados.length - 1];
  }

  void removerUltimoResultado(){
    resultados.removeLast();
  }

  void modificarUltimoResultado(String resultado) {
    resultados[resultados.length - 1] = resultado;
  }
}
```

Fonte: Elaborado pelos autores.

O uso de controladores em Flutter/Dart é fundamental pois facilita a separação do que é código visual (para exibição) e o que é lógica de negócio/dados (para gestão da informação). Além disso a estrutura de criação hierárquica de widgets também direciona ao uso de controladores, isso inclusive já foi tratado quando foram discutidos os controladores (TextEditingController) dos TextFormField. Especificamente através do ControladorLinhaConta é feita a gestão dos resultados das contas de TabSimplesCalc pois através dos *Botões* + e – é possível inserir (*Botão* +) ou eliminar (*Botão* -) uma LinhaConta da listagem de LinhaConta da TabSimplesCalc.

A Figura 5.23 ilustra as importações do widget customizado LinhaConta e a necessidade de ser passado, como parâmetro, um ControladorLinhaConta para a construção de uma LinhaConta:

Figura 5.23 - *Definição do widget customizado LinhaConta*

```dart
import 'package:calculadoraimcflutter/widgets/botao_azul.dart';
import 'package:calculadoraimcflutter/widgets/controlador_linha_conta.dart';
import 'package:calculadoraimcflutter/widgets/seletor_opcoes.dart';
import 'package:flutter/material.dart';
import 'package:ftoast/ftoast.dart';
import 'campo_edicao_double.dart';


class LinhaConta extends StatefulWidget {
  // Controlador para esse objeto em si (utilizado para acessar os valores externamente)
  ControladorLinhaConta controladorLinhaConta;

  LinhaConta({
    Key? key,
    required this.controladorLinhaConta}): super(key: key);

  @override
  _LinhaContaState createState() => _LinhaContaState();
}
```

Fonte: Elaborado pelos autores.

O passo seguinte é conhecer os atributos privados (todos precedidos por '_') de um _LinhaContaState:

_controladorValor1: controlador associado ao TextFormField que fornece o primeiro valor para uma conta;

_controladorValor2: controlador associado ao TextFormField que fornece o segundo valor para uma conta;

_resultado_ja_adicionado: booleano utilizado para identificar se a conta já foi feita ou não, ou seja, se a operação de "=" (igualdade) já foi executada;

_operacoes: array contendo as 4 operações;

_operacao_selecionada: controla a operação corrente, ou seja, a operação selecionada;

_resultado: armazena o resultado da execução de uma conta;

_focoValor2: vincula o foco ao widget de CampoEdicaoDouble que contempla o segundo valor. É utilizado para controlar que, ao clicar em "Próximo", o foco deve ir do primeiro CampoEdicaoDouble para o segundo CampoEdicaoDouble;

_focoBotaoIgual: vincula o foco ao widget de BotaoAzul que contempla o "Botão =". É utilizado para controlar que, ao clicar em "Próximo", o foco deve ir do segundo CampoEdicaoDouble para o "Botão =";

_formkey: é utilizado para gerenciar o Form interno de Linha-Conta controlando a validação dos CampoEdicaoDouble (*_formkey.currentState!.validate()*).

A Figura 5.24 apresenta esses atributos de um _LinhaContaState:

Figura 5.24 - *Atributos privados de _LinhaContaState*

```
class _LinhaContaState extends State<LinhaConta> {
  // Controladores para os campos de edição
  final _controladorValor1 = TextEditingController();
  final _controladorValor2 = TextEditingController();

  // Boolenao para informar se já foi executada a operação de "=",
  // isso serve para sabermos se é para modificar uma gravação de resultado anterior
  // ou gerar um novo resultado
  bool _resultado_ja_adicionado = false;

  // Dados para o DropdownButton
  var _operacoes = ["+", "-", "x", "/"];
  var _operacao_selecionada = "+";

  // Texto da resposta das contas
  var _resultado = "";

  // Controladores de foco
  final _focoValor2 = FocusNode();
  final _focoBotaoIgual = FocusNode();

  // Variável utilizada para validar os campos do formulário
  final _formkey = GlobalKey<FormState>();
```

Fonte: Elaborado pelos autores.

O próximo passo é conhecer, em linhas gerais, o método *build* do widget customizado **LinhaConta**, a Figura 5.25 apresenta o esse método:

Figura 5.25 - *build de LinhaConta*

```
@override
Widget build(BuildContext context) {
  _inicializarValor1();
  return Form(
    // Com o formulário conseguimos acionar os validadores de cada um dos campos de
    // edição
    key: _formkey,
    child: Container(
      margin: EdgeInsets.only(left: 5, right: 5, top: 5, bottom: 5),
      child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Expanded(...),  // Expanded
          Expanded(...),  // Expanded
          Expanded(...),  // Expanded
          Expanded(...),  // Expanded
          Expanded(...),  // Expanded
        ],  // <Widget>[]
      ),  // Row
    ),  // Container
  );  // Form
}
```

Fonte: Elaborado pelos autores.

Basicamente tem-se um **Form** para validar os **CampoEdicao** que estão dentro do primeiro e terceiro **Expanded**. No segundo **Expanded** está o **SeletorOpcoes** com a possibilidade da escolha de operação que será executada. No quarto **Expanded** está o **BotaoAzul** (Botão =). No último **Expanded** tem-se o **Text** que mostra o resultado da operação.

O método privado *_inicializarValor1()* serve para preencher o controlador (*final _controladorValor1 = TextEditingController();*) do **CampoEdicao** do primeiro **Expanded**, ou seja, possibilita inicializar o primeiro **CampoEdicao** com o valor obtido da **LinhaConta** anterior e fornecido via **ControladorLinhaConta**.

Apresentando o código, Figura 5.26, do quarto **Expanded**, é possível visualizar o acionamento do **BotaoAzul** (*Botao* =). Basicamente ocorre o acionamento do método *setState* para posterior atualização do widget. No método *setState* ocorre a validação dos 2 (dois) **CampoEdicao** da **LinhaConta**. Caso a validação tenha ocorrido

com sucesso a operação é executada através da chamada do método _*executarOperacao()*. Em seguida é feita uma validação para saber se já há resultado anterior, se não houver acrescenta-se um novo resultado no **ControladorLinhaConta**, se houver modifica-se o último resultado.

Figura 5.26 - *Código do "botão ="*

```
- Expanded(
    // Peso desse componente (como no Android)
    flex: 1,
    child: BotaoAzul(
      texto: "=",
      ao_clicar: (){
        setState(() {
          // Chamando os validadores dos campos de edição
          bool formOk = _formkey.currentState!.validate();
          if (formOk) {
            _executarOperacao();
            if (_resultado_ja_adicionado) {
              widget.controladorLinhaConta.modificarUltimoResultado(
                  _resultado);
            } else {
              widget.controladorLinhaConta.adicionarNovoResultado(
                  _resultado);
              _resultado_ja_adicionado = true;
            }
          }
        });
      },
      marcador_foco: _focoBotaoIgual,
    ),  // BotaoAzul
  ),  // Expanded
```

Fonte: Elaborado pelos autores.

Já a Figura 5.27 apresenta o código do método _*executarOperacao()*. É nesse método que os valores são capturados e a conta efetivamente ocorre. Basicamente esse código captura os valores digi-

tados, os converte para *double*, em seguida verifica a operação selecionada e faz a conta de acordo com essa operação. Caso haja algum problema, apresenta um Toast (*_makeToast*) com a mensagem erro (o usuário pode, por exemplo, tentar dividir um número por 0(zero)). Na sequência, atualiza o atributo *_resultado* para que possa ser atualizada a representação visual dele.

Figura 5.27 - *Código do método _executarOperacao*

```
void _executarOperacao() {
  try{
    double valor1 = double.parse(_controladorValor1.text);
    double valor2 = double.parse(_controladorValor2.text);
    double resultado = 0;
    switch(_operacao_selecionada){
      case "+":
        resultado = valor1 + valor2;
        break;
      case "-":
        resultado = valor1 - valor2;
        break;
      case "x":
        resultado = valor1 * valor2;
        break;
      case "/":
        resultado = valor1 / valor2;
        break;
    }
    // 2 casas depois da vírgula
    _resultado = resultado.toStringAsFixed(2);
  } on Exception catch (erro){
    _makeToast(erro.toString());
  }
}
```

Fonte: Elaborado pelos autores.

Por fim tem-se o código do método *_makeToast* apresentado na Figura 5.28:

Figura 5.28 - *Código do método makeToast*

```
_makeToast(String mensagem){
  FToast.toast(
      context,
      msg: "Erro: $mensagem",
      color: Colors.red ,
      duration: 5000,
      msgStyle: TextStyle(
      color: Colors.white,
      fontSize: 16.0,
  ));
}
```

Fonte: Elaborado pelos autores.

Basicamente tem-se o mesmo uso apresentado no tópico 5.3: *context* é um método que vem da classe **State** e vincula o Toast com o widget **LinhaConta**, *msg* mostra a mensagem de erro que será exibida, já *color* define que a cor de exibição do fundo do Toast será vermelha, com duração de 5 (cinco) segundos (*duration: 5000*) e por fim a fonte utilizada será branca e de tamanho 16 (dezesseis). A Figura 5.29 mostra a exibição resultante da chamada do método *_makeToast* para a tentativa de dividir o número 3 (três) pelo número 0 (zero):

Figura 5.29 - *Tentando dividir um número por 0 (zero)*

Fonte: Elaborado pelos autores.

5.6 TabSimplesCalc

TabSimplesCalc é um widget customizado que representa o conteúdo da segunda aba do aplicativo. Também precisará ser um StatefullWidget pois tem-se mudança de estado e a chamada do método *setState*.

Para esse widget optou-se por começar a análise a partir do método *build*. Como já foi visto anteriormente o método *build* é responsável por

"desenhar" o widget. A Figura 5.30 apresenta o método *build* do widget TabSimplesCalc:

Figura 5.30 - *Método build de TabSimplesCalc*

```
@override
Widget build(BuildContext context) {
  return SingleChildScrollView(
    // Define as margens do Container
    padding: EdgeInsets.only(top: 20, left: 10, right: 10),
    child: Column(
      // No eixo y (principal da coluna) vai colocar
      // os objetos no topo
      mainAxisAlignment: MainAxisAlignment.start,
      children: _body(),
    ),  // Column
  );  // SingleChildScrollView
}
```

Fonte: Elaborado pelos autores.

Logo no início do método *build* tem-se o widget **SingleChildScrollView**. Esse é um widget utilizado para a apresentação de listagens de itens permitindo que sejam "*scrollados*", ou seja, quando não houver mais espaço para a exibição dos itens uma barra de *scroll* será acionada e o Flutter não apresentará erro de falta de espaço para widgets. Especificamente para o método *build*, do **TabSimplesCalc**, o objetivo do uso do **SingleChildScrollView** é permitir o uso de quantas **LinhaConta** forem desejadas, ou seja, a colocação de uma nova **LinhaConta** não fica limitada ao espaço da tela, podendo ser "*scrollada*" se necessário.

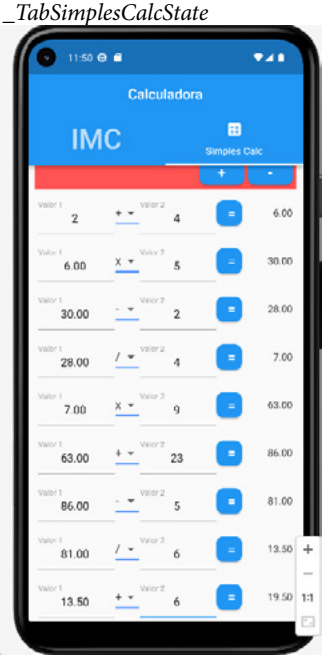
Na sequência ocorre a definição das margens e o uso de uma coluna (**Column**) que vai organizar as **LinhaConta** a partir do top (*start*) da tela.

O método *_body()* fica responsável por criar o menu de botões que acrescenta (**Botão +**) e retira (**Botão -**) uma **LinhaConta**, além de controlar, com o apoio do **ControladorLinhaConta**, a listagem das **LinhaConta**.

A Figura 5.31 mostra a definição do widget customizado **TabSimplesCalc** e seu State (**_TabSimplesCalcState**) além do resultado do uso do **SingleChildScrollView** para fazer o *scroll* de **LinhaConta** apresentado no emulador do Android 13:

Figura 5.31 - *Código de TabSimplesCalc e _TabSimplesCalcState*

```
class TabSimplesCalc extends StatefulWidget {
  @override
  _TabSimplesCalcState createState() => _TabSimplesCalcState();
}

class _TabSimplesCalcState extends State<TabSimplesCalc> {
  List<Widget>? _linhas;
  ControladorLinhaConta _controlador = ControladorLinhaConta();

  List<Widget> _body() {
    if (_linhas == null) {
      _linhas = <Widget>[
        Container(
          color: Colors.redAccent,
          padding: EdgeInsets.only(right: 5),
          child: Row(
            mainAxisAlignment: MainAxisAlignment.end,
            children: <Widget>[
              BotaoAzul(...),  // BotaoAzul
              SizedBox(
                width: 5,
              ),  // SizedBox
              BotaoAzul(...),  // BotaoAzul
            ],  // <Widget>[]
          ),  // Row
        ),  // Container
        LinhaConta(controladorLinhaConta: _controlador),
      ];  // <Widget>[]
    }
    return _linhas!;
  }
```



Fonte: Elaborado pelos autores.

O atributo *_linhas*, de **_TabSimplesCalcState**, é uma listagem de **LinhaConta**. Já *_controlador* é o **ControladorLinhaConta** que tratará da gestão das **LinhaConta** criadas.

No início do método privado *_body* já é possível ver a verificação se o atributo *_linhas* foi ou não inicializado (*_linhas == null*). Na primeira vez que *_body* for acionado *_linhas* será nulo, nesse caso a listagem de **LinhaConta** será criada e a primeira **LinhaConta** será definida, além disso, nos códigos de clique dos 2 (dois) **BotaoAzul** (não exibidos na Figura 5.31) ocorrerá a inclusão (Botão +) ou a exclusão (Botão -) de **LinhaConta**. Caso *_linhas* não seja nulo ele é apenas retornado.

A Figura 5.32 apresenta o código referente ao **BotaoAzul** que insere uma **LinhaConta**, o "Botão +":

Figura 5.32 - *Código do BotaoAzul "+"*

```
BotaoAzul(
  texto: "+",
  ao_clicar: () {
    setState(() {
      // Condição para evitar que o botão de + acione novas linhas sem que
      // já tenhamos resultados das linhas anteriores
      if ((_linhas!.length-1) == _controlador.obterQuantidadeResultados()) {
        _linhas!.add(
          LinhaConta(
            controladorLinhaConta: _controlador,
          ),  // LinhaConta
        );
      } else {
        FToast.toast(
          context,
          msg: "Você deve terminar a operação anterior ( clicar em '=' ) ",
          color: Colors.red ,
          duration: 5000,
          msgStyle: TextStyle(
              color: Colors.white,
              fontSize: 16.0,
          ),  // TextStyle
        );
      }
    });
  },
),  // BotaoAzul
```

Fonte: Elaborado pelos autores.

O primeiro atributo a ser definido é *texto*, esse atributo apresenta o texto que será apresentado no botão. Em *ao_clicar* tem-se o mecanismo de atualização da listagem através da chamada ao método *setState* e do corpo do método anônimo passado com parâmetro para ele.

Basicamente é feita uma validação, com apoio do *_controlador* (**ControladorLinhaConta**) para saber se a operação anterior foi finalizada ou não, ou seja, se na última **LinhaConta** o "Botão =" foi ou não acionado. Essa validação é fundamental para que uma nova **Linha-**

Conta não seja criada sem o respectivo resultado da **LinhaConta** anterior. É importante compreender o papel do *_controlador*, ele é determinante para comunicar uma **LinhaConta** com o **TabSimplesCalc**.

O uso de widgets customizados e de objetos não visuais para troca de dados é uma prática recomendável em Flutter pois possibilita maior reaproveitamento de widgets, maior facilidade de manutenção e até de adaptabilidade a mudanças na tecnologia.

Com relação a essa maior adaptabilidade a mudanças, um exemplo interessante é o código do widget **BotaoAzul**, em sua primeira versão ele utilizava como base um **RaisedButton** mas esse foi tornado obsoleto e teve de ser substituído por um **ElevatedButton**. Nesse contexto, bastou a troca de um widget (**RaisedButton**) por outro (**ElevatedButton**) no **BotaoAzul** que todos os códigos que usavam esse widget (**BotaoAzul**) já receberam essa mudança. Essa é uma prática recomendável pois o Flutter tem mudado muito rapidamente e essa abordagem ajuda o programador a diminuir a necessidade de ajustes, uma vez que, possivelmente, a maioria das mudanças poderá ocorrer num local só.

Uma outra questão importante é o uso de objetos não visuais para troca de dados. Em Flutter ocorre comumente um encadeamento de widgets, ou seja, um widget é passado como parâmetro para outro e esse mais externo como parâmetro para outro e assim sucessivamente. Além disso, o ponto de acesso mais comum para a implementação de comportamentos é através de parâmetros que em boa parte das vezes são programados através de métodos anônimos. Essa abordagem pode dificultar o acesso a determinado objeto quando precisamos dele, dessa forma o uso de objetos não visuais como apoio é uma prática fundamental para a facilitação da programação pois além de separar comportamento de representação/visualização de tela organiza dados e comportamentos que ficariam dispersos em várias classes.

Agora que o código do "Botão +" já é conhecido faz-se necessário apresentar o código do "Botão -" (que também é um **BotaoAzul**). Basicamente o "Botão -" remove a última **LinhaConta** criada e verifica se essa **LinhaConta** já havia tido seu resultado calculado. Caso essa

última **LinhaConta** já tenha tido seu resultado calculado é necessário fazer um ajuste no *_controlador* (**ControladorLinhaConta**) para a atualização do último resultado calculado. Caso essa última **LinhaConta** não tenha tido seu resultado calculado (não foi clicado no "Botão =" da **LinhaConta**) não é necessário fazer nada pois o *_controlador* ainda não havia sido modificado. Na Figura 5.33 é possível visualizar o código referente ao "Botão -":

Figura 5.33 - *Código do BotaoAzul "-"*

```
BotaoAzul(
  texto: "-",
  ao_clicar: () {
    setState(() {
      _linhas!.removeLast();
      // Posso ter clicado ou não no operador "=" da LinhaConta
      // se foi clicado a linha já tem valor no controlador, caso contrário não
      // por isso se faz necessária essa validação
      if (_linhas!.length ==
          _controlador.obterQuantidadeResultados())
        _controlador.removerUltimoResultado();
    });
  },
),  // BotaoAzul
```

Fonte: Elaborado pelos autores.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Esse livro não se propõe a esgotar a temática de desenvolvimento para dispositivos móveis em Flutter/Dart, muito pelo contrário, seu objetivo é trazer de forma prática e didática uma introdução a essa tecnologia, a compreensão básica de seus fundamentos.

Buscou-se tratar dos aspectos principais de Linguagem Dart bem como apresentar os principais widgets utilizados em Flutter com foco no desenvolvimento de aplicativos móveis. Para tanto, o uso de muitas imagens e a apresentação de muitos códigos se fez necessário. Um objeto de estudo futuro certamente deve ser a adaptação/uso dessas tecnologias para aplicações desktop padrão ou ainda seu uso para a web, já que esse é um caminho que essas tecnologias já começaram a trilhar.

Além do uso dos widgets básicos buscou-se apresentar como é possível obter widgets externos no site pub.dev[17], além de discutir como potencializar e facilitar a manutenibilidade através da criação de widgets customizados e de objetos não visuais para apoio e gestão dos dados.

Por fim, também como trabalhos futuros, será necessária a criação de outros livros para a discussão de tópicos e tecnologias mais avançados, por exemplo:

17 https://pub.dev/

Tabs persistentes: Exploração de como criar abas com persistência de estado em um aplicativo;

Comunicação com serviços: Discussão sobre como se comunicar com APIs e serviços externos;

Streams e StreamBuilder: Introdução à programação reativa e como trabalhar com fluxos de dados;

Futures e FutureBuilder: Como lidar com operações assíncronas e atualizações de UI no Flutter;

Banco de dados local - SQLite: Utilização de bancos de dados locais para armazenamento de dados;

Serviços do Google Firebase: Integração com os serviços do Firebase para recursos como autenticação, banco de dados em tempo real, armazenamento de arquivos, etc.;

Navegação entre telas: Gerenciamento da navegação entre diferentes telas do aplicativo;

Shared Preferences: Armazenamento simples de dados no dispositivo;

PopupMenuButton: Criação de menus pop-up personalizados;

Font Awesome: Uso de ícones e recursos do Font Awesome no Flutter;

Menu Lateral: Implementação de um menu lateral deslizante para navegação;

Execução de Vídeo: Incorporação e reprodução de vídeos no aplicativo;

Calendário: Integração de recursos de calendário e eventos no aplicativo;

Aplicativos de foto e Galeria: Utilização da câmera e da galeria para manipulação de fotos;

Mapas e GPS: Incorporação de mapas e funcionalidades de localização ao aplicativo;

WebView: Exibição de conteúdo web dentro do aplicativo;

GridView: Uso de layout de grade para exibir conteúdo em formato de grade.

Os tópicos e tecnologias acima apresentados fornecerão aos leitores conhecimentos mais avançados e permitirão que eles explorem diferentes recursos e funcionalidades no desenvolvimento de aplicativos móveis em Flutter/Dart. Ao criar outros livros para discutir esses tópicos, os autores ampliam o escopo do aprendizado e oferecem oportunidades para os leitores aprofundarem ainda mais seus conhecimentos nessa tecnologia. Por fim, a ideia em si é dividir os tópicos e tecnologias acima apresentados em 2 (dois) livros apresentando de forma didática e organizada o Flutter e suas bibliotecas.

REFERÊNCIAS

APACHE. **Welcome to Apache NetBeans**. Delaware, 2023. Disponível em: https://netbeans.apache.org/. Acesso em: 21 jun. 2023.

CARVALHO, G. A. de; S, A. T. D. **Estado nutricional versus equilíbrio estático**: Influência do IMC e da composição corporal na projeção do centro de gravidade de idosos atendidos ambulatorialmente. *[S. l.]*: Novas Edições Acadêmicas, 2014.

DUARTE, W. **Delphi para Android e IOS**: desenvolvendo aplicativos móveis. São Paulo: Brasport, 2015.

ENGHOLM JR., H. **Análise e design orientados a objetos**. São Paulo: Novatec Editora, 2013.

FURGERI, S. **Java**: Ensino didático: Desenvolvimento e implementação de aplicações: Compatível com versão 9 e Jshell com Netbeans. São Paulo: Editora Érica, 2018a.

FURGERI, S. **Programação orientada a objetos**: conceitos e técnicas. São Paulo: Érica, 2018b.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. São Paulo: Bookman, 2000.

GOOGLE. **Criar e gerenciar dispositivos virtuais | Android Studio | Android Developers**. Mountain View, 2023a. Disponível em: https://developer.android.com/studio/run/managing-avds?hl=pt-br. Acesso em: 2 jun. 2023.

GOOGLE. **Executar apps em um dispositivo de hardware | Android Studio | Android Developers**. Mountain View, 2023b. Disponível em: https://developer.android.com/studio/run/device?hl=pt-br. Acesso em: 2 jun. 2023.

GOOGLE. **DartPad**. Mountain View, 2023c. Disponível em: https://dart.dev/tools/dartpad/. Acesso em: 12 jun. 2023.

GOOGLE. **Object class - dart:core library - Dart API**. Mountain View, 2023d. Disponível em: https://api.flutter.dev/flutter/dart-core/Object-class.html. Acesso em: 22 jun. 2023.

GOOGLE. **Mixins**. Mountain View, 2023e. Disponível em: https://dart.dev/language/mixins.html. Acesso em: 22 jun. 2023.

GOOGLE. **Widget catalog**. Mountain View, 2023f. Disponível em: https://docs.flutter.dev/ui/widgets. Acesso em: 26 jun. 2023.

GOOGLE. **Work with tabs**. Mountain View, 2023g. Disponível em: https://docs.flutter.dev/cookbook/design/tabs. Acesso em: 24 jul. 2023.

GOOGLE. **Hot reload**. Mountain View, 2023h. Disponível em: https://docs.flutter.dev/tools/hot-reload. Acesso em: 1 jun. 2023.

GOOGLE. **Border class - painting library - Dart API**. Mountain View, 2023i. Disponível em: https://api.flutter.dev/flutter/painting/Border-class.html. Acesso em: 10 jul. 2023.

GOOGLE. **Flutter - Dart API docs**. Mountain View, 2023j. Disponível em: https://api.flutter.dev/index.html. Acesso em: 10 jul. 2023.

LECHETA, R. R. **Android essencial**: edição resumida do Livro Google Android. São Paulo: Novatec Editora, 2016.

LECHETA, R. R. **Android essencial com Kotlin**. São Paulo: Novatec Editora, 2017.

LIMA, V. **Desenvolvimento mobile**: React-Native, Flutter ou Ionic?. Florianópolis, 2020. Disponível em: https://blog.geekhunter.com.br/desenvolvimento-para-mobile-reactnative-flutter-ionic/. Acesso em: 1 set. 2023.

MELÃO JR., H. **IMCH - Análise matemática dos erros da fórmula de IMC**: conheça o novo IMCH e entenda por que a fórmula tradicional produz diagnósticos incorretos que prejudicam 394 MILHÕES DE PESSOAS. Pindamonhangaba: Hindemburg Melão Jr., 2021.

MEW, K. **Aprendendo Material Design**: domine o Material Design e crie interfaces bonitas e animadas para aplicativos móveis e web. São Paulo: Novatec Editora, 2016.

MICROSOFT. **Xamarin | Plataforma de aplicativo móvel de código aberto para .NET**. Redmond, 2023. Disponível em: https://dotnet.microsoft.com/pt-br/apps/xamarin. Acesso em: 1 jun. 2023.

MIOLA, A. **Flutter complete reference**: create beautiful, fast and native apps for any device. *[S. l.]*: Independently Published, 2020.

NYSTROM, B. **Understanding null safety**. Mountain View, 2020. Disponível em: https://dart.dev/null-safety/understanding-null-safety/. Acesso em: 14 jun. 2023.

STROUSTRUP, B. **The C++ Programming Language**. 4th revised ed. Westport, Conn: Addison-Wesley Professional, 2013.

WINDMILL, E. **Flutter in action**. Shelter Island, NY: Manning Publications, 2020.

ZUKOWSKI, J. **The Definitive Guide to Java Swing**. 3. ed. Berkeley, Califórnia: Apress, 2005.

Sobre os autores

Giovany Frossard Teixeira

Graduado em Ciência da Computação - UFES (2004), Mestre em Informática - UFES (2006) e Doutor em Educação - UniNorte (2015) - revalidado na UFPR (Universidade Federal do Paraná). Professor Titular do Instituto Federal do Espírito Santo - Campus Colatina vinculado ao curso de Bacharelado em Sistemas de Informação. Professor Colaborador do Mestrado Profissional em Propriedade Intelectual e Transferência de Tecnologia para Inovação (PROFNIT) Ponto Focal IFES - Campus Colatina. Tem experiência na área de Ciência da Computação, com ênfase em Linguagens de Programação, Otimização, Software Básico e Programação para Dispositivos Móveis, além de atuar na área de Propriedade intelectual e transferência de tecnologia para inovação em núcleos de inovação tecnológica (NITs). Na área de educação tem foco de estudo no ensino a distância e em avaliação da aprendizagem.

Currículo: http://lattes.cnpq.br/7406806998563478

Alextian Bartholomeu Liberato

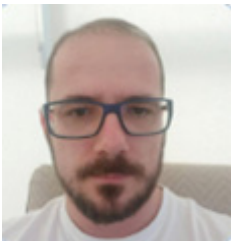
Doutorado em Ciência da Computação UFES (2018), Mestre em Pesquisa Operacional e Inteligência Computacional - UCAM (2009), possui graduação em Sup. Tecnologia em Processamento de Dados pelo Centro Universitário do Espírito Santo - UNESC (2004). Especialista em Tecnologia de Redes com Cabeamento Estruturado - UFLA (2006), Licenciado em Computação pelo Centro Universitário Claretiano (2021). Atualmente é Coordenador do Mestrado Profissional em Propriedade Intelectual e Transferência de Tecnologia para Inovação (PROFNIT) Ponto Focal IFES - Campus Colatina. Tem como áreas de interesse os seguintes temas: Redes de Computadores e Redes Definidas por Software - SDN e Propriedade Intelectual e Transferência de Tecnologia para Inovação.

Currículo: http://lattes.cnpq.br/5443992982789294

Renan Osório Rios

Professor efetivo do Instituto Federal do Espírito Santo (IFES) - Campus Colatina. Doutor em Ciências da Educação pela Universidad del Norte (Uninorte) e revalidado na UFPR (Universidade Federal do Paraná). Mestre em Modelagem Matemática e Computacional pelo Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG). Graduado em Sistemas de Informação pelo Centro Universitário do Espírito Santo (UNESC) e técnico de Informática pelo Centro de Federal de Educação Tecnológica do Espírito Santo (CEFET-ES). Atuando principalmente nos seguintes temas: Educação, Robótica Educacional, Extensão Universitária, Programação Inicial e desenvolvimento de e-commerce.

Currículo: http://lattes.cnpq.br/3555360133532677



Igor Carlos Pulini

Doutor em Engenharia de Produção Universidade Federal do Rio Grande do Sul - UFRGS (2018), Mestrado em Pesquisa Operacional e Inteligência Computacional - UCAM (2012), Pós-graduação em Java-Tecnologias de Desenvolvimento de Sistemas - UFES (2006) Graduação em Ciência da Computação - UVV (2004), Licenciatura em Computação - Claretiano (2021), Técnico em Processamento de Dados - ETFES (1998), Atualmente é professor Titular do Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo (IFES), Coordenador do Curso de Bacharelado em Sistemas de Informação (IFES-Colatina), Coordenador do Laboratório de Estudo e Desenvolvimento de Soluções (Leds-Colatina), Orientador da Equipe de Robótica SEK (Titãs). Tem experiência em Desenvolvimento de Sistemas nas áreas administrativa, financeira, contábil e produção. Temas de estudo: Algoritmos genéticos, Algoritmos evolucionários com múltiplos objetivos, Análise multicritério, Otimização de sequenciamento e balanceamento de linhas de produção, Simulação, Lego mindStorms e Arduino.

Currículo: http://lattes.cnpq.br/7478661826324730

Raphael Magalhães Gomes Moreira

Professor Efetivo do Ifes campus Itapina, docente permanente do Mestrado profissional stricto sensu em Propriedade Intelectual e Transferência de Tecnologia para a Inovação, Coordenador de Programas e Cursos de Pós-graduação e Coordenador Substituto do curso Superior em Agronomia. Atua nos seguintes temas: Mecanização Agrícola > Desenvolvimento de Máquinas Agrícolas, Tecnologia em Aplicação de Agrotóxicos, Ensaios de Máquinas Agrícolas e Florestais, Ergonomia de Máquinas Agrícolas e Florestais. Armazenamento > Inovação e patentes em sistemas agrícolas Projetos de Unidades Armazenadoras. Recursos Hídricos > Iniciante em Projetos de Irrigação e Drenagem, Fertirrigação, Água Residuária de Café, Fertirrigação Potássica e Pós-Colheita do Cafeeiro, iniciante em Projeto de ETEs e ETAs, Conservação do Solo e Água, Iniciante em Projetos de Abastecimento Rural, Iniciante em Projetos de Barragens de Terra, Outorga de Água. Administração Rural. Agroecologia. Legislação Ambiental. Manejo e gestão ambiental da propriedade rural.

Currículo: https://lattes.cnpq.br/6358999333136028