

# Curso de Desenvolvimento Android com Flutter

## 2 - Framework Flutter

**NEMOBIS**

Vinicius Takeo Friedrich Kuwaki



## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

Exercícios

Compilação e Instalação

## ● Introdução

- Até agora vimos códigos que executavam no console;
- Veremos agora como executar nossos códigos em um **dispositivo Android**, seja ele um emulador ou um dispositivo físico;
- Utilizaremos vários **componentes do Flutter** para isso;

## ● Introdução

- Já falamos um pouco do **Flutter** nas aulas passadas;
- Resumindo, o Flutter é um *framework* híbrido que permite ao programador compilar seus códigos **nativamente** para as plataformas:
  - Mobile (Android e iOS);
  - Web;
  - Desktop (Windows, Linux e MacOS);
- Porém, veremos mais a frente que cada uma dessas plataformas possui recursos **exclusivos** a ela, por exemplo, um *Desktop* não possui sensores iguais um celular;

## ● Introdução

- Nessa aula iremos focar em recursos que estão **presentes para todas** as plataformas;
- Embora alguns que veremos só façam sentido seu uso em dispositivos móveis, tal como o **Scaffold**;
- Começaremos criando um **projeto Flutter**;
- Basicamente existem duas formas:
  - Usando o VSCode;
  - Executando o comando **flutter create nome\_projeto**;
- Utilizaremos o VSCode devido a sua simplicidade;

## ● Introdução

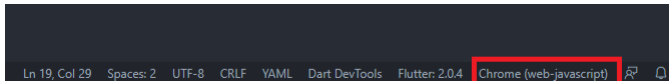
- Com o VSCode aberto, vamos pressionar **Ctrl + Shift + P** para abrirmos a **paleta de comandos**;
- Nela iremos procurar pelo comando: **Flutter: New Application Project**, ou algo similar a **Flutter: Novo Projeto** em português;
- Selecionaremos uma pasta e digitaremos o nome do nosso projeto, novamente usando o padrão **snake\_case** (letras minúsculas e espaço substituídos por underline);
- Ao escolher o nome do projeto, uma nova janela será aberta e o projeto será criado;

## ● Introdução

- Os seguintes arquivos e pastas serão gerados:
  - **android:** pasta contendo algumas configurações para a versão Android do aplicativo;
  - **build:** pasta onde serão gerados os compilados (apk, HTML, etc.);
  - **ios:** pasta contendo algumas configurações para a versão iOS do aplicativo;
  - **lib:** pasta onde iremos escrever os códigos-fontes em Dart;
  - **test:** pasta contendo as classes para a realização de testes no código;
  - **web:** pasta contendo algumas configurações para a versão Web do aplicativo;
  - **pubspec.yaml:** arquivo contendo as dependências do projeto, tais como pacotes importados, imagens externas, áudios, fontes utilizados, etc. Veremos como utilizar a importação de pacotes nas aulas seguintes.

## ● Introdução

- Vamos nos concentrar no momento na pasta **lib**;
- Como dito anteriormente, é nela que iremos escrever os códigos-fontes do projeto;
- Inclusive ao gerar o projeto, um arquivo **main.dart** é gerado também;
- Para executar esse “hello world”, precisamos selecionar a plataforma a ser utilizada, note no **canto inferior direito** ao lado da versão do Flutter **no VSCode**:



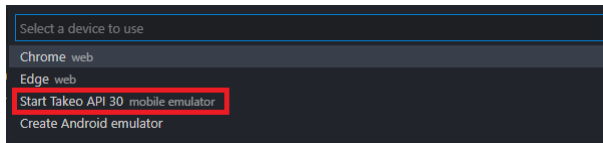
**Figura 1:** Canto inferior direito do VSCode.

- No caso da Figura 1, o dispositivo no qual o código será executado é o **Chrome**;



## ● Introdução

- Ao clicar sobre o dispositivo selecionado, é possível escolher outro para executar o código;
- Uma barra será aberta no campo superior central:

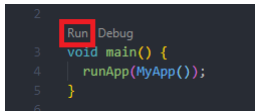


**Figura 2:** Dispositivos disponíveis para executar o código.

- Nela é possível escolher um dispositivo, no caso da Figura 2, o que será selecionado é um **mobile emulator** (emulador Android);

## ● Introdução

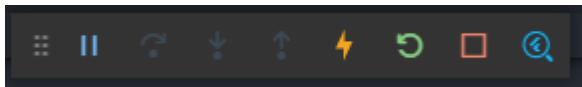
- Ao selecionar um emulador, ele será aberto;
- Caso esteja utilizando um dispositivo físico, é necessário ativar o modo desenvolvedor e ativar a opção **Depuração USB**;
- Para executar o código, vamos acessar o arquivo **main.dart**;
- Todo arquivo que conter uma função **main()** em Dart, se a extensão Flutter/Dart estiver instalada, uma opção de executar (**run**) será exibida:



**Figura 3:** Botão run do VSCode.

## ● Introdução

- Ao executar uma aplicação Flutter no VSCode, no canto superior central (as vezes nos lados), uma barra similar a da Figura 4 será exibida;



**Figura 4:** Barra de execução do Flutter

- Falaremos de cada uma das funcionalidades dessa barra agora;

## ● Introdução

- Da esquerda para a direita:
  - **Arrastar:** o primeiro ícone permite ao usuário arrastar a barra para qualquer outro lugar dentro do VSCode;
  - **Pausar:** o ícone de pause azul permite ao usuário interromper momentaneamente a execução do código e após, retornar se desejar;
  - **Step Over:** ao realizar debug, desce para o próximo comando executado , mesmo que seja em outro escopo;
  - **Step Into:** ao realizar debug, desce para o próximo comando apenas do mesmo escopo
  - **Step Out:** ao realizar debug, encontra o **return** mais próximo;
  - **Hot reload:** apenas recompila e executa, sem perder o valor das variáveis;
  - **Restart:** volta para o ponto de início da aplicação, limpando o estado das variáveis;
  - **Stop:** para a execução do código;
  - **Open Dev Tools:** abre a ferramenta de desenvolvedores para análise do layout dos widgets;

## ● Introdução

- Não abordaremos o uso de **debug** nesse curso e nem o uso da ferramenta [Dart Dev Tools](#);
- Frequentemente estaremos utilizando os atalhos para executar o **Hot Reload**;
- Com a extensão do Flutter instalada no VSCode, basta pressionar **Ctrl + S** e o Hot Reload será executado;

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

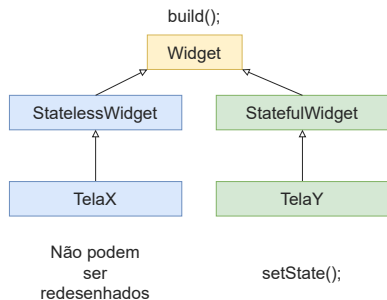
Navegação de Telas

Exercícios

Compilação e Instalação

## ● Classes do Flutter

- Agora que vimos como executar o “hello world” e como selecionar um dispositivo, precisamos ver alguns conceitos básicos do Flutter;
- Começaremos pela **classe Widget**;
- Basicamente em Flutter, a **maioria dos componentes** gráficos que utilizaremos (botões, caixas de textos, etc) **estendem a classe Widget**;
- Veremos com estender a classe Widget para criar nossos próprios;



**Figura 5:** Resumo das classes StatelessWidget e StatefulWidget que utilizaremos.

## ● Classes do Flutter

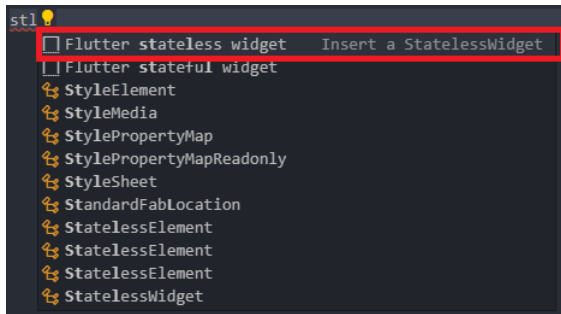
- Basicamente as classes `StatelessWidget` e `StatefulWidget` possuem um método **`build()`**;
- Tal método retorna um **Widget** que será desenhado na tela;
- Além, o método recebe como parâmetro um objeto **BuildContext** que lida com as questões da árvore de *widgets* do Flutter;
- No Flutter iremos agrupar **widgets** dentro de **widgets**, logo, eles podem ser representados como uma estrutura de árvore;
- Não entraremos tanto nesses detalhes;
- É possível também enxergar o objeto **BuildContext** como sendo uma região de tela ao qual serão desenhados os recursos;



## ● Classes do Flutter - StatelessWidget

- Começaremos agora vendo a classe **StatelessWidget**;
- Utilizaremos ela para criar novas telas e/ou *widgets* próprios (agrupando outros já existentes do Flutter);
- Para criar uma classe que estende a StatelessWidget, basta usar a palavra **extends**;
- No VSCode é possível utilizar um **snippet** para criar uma classe que estende a StatelessWidget;
- Basta criar um novo arquivo **.dart** e digitar **stl**;
- O VSCode então irá sugerir criar um StatelessWidget;

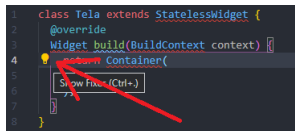
## ● Classes do Flutter - StatelessWidget



**Figura 6:** Snippet para criar classes que estendem a classe StatelessWidget.

## ● Classes do Flutter - StatelessWidget

- Com o arquivo criado, basta clicar sobre os erros sublinhados de vermelho pelo VSCode e clicar na lanterna.



**Figura 7:** Lanterna para corrigir erros no VSCode.

- Recomenda-se utilizar ao máximo a lanterna do VSCode, pois além de corrigir erros, ela auxiliar também na refatoração do código;

## ● Classes do Flutter - StatelessWidget

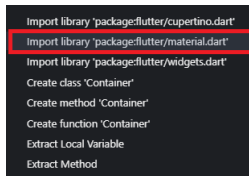


Figura 8: Imports sugeridos.

- Em seguida, vamos importa o arquivo **material.dart**, é ele quem contém componentes que utilizaremos para todas as plataformas;

## ● Classes do Flutter - StatelessWidget

- Feito isso, temos uma classe que estende a classe StatelessWidget:

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

class Tela extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

- Note que ela retorna um Container, falaremos dele mais a frente;

## ● Classes do Flutter - StatelessWidget

- As classes que estenderem a classe StatelessWidget apenas possuirão a propriedade de serem **Widget's**;
- São classes que não podem ser redesenhadas, isto é, desde a sua criação até a sua destruição serão iguais;
- Como o próprio nome já diz, a classe não possui um estado;
- Diferentemente da **StatefulWidget** que veremos em seguida;

## ● Classes do Flutter - StatefulWidget

- Como o próprio nome já diz, a classe **StatefulWidget** possui vários estados;
- De fato, tal classe permite que seus componentes sejam redesenhados a partir da chamada de um método especial;
- Ela é composta de duas classes, mas veremos ela como sendo uma única;
- Para criar ela no VSCode, utilizaremos o *snippet* **stf**;
- Serão geradas duas classes:
  - Uma estende a classe StatefulWidget;
  - A outra estende a classe **State**;

## ● Classes do Flutter - StatefulWidget

```
class NOME extends StatefulWidget {  
  
  @override  
  _NOMEState createState() => _NOMEState();  
}  
  
class _NOMEState extends State<NOME> {  
  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
}
```



## ● Classes do Flutter - StatefulWidget

- Basicamente por estender a classe **State**, ela possui alguns métodos que utilizaremos/sobreescreveremos:
  - **initState()**: define uma série de comandos a serem executados juntamente a criação do objeto;
  - **dispose()**: define uma série de comandos a serem executados quando o objeto for destruído;
  - **setState(VoidCallback funcao)**: executa a função passada como parâmetro redesenhando a tela;
- Gerenciar o estado de uma aplicação é uma das tarefas mais árduas do Flutter;
- A comunidade desenvolveu alguns [pacotes](#) para auxiliar nessa tarefa;
- Nesse curso utilizaremos apenas o **setState()** e métodos síncronos/assíncronos;

## ● Classes do Flutter - StatefulWidget

- É interessante comentar que uma classe **StatelessWidget** pode ser convertida para uma classe **StatefulWidget**;
- Mas não o contrário;
- Para isso, basta colocar o cursor do mouse sobre a palavra **StatelessWidget** e clicar na lanterna;
- Uma opção “**Convert to StatefulWidget**” irá aparecer;
- No slide seguinte veremos como utilizar a funções da classe **State**:

## ● Classes do Flutter - StatefulWidget

```
class _TelaState extends State<Tela> {  
  @override  
  void initState() {  
    super.initState();  
    print('executado apos a criacao');  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
  
  @override  
  void dispose() {  
    super.dispose();  
    print('executado apos a destruicao do objeto');  
  }  
}
```

## ● Classes do Flutter - StatefulWidget

- Note que apenas a “parte do estado da classe” foi exibida no código.
- Em seguida veremos uma função que incrementa uma variável inteira e redesenha a tela:

```
class _TelaState extends State<Tela> {  
  int x = 3;  
  
  void incrementaRedesenhando() {  
    setState(() {  
      x++;  
    });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
}
```

## ● Classes do Flutter - StatefulWidget

- Se algum componente estiver desenhando o valor de x na tela, esse valor será atualizado quando o método **incrementaRedesenhando()** for executado;
- Outro ponto interessante de comentar sobre classes que estender a classe **StatefulWidget** é a questão do acesso de variáveis;
- As variáveis podem ser declaradas em ambas as classes (que estende a StatefulWidget ou que estende a State);
- Em certos cenários, precisaremos acessar variáveis que estão na classe de cima (que estende a StatefulWidget);
- Para isso, podemos acessar tais variáveis usando a propriedade **widget**:

## ● Classes do Flutter - StatefulWidget

```
class Tela extends StatefulWidget {  
  
  int x;  
  
  @override  
  _TelaState createState() => _TelaState();  
}  
  
class _TelaState extends State<Tela> {  
  
  void zeraX() {  
    widget.x = 0;  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
}
```

## ● Classes do Flutter

- Agora que vimos os principais conceitos, estamos prontos para ver alguns componentes nativos do Flutter;
- Para executar uma aplicação Flutter, utilizaremos a função **runApp()** na **main()** passando uma instancia do componente a ser exibido na plataforma;

```
void main() {  
  runApp(MyApp());  
}
```

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

**Widgets Básicos**

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

Exercícios

Compilação e Instalação



## ● Widgets Básicos

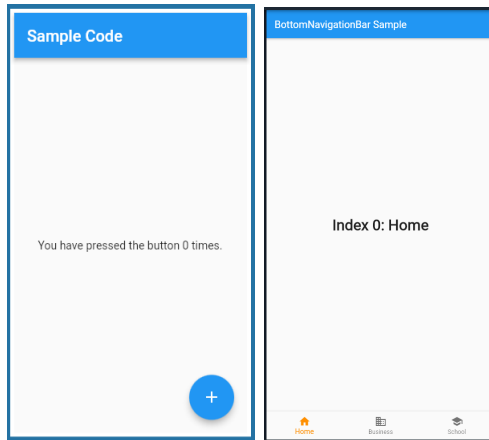
- Veremos alguns componentes básicos primeiro:
  - MaterialApp (slide 34);
  - Scaffold (slide 37);
  - AppBar (slide 42);
  - Container (slide 45);
  - Text (slide 51);
  - Icon (slide 54);
  - IconButton (slide 55);
  - ElevatedButton (slide 59);
  - FloatingActionButton (slide 60);
  - TextField (slide 63);
  - Image (slide 76);

## ● Widgets Básicos - MaterialApp

- Começaremos pelo `MaterialApp`;
- O widget basicamente agrupa recursos que são muito utilizados no Material Design;
- Nele é possível definir cores padrões, animações, rotas, etc;
- Não entraremos em detalhes nessa parte;
- Veremos apenas o uso de alguns atributos de seu construtor:
  - **`bool debugShowCheckedModeBanner`**: define se o banner de debug irá aparecer na aplicação;
  - **`Widget home`**: define a página inicial da aplicação;
  - **`ThemeData theme`**: define o tema padrão do aplicativo;
  - **`String title`**: uma descrição de uma linha para o dispositivo identificar o app.

## ● Widgets Básicos

- É normal termos apenas um `MaterialApp` em uma aplicação;
- Normalmente deixamos ela no próprio arquivo `.main`;
- Antes de vermos um exemplo de seu uso, vamos falar da classe `Scaffold`;



**Figura 9:** Exemplos de Scaffold.

## ● Widgets Básicos - Scaffold

- A classe **Scaffold** possui alguns atributos no seu construtor que utilizaremos:
  - **PreferredSizeWidget appBar:** define uma barra para a tela. Veremos o objeto AppBar em seguida;
  - **Color backgroundColor:** define uma cor de fundo. Para isso existe um enumerável **Colors** contendo várias cores já pré-definidas;
  - **Widget body:** define os widgets que irão compor a parte abaixo da barra.
  - **Widget bottomNavigationBar:** uma barra de navegação na parte inferior da tela, tal como a imagem da direita na Figura 9.
  - **Widget drawer:** abre um **menu de sanduíche** ao lado da tela;
  - **Widget floatingActionButton:** exibe um botão flutuante na tela (padrão é o canto inferior direito);
  - **FloatingActionButtonLocation floatingActionButtonLocation:** define a localização do botão flutuante.

## ● Widgets Básicos - Scaffold

- Criaremos agora a nossa primeira tela, estendendo a classe `StatefulWidget`;
- Dentro do método **`build()`**, vamos retornar um objeto do tipo **Scaffold**;
- A princípio vamos apenas colocar um `Container` no atributo **`body`**:

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

class Tela extends StatefulWidget {
  @override
  _TelaState createState() => _TelaState();
}

class _TelaState extends State<Tela> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(),
    );
  }
}
```

## ● Widgets Básicos - Scaffold

- Agora podemos criar uma classe App (em um arquivo tela.dart) que irá estender a classe **StatelessWidget** para colocarmos um **MaterialApp**;
- Como não precisaremos redesenhar essa tela, ela pode ser um **StatelessWidget** mesmo;
- No parâmetro **home**, vamos instanciar um novo objeto da classe Tela que criamos para o Scaffold:

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Tela(),  
    );  
  }  
}
```

## ● Widgets Básicos - Scaffold

- Vamos adicionar um **title** para a aplicação:

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Tela(),  
      title: 'Aula 2 — Flutter',  
    );  
  }  
}
```



## ● Widgets Básicos - Scaffold

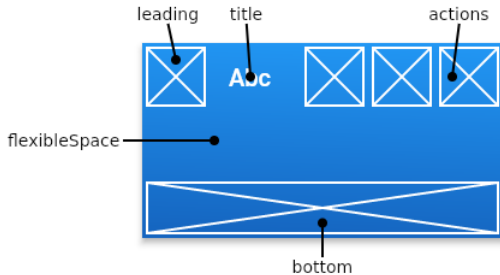
- Para rodar a aplicação, vamos chamar o método **runApp()** na função **main()** passando uma instancia da classe App:

```
void main(){  
  runApp(App());  
}
```

- Não esqueça de fazer todos os *imports* (utilizando a lanterna do VSCode) para que o código possa ser executado.

## ● Widgets Básicos - AppBar

- Agora vamos ver a classe [AppBar](#) que pode ser colocada no atributo de mesmo nome do **Scaffold**;



**Figura 10:** Algumas propriedades da classe AppBar.

## ● Widgets Básicos - AppBar

- A classe possui alguns atributos que vale mencionar (veja a Figura 10):
  - **List<Widget> actions:** uma lista de widgets que ficarão na região da direita da AppBar.
  - **Color backgroundColor:** a cor de fundo da AppBar;
  - **bool centerTitle:** define se o título será centralizado ou alinhado com a esquerda;
  - **Widget leading:** define um widget para ficar na esquerda do título;
  - **Widget title:** define um widget para representar o título, podendo ser um texto, uma imagem, etc.

## ● Widgets Básicos - AppBar

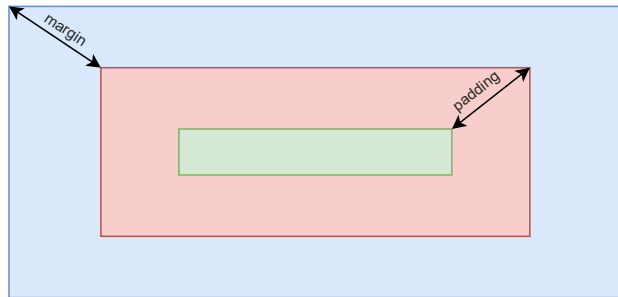
- Continuando o código do Scaffold que fizemos nos slides anteriores, vamos adicionar uma AppBar no atributo **appBar**;
- Nela vamos colocar um texto de título, utilizaremos a classe Text que veremos em detalhes mais a frente;
- Para isso, passaremos uma String para o construtor da classe Text;
- Além de definirmos a cor de fundo da AppBar como sendo roxa (valor Purple do enumerável Colors);

```
appBar: AppBar(  
  title: Text('Exemplo de tela'),  
  centerTitle: true,  
  backgroundColor: Colors.purple,  
),
```

## ● Widgets Básicos - Container

- Agora veremos a classe Container que já utilizamos anteriormente;
- A classe basicamente representa uma caixa que pode abrigar outros widgets dentro;
- Definindo algumas propriedades de espaçamento, cor, bordas, etc.
- Duas das propriedades mais interessantes e que utilizaremos muito nos layouts são as propriedades **margin** e **padding**;

## ● Widgets Básicos - Container



**Figura 11:** Container em vermelho, destacando o que representa as propriedades **margin** e **padding** do Container.

## ● Widgets Básicos - Container

- Na Figura 11 é possível ter uma noção visual de tais atributos;
- Basicamente o **margin** define o espaçamento do Container com o seu widget pai;
- Enquanto que o **padding** define o espaçamento entre o widget filho e ele;
- Veremos em seguida os principais atributos da classe que podem ser passados ao construtor;

## ● Widgets Básicos - Container

- Na documentação da classe [Container](#) é possível ter a acesso a lista completa de atributos e métodos, veremos alguns:
  - **Widget child:** define o widget filho do Container (o widget que ficará dentro dele);
  - **Color color:** define a cor de fundo do Container;
  - **Decoration decoration:** define uma decoração para o Container (veremos mais a frente esse objeto);
  - **double height:** define a altura em pixels do Container;
  - **EdgeInsetsGeometry margin:** define o espaçamento da margem usando um objeto [Edge Insets](#);
  - **EdgeInsetsGeometry padding:** define o espaçamento da margem usando também um objeto [EdgeInsets](#);
  - **double width:** define a largura em pixels do Container;
- No layout mobile, serão raras as vezes em que iremos definir algum widget em função de sua altura/largura em pixels;



## ● Widgets Básicos - EdgInsets

- A classe `EdgInsets` possui alguns construtores que utilizaremos:
  - **`all(double valor)`**: define um espaçamento em todas as direções;
  - **`only(double left: 0.0, double top: 0.0, double right: 0.0, double bottom: 0.0)`**: define espaçamentos personalizados para cada uma das quatro direções;
  - **`symmetric(double vertical: 0.0, double horizontal: 0.0)`**: define o espaçamento eixo a eixo (x ou y);

## ● Widgets Básicos - Container

- Para exemplificar, vamos retornar ao **body** do Scaffold e adicionar dentro do Container, seu filho (um Text), espaçamentos na margem (10 em todos os lados) e no padding (10 em cima e 15 na direita);
- Além da cor de fundo como sendo cinza:

```
body: Container(  
  margin: EdgeInsets.all(10),  
  padding: EdgeInsets.only(top: 10, right: 15),  
  child: Text('Hello World!'),  
  color: Colors.grey,  
),
```

## ● Widgets Básicos - Text

- Agora que usamos a classe `Text`, vamos ver algumas propriedades que ela possui;
  - **String data:** parâmetro obrigatório contendo o texto a ser desenhado;
  - **int maxLines:** define o número máximo de linhas que o texto pode ter;
  - **TextOverflow overflow:** define o que acontece quando o texto ultrapassa o tamanho máximo do widget. Utiliza um enumerável `TextOverflow`;
  - **TextStyle style:** define algumas propriedades de layout do texto, veremos em detalhes em seguida;
  - **TextAlign textAlign:** enumerável que define o alinhamento do texto na horizontal com relação ao espaço que ele possui para ser desenhado;
  - **TextDirection textDirection:** enumerável que define a direção de escrita do texto;

## ● Widgets Básicos - Text Style

- Já a classe **Text Style** possui um número muito maior de propriedades, veremos apenas algumas básicas:
  - **Color backgroundColor:** cor de fundo do texto;
  - **Color color:** cor do texto;
  - **double fontSize:** define o tamanho da fonte;
  - **fontStyle FontStyle:** enumerável que define o estilo da fonte (itálico, normal, etc);
  - **fontWeight FontWeight:** define a grossura do texto (use o valor Bold para negrito);
  - **double letterSpacing:** define o espaçamento entre as letras;

## ● Widgets Básicos - Text

- Para exemplificar os conceitos vistos, vamos voltar a alterar o Text que colocamos dentro do **body**:

```
child: Text(  
  'Hello World!',  
  style: TextStyle(  
    fontSize: 20,  
    color: Colors.white,  
  ),  
  overflow: TextOverflow.ellipsis,  
),
```

- Note que colocamos o tamanho da fonte como 20 e na cor branca, além de definirmos que quando faltar espaço para o texto, reticências serão colocadas.

## ● Widgets Básicos - Icon

- Agora antes de utilizarmos os botões do Flutter, vamos ver um classe para desenhar ícones;
- Tal classe se chama **Icon**;
- Assim como para as cores, o Flutter possui alguns ícones já pré-definidos na interface do Material, para isso, utilizaremos a classe **Icons**;
- A classe **Icon** possui uma atributo obrigatório do tipo **IconData** e a classe **Icons** é uma das classes que estende essa classe;
- Alguns dos atributos que mais utilizaremos da classe **Icon**:
  - **IconData icon**: define o ícone a ser exibido;
  - **Color color**: define a cor do ícone;

## ● Widgets Básicos - IconButton

- Seu uso é muito comum em conjunto com a classe **IconButton** que veremos em seguida;
- A classe `IconButton` é um botão que utiliza um ícone;
- Os principais atributos que utilizaremos são:
  - **Color color:** define a cor do botão;
  - **Widget icon:** define um widget para ser exibido;
  - **VoidCallback onPressed:** define uma função para ser executada quando o botão for pressionado. Tal função deve ser do tipo `void` ou `Future<void>` e não pode receber parâmetros;
  - **String tooltip:** define uma dica que irá aparecer quando o usuário segurar o botão no mobile ou passar o mouse por cima na web/desktop;

## ● Widgets Básicos - IconButton

- Para exemplificar seu uso, vamos construir a lógica do exemplo do contador do Flutter;
- Começaremos definindo um atributo inteiro “contador” que começará em 0;
- Em seguida, definiremos um método privado **`_incrementa()`** que incrementa o valor do contador e redesenha a tela:

```
class _TelaState extends State<Tela> {  
  int contador = 0;  
  
  @override  
  Widget build(BuildContext context) {  
    ...  
  }  
  
  _incrementa() {  
    setState(() {  
      contador++;  
    });  
  }  
}
```



## ● Widgets Básicos - IconButton

- Voltaremos agora para o nosso Text que está dentro do Scaffold e vamos fazer com que ele desenhe o valor do contador:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      margin: ...,
      padding: ...,
      child: Text(
        '$contador',
        style: ...
        overflow: ...,
      ),
      color: ...,
    ),
    appBar: ...
  );
}
```

## ● Widgets Básicos - IconButton

- Agora, no atributo **leading** da AppBar, vamos adicionar um IconButton;
- Seu ícone será a constante “Add” e sua função **onPressed** será a função **\_incrementa()** que criamos a pouco:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ...,
    appBar: AppBar(
      title: Text('Exemplo de tela'),
      centerTitle: true,
      backgroundColor: Colors.purple,
      leading: IconButton(
        icon: Icon(Icons.add),
        onPressed: _incrementa,
      ),
    ),
  );
}
```

## ● Widgets Básicos - ElevatedButton

- Note um detalhe bem importante, estamos passando a referência da função **`_incrementa()`** e não chamando-a;
- Com isso vemos o primeiro tipo de botão que podemos usar no Flutter;
- Em seguida veremos a classe **`ElevatedButton`**;
- Tal classe também é um botão, inclusive ao passar um objeto `Icon` para seu atributo **`child`**, ela “meio” que se tornará um;
- Porém, é mais comum utilizar ela com textos ou imagens;
- Ela possui dois atributos que mais utilizaremos:
  - **Widget child:** define o que será exibido no botão;
  - **VoidCallback onPressed:** define uma função para ser executada quando o botão for pressionado. Tal função deve ser do tipo `void` ou `Future<void>` e não pode receber parâmetros;

## ● Widgets Básicos - FloatingActionButton

- O terceiro tipo de botão que veremos é um botão muito comum em interfaces para Android;
- A classe `FloatingActionButton` possui vários atributos similares aos que vimos nos outros botões;
- Seu uso é comum no Scaffold;
- Alguns de seus atributos:
  - **Color backgroundColor:** a cor de fundo do botão;
  - **Widget child:** o widget que irá aparecer no botão;
  - **VoidCallback onPressed:** a função a ser executada quando o botão for pressionado;

## ● Widgets Básicos - FloatingActionButton

- Como mencionamos, para utilizarmos, colocaremos ela no atributo **floatingActionButton** do Scaffold;
- Utilizaremos a mesma função que já havíamos definido para o exemplo do IconButton:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ...
    appBar: ...
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementa,
      child: Icon(Icons.add),
    ),
  );
}
```

## ● Widgets Básicos - FloatingActionButtonLocation

- Agora que temos um FloatingActionButton, vamos utilizar a classe `FloatingActionButtonLocation` para alterarmos sua localização no Scaffold;
- Colocaremos ele centralizado;

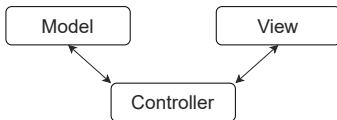
```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ...,
    appBar: ...,
    floatingActionButton: ...,
    floatingActionButtonLocation: FloatingActionButtonLocation.centerFloat,
  );
}
```

- Lembre-se que a documentação da classe `FloatingActionButtonLocation` mostra todos os valores possíveis do enumerável, assim como a extensão do Flutter no VSCode;

## ● Widgets Básicos - TextField

- O widget que veremos em seguida, faz a entrada de dados do usuário;
- Utilizaremos a classe `TextField` para isso;
- Existe também a classe `TextFormField` para criar formulários, porém iremos abordar apenas a `TextField`;
- Tal classe trabalha com um conceito chamado **MVC**, que consiste em dividir as classes de acordo com as suas funcionalidades:
  - **Model**: classes que representam entidades (pessoas, animais, etc);
  - **View**: classes que compõe a interface gráfica (widgets, telas, etc);
  - **Controller**: classes que compõe a lógica de negócio e fazem a “ponte” entre o modelo e a view;

## ● Widgets Básicos - TextField



**Figura 12:** Diagrama representando a estrutura de um MVC.

- A classe `TextField` em conjunto com a `TextEditingController` utiliza esse conceito e também o utilizaremos quando formos construir nossas aplicações;
- Basicamente o padrão serve para “organizar” o código;
- Veremos-o em detalhes mais a frente.



## ● Widgets Básicos - TextField

- A classe `TextField` possui alguns atributos que utilizaremos:
  - **`bool autocorrect`**: atributo que define se a autocorreção está ligada ou não;
  - **`TextEditingController controller`**: define o controller que irá armazenar o valor do texto digitado;
  - **`TextInputType keyboardType`**: define o tipo do teclado (numérico, email, etc.);
  - **`int maxLength`**: número máximo de letras permitidas;
  - **`bool obscureText`**: define se o texto será ocultado ou não, por exemplo senhas;
  - **`ValueChanged<String> onSubmitted`**: define uma função (que recebe uma `String` e retorna `void`) a ser executada quando o usuário digitar ENTER;
  - **`TextStyle style`**: define o estilo do texto, tal como na classe `Text`;
  - **`TextAlign textAlign`**: define o alinhamento do texto, tal como na classe `Text`;

## ● Widgets Básicos - TextEditingController

- Como utilizaremos a classe `TextEditingController`, basicamente utilizaremos um atributo dessa classe:
  - **String text:** acessa o valor atual do controller;
- Também utilizaremos um método dela:
  - **clear():** limpa a String salva no controller;

## ● Widgets Básicos - TextField

- A fim de exemplo, faremos vamos colocar um **TextField** que quando o **FloatingActionButton** for pressionado, o valor do **title** da **AppBar** será substituído pelo valor lido do usuário;
- Precisamos de um objeto do tipo **TextEditingController** para armazenar esse valor, para isso, vamos colocá-lo como atributo da classe, visto que ele será acessado pelo **TextField** e pela **AppBar**;
- Daremos o nome a esse atributo de **caixaTexto**:

```
class _TelaState extends State<Tela> {  
  ..  
  TextEditingController caixaTexto = TextEditingController();  
  
  @override  
  Widget build(BuildContext context) {  
    return ...;  
  }  
}
```

## ● Widgets Básicos - TextField

- Criaremos também uma String chamada **titulo**, onde iremos armazenar o valor do titulo;

```
class _TelaState extends State<Tela> {  
  ..  
  TextEditingController caixaTexto = TextEditingController();  
  String titulo = "";  
  
  @override  
  Widget build(BuildContext context) {  
    return ...;  
  }  
}
```

## ● Widgets Básicos - TextField

- Usaremos tal variável **titulo** no campo **title** da **AppBar**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ...,
    appBar: AppBar(
      title: Text(titulo),
      centerTitle: true,
      backgroundColor: Colors.purple,
      leading: ...,
    ),
    floatingActionButton: ...,
    floatingActionButtonLocation: ...,
  );
}
```

## ● Widgets Básicos - TextField

- Agora, faremos uma função que utilizaremos no botão do **FloatingActionButton** para que quando o botão for pressionado, a caixa seja limpa e a tela redesenhada, substituindo o valor do atributo **titulo** pelo valor lido do usuário no controller;
- Chamaremos tal método de **\_novoTitulo()**:

```
class _TelaState extends State<Tela> {  
  ...  
  @override  
  Widget build(BuildContext context) {  
    return ...;  
  }  
  
  _novoTitulo() {  
    setState(() {  
      titulo = caixaTexto.text;  
      caixaTexto.clear();  
    });  
  }  
}
```

## ● Widgets Básicos - TextField

- Agora, vamos substituir a função **onPressed** do **FloatingActionButton**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ...,
    appBar: ...,
    floatingActionButton: FloatingActionButton(
      onPressed: _novoTitulo,
      child: Icon(Icons.add),
    ),
    floatingActionButtonLocation: ...,
  );
}
```

## ● Widgets Básicos - TextField

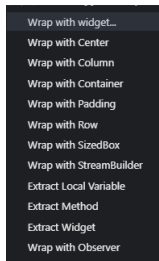
- Para finalizar, precisamos colocar a **TextField** no **body**, colocando a **caixaNome** como sendo o parâmetro **controller**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: TextField(
      controller: caixaTexto,
    ),
    appBar: ...,
    floatingActionButton: ...,
    floatingActionButtonLocation: ...,
  );
}
```



## ● Widgets Básicos

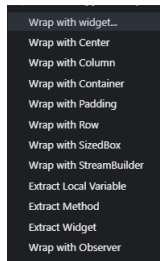
- Logo, finalizamos o exemplo.
- Vamos apenas alterar uma questão no layout;
- A `TextField` nasce colada em todos os cantos da tela;
- Vamos adicionar um **padding** nela;
- Para isso, precisamos adicionar um **Container** e um atributo **padding**;
- Para colocar o **TextField** dentro de um **Container**, basta colocar o cursor do mouse sobre o **TextField** e clicar na lanterna do VSCode;



**Figura 13:** Auxiliar do VSCode.

## ● Widgets Básicos

- Note que existem uma série de opções, as principais são:
  - **Wrap with ALGO:** tal comando “envolve” o widget atual com ALGO;
  - **Extract Local Variable:** cria um atributo que será igual ao objeto selecionado
  - **Extract Method:** cria um método (será solicitado a escolha do nome) que retorna tal objeto
  - **Extract Widget:** cria uma nova classe (será solicitado a escolha do nome) que seu método **build()** retornará tal objeto;



**Figura 14:** Auxiliar do VSCode.

## ● Widgets Básicos

- Utilizaremos o **Wrap with Container**;
- Após, vamos adicionar o padding de 10 em todas as direções usando o **EdgeInsets.all()**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      padding: EdgeInsets.all(10),
      child: TextField(
        controller: caixaTexto,
      ),
    ),
    appBar: ...,
    floatingActionButton: ...,
    floatingActionButtonLocation: ...,
  );
}
```

## ● Widgets Básicos - Image

- Para fechar a parte dos widgets básicos, vamos ver a classe `Image`;
- Utilizaremos ela para desenhar imagens no Flutter;
- Vamos utilizar um de seus quatro construtores nomeados:
  - **`Image.asset(String path)`**: cria um objeto do tipo `Image` a partir de um *asset* definido no **`pubspec.yaml`**;
  - **`Image.file(File file)`**: cria um objeto do tipo `Image` a partir de um objeto `File`;
  - **`Image.memory(Uint8List bytes)`**: cria um objeto do tipo `Image` a partir de uma lista de bytes (essa é a forma que utilizaremos para acessar fotos da câmera);
  - **`Image.network(String src)`**: cria um objeto do tipo `Image` a partir da url de uma foto da internet;
- No momento utilizaremos apenas o construtor **`network()`**, na aula seguinte veremos o uso do **`memory()`**;

## ● Widgets Básicos - Image

- A classe também possui alguns atributos que utilizaremos:
  - **BoxFit fit:** determina como a imagem será cortada/ajustada com relação ao espaço disponível para ela ser desenhada;
  - **double height:** define em pixels a altura da imagem;
  - **double width:** define em pixels a largura da imagem;

## ● Widgets Básicos - Image

- A fim de exemplos, vamos substituir o **TextField** que tínhamos por uma Image;
- Utilizaremos uma foto de Joinville:

[https://files.nsctotal.com.br/s3fs-public/graphql-upload-files/centro%20de%20Joinville\\_59.jpg?k6V.e9Jc8lWNdmps\\_OyeQNng0kTEaJ4l](https://files.nsctotal.com.br/s3fs-public/graphql-upload-files/centro%20de%20Joinville_59.jpg?k6V.e9Jc8lWNdmps_OyeQNng0kTEaJ4l):

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      padding: EdgeInsets.all(10),
      child: Image.network(
        'https://files.nsctotal.com.br/s3fs-public/graphql-upload-files/centro%20de%20Joinville_59.jpg?k6V.e9Jc8lWNdmps_OyeQNng0kTEaJ4l'),
    ),
    appBar: ...,
    floatingActionButton: ...,
    floatingActionButtonLocation: ...,
  );
}
```

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

**Exercícios**

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

Exercícios

Compilação e Instalação

## ● Exercícios

1. Crie um projeto no Flutter e execute o “Hello World”;
2. Crie uma classe Aplicativo, que estende a classe **StatelessWidget** (slide 17) e contém um **MaterialApp** (slide 34);
3. Crie uma classe TelaInicial que estende a classe **StatefulWidget** (slide 23) e contém um Scaffold (slide 37). Adicione ao Scaffold uma AppBar (slide 42) contendo uma Text (slide 51) centralizado informando o nome da tela: “Tela Inicial”.
4. Na classe TelaInicial do exercício anterior, adicione um IconButton (slide 55), contendo um ícone (slide 54) de sua escolha e uma função que faça com que o título hora apareça, hora desapareça;



## ● Exercícios

5. Crie uma classe Tela2 contendo um Scaffold com dois IconsButtons na AppBar no atributo **actions**, além de um Text contendo um contador no **body** do Scaffold, um dos botões da AppBar deve incrementar o valor do contador e o outro deve decrementar o valor do contador. Altere a classe MaterialApp dos exercícios anteriores para que o atributo **home** agora seja uma instancia da Tela2;
6. Adicione na Tela2 um FloatingActionButton na esquerda que deve zerar o valor do contador.
7. Adicione um TextField com um teclado numérico no atributo **title** da Tela2. Altere os botões que incrementam e decrementam, para que usem o valor da TextField para aumentar ou reduzir o valor do contador;

8. Crie uma classe Tela3, substitua-a no MaterialApp do item anterior. A classe Tela3 deve conter apenas um botão e um Container no **body** do Scaffold, tal botão deve alternar a cor do Container a partir de um grupo definido de cores (utilize ao menos três cores). No título da AppBar deve-se estar escrito o nome da cor que está pintado o **body**;

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

**Widgets de Layout**

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

Exercícios

Compilação e Instalação

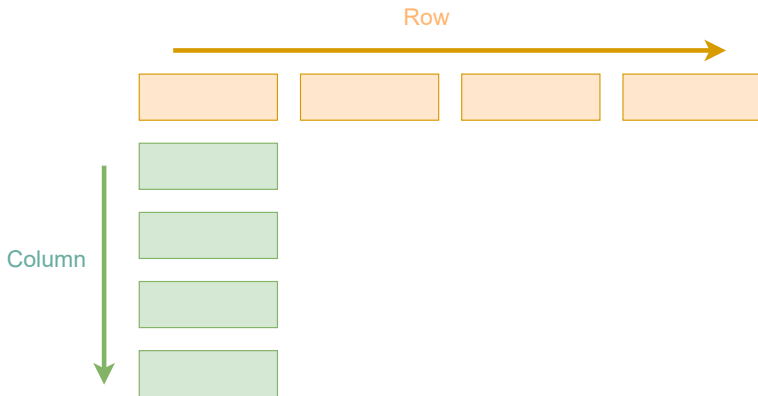
## ● Widgets de Layout

- Após vermos alguns widgets que o Flutter possui para exibir coisas básicas como textos, imagens, caixas de textos, botões e etc, vamos ver como agrupar tais componentes na tela;
- Para isso, veremos as classes:
  - Column;
  - Row;
  - ListView;
  - Expandend;
  - Flexible;
  - Center;

## ● Widgets de Layout

- Alguns dos widgets que veremos, como o Column, o Row e o ListView, permitem que coloquemos múltiplos widgets, lado a lado;
- A partir disso, podemos criar *designs* com mais componentes e mais dinâmicos;
- Na Figura 15 é possível ver uma comparação entre o Column e o Row.

## ● Widgets de Layout



**Figura 15:** Comparação entre o Row e Column em função da orientação em que os widgets serão agrupados.

## ● Widgets de Layout

- Note que enquanto o Row agrupa widgets na horizontal (esquerda para a direita), o Column agrupa-os na vertical (cima para baixo);
- Veremos agora características dessas duas classes;

## ● Widgets de Layout - Column

- Começaremos pela classe **Column**;
- Alguns de seus atributos que utilizaremos:
  - **List<Widget> children**: uma lista de widgets que estarão dentro do Column;
  - **CrossAxisAlignment crossAxisAlignment**: enumerável que representa o alinhamento dos widgets no eixo cruzado (esquerda para direita);
  - **MainAxisAlignment mainAxisAlignment**: enumerável que representa o alinhamento dos widgets no eixo principal (cima para baixo);
- Note que agora não há um atributo **child** igual vimos em outros widgets;
- Agora um Row possui uma lista de Widget em seu atributo **children**;
- Note também que nada impede a existência de uma Row dentro de um Column e assim por diante;



## ● Widgets de Layout - Column

- O grande problemas que enfrentamos agora é o espaço;
- Quando não houver mais espaço na tela, simplesmente uma mensagem de erro na própria interface gráfica será exiba;
- Não é nem possível tratá-lo via exceções!
- Veremos um solução para isso a frente;

## ● Widgets de Layout - Column

- A fim de exemplos, vamos criar um Column e agrupar uma série Container's coloridos dentro dele, todos contendo uma altura de 100 pixels;
- Começaremos limpando nossa classe **Tela** utilizada anteriormente, deixando apenas o Scaffold com uma AppBar e um Container;

```
class _TelaState extends State<Tela> {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Container(),  
      appBar: AppBar(  
        title: Text('Exemplos de Layouts'),  
        centerTitle: true,  
        backgroundColor: Colors.purple,  
      ),  
    );  
  }  
}
```

## ● Widgets de Layout - Column

- Agora, no **body** vamos declarar um **Column** contendo quatro Container's:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Column(
      children: [
        Container(color: Colors.red, height: 100),
        Container(color: Colors.blue, height: 100),
        Container(color: Colors.green, height: 100),
        Container(color: Colors.yellow, height: 100),
      ],
    ),
    appBar: ...,
  );
}
```

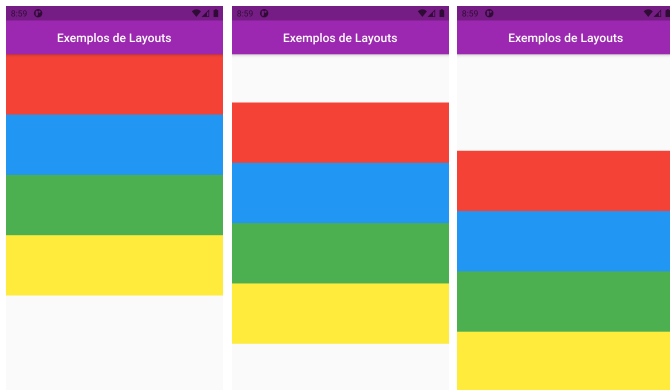
## ● Widgets de Layout - Column

- Dependendo do número de polegadas de seu dispositivo/emulador, o exemplo irá acusar um **overflow**;
- Tal **overflow** ocorre quando falta espaço na tela para desenhar;
- No dispositivo utilizado para testar os código dessa aula, ao adicionar seis dos Container's apresentados no trecho de código anterior, um **overflow** ocorreu;



**Figura 16:** Exemplo de **overflow** ao incluir mais dois Container's coloridos ao Column do código anterior.

## ● Widgets de Layout - Column



**Figura 17:** Exemplo de layout utilizando o atributo **mainAxisAlignment**, da esquerda para direita, o atributo assume os valores: `MainAxisAlignment.start`, `MainAxisAlignment.center` e `MainAxisAlignment.end`.

## ● Widgets de Layout - Row

- A classe `Row` possui os mesmos atributos comentados nos slides anterior do `Column`, a grande diferença é a inversão do **`mainAxisAlignment`** com o **`crossAxisAlignment`**;
- Como o **`mainAxisAlignment`** representa o alinhamento no eixo principal, tal atributo na classe **`Row`** determina o alinhamento dos widgets da esquerda para a direita;
- Enquanto que o **`crossAxisAlignment`** determina o alinhamento de cima para baixo;

## ● Widgets de Layout - Row

- Adaptando o exemplo anterior, vamos adicionar **Container's** coloridos de largura 50 em um **Row**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Row(
      children: [
        Container(color: Colors.red, width: 50),
        Container(color: Colors.blue, width: 50),
        Container(color: Colors.green, width: 50),
        Container(color: Colors.yellow, width: 50),
      ],
    ),
    appBar: ...
  )
}
```

## ● Widgets de Layout - ListView

- Agora, de forma a “contornar” o problema que comentamos anteriormente, do **overflow**, vamos ver a classe [ListView](#);
- Tal classe permite criarmos **Column's** ou **Row's** que cresçam de acordo com a necessidade de tamanho dos widgets filhos;
- É comum dizer que o ListView desenha widgets sobre “demanda”.
- Nesse curso, utilizaremos dois de seus construtores:
  - **ListView(List<Widget> childrens)**: recebe uma lista contendo seus filhos;
  - **ListView.builder(int itemCount, IndexedWidgetBuilder builder)**: recebe a quantidade de elementos a serem desenhados e uma função para desenhar o i-ésimo widget;



## ● Widgets de Layout - ListView

- Como o construtor padrão **ListView()** é análogo ao construtor do **Column** e do **Row**, veremos apenas o uso do **ListView.builder()** no momento;
- É possível fazer uma analogia desse construtor com o laço **for**;
- Tal construtor recebe uma condição de parada, o atributo **itemCount**;
- E possui uma função que retorna o widget da **i-ésima** posição;
- Tal função deve receber um **BuildContext** e um **int** que representa a i-ésima posição;

## ● Widgets de Layout - ListView

- Para exemplificarmos seu uso, vamos adicionar uma lista de inteiros a classe:

```
class _TelaState extends State<Tela> {  
  List<int> valores = [1, 2, 3, 4];  
  
  @override  
  Widget build(BuildContext context) {  
    return ...  
  }  
}
```

- Para cada posição “i” da lista, vamos retornar um Text apresentando tal valor na interface gráfica;

## ● Widgets de Layout - ListView

- Colocaremos o **ListView.builder()**, passando o tamanho da lista para o seu atributo **itemCount**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ListView.builder(
      itemCount: valores.length,
    ),
    appBar: ...,
  );
}
```

## ● Widgets de Layout - ListView

- Agora, para o atributo **itemBuilder** passaremos uma função que retorna um Text com o número na **i-ésima** posição;
- Tal função deve receber um **BuildContext** e **int**, inteiro esse que usaremos para acessar a **i-ésima** posição da lista:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: ListView.builder(
      itemCount: valores.length,
      itemBuilder: (BuildContext context, int index) {
        return Text('${valores[index]}');
      },
    ),
    appBar: ...,
  );
}
```

## ● Widgets de Layout - Expanded

- Veremos um exemplo de um uso mais “real” ao final dessa aula;
- Agora que vimos as classes **Column**, **Row** e **ListView**, vamos ver duas outras que permitem que dividamos o espaço que cada filho terá dentro de uma dessas classes;
- Começaremos pela classe **Expanded**;
- Utilizaremos tal classe, quando precisamos que um widget assuma todo o espaço disponível dentro de seu widget pai;
- Note que quando colocamos um **Container** contendo apenas um atributo **color** dentro de um **Column**, nada acontece;
- Precisamos de uma forma de informar ao Flutter que queremos que esse Container assuma todo o espaço disponível;
- Para tal, vamos utilizar o Expanded;
- Utilizaremos apenas seu atributo **child**;

## ● Widgets de Layout - Expanded

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Column(
      children: [
        Expanded(
          child: Container(
            color: Colors.blue,
          ),
        ),
      ],
    ),
    appBar: ...,
  );
}
```

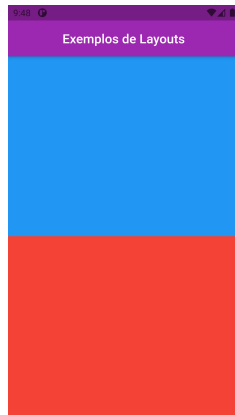
## ● Widgets de Layout - Expanded

- É possível até mesmo colocar vários **Expanded's** dentro de um mesmo **Column** ou **Row**;
- Eles irão disputar o espaço disponível, e por fim, irão dividir o espaço entre eles;
- Veja um exemplo de dois **Container's**;

## ● Widgets de Layout - Expanded

@override

```
Widget build(BuildContext context) {  
  return Scaffold(  
    body: Column(  
      children: [  
        Expanded(  
          child: Container(  
            color: Colors.blue,  
          ),  
        ),  
        Expanded(  
          child: Container(  
            color: Colors.red,  
          ),  
        ),  
      ],  
    ),  
    appBar: ...,  
  );  
}
```



**Figura 18:** Uso do Expanded.



## ● Widgets de Layout - Flexible

- Um widget “irmão” do Expanded é o **Flexible**;
- Basicamente sua única diferença é que o widget não irá ocupar todo o espaço disponível para ele, e sim, utilizar o suficiente;
- Caso precise, ai sim irá utilizar todo o espaço;
- Possui dois atributos que utilizaremos:
  - **Widget child:** seu widget filho;
  - **int flex:** determina a fração de espaço ocupada pelo filho;

## ● Widgets de Layout - Flexible

- Utilizaremos-o em nossos exemplos mais a frente;
- A princípio, vamos modificar o **body** anterior para determinar porcentagens aos widgets:

```
body: Column(  
  children: [  
    Flexible(  
      flex: 3,  
      child: Container(  
        color: Colors.blue,  
      ),  
    ),  
    Flexible(  
      flex: 7,  
      child: Container(  
        color: Colors.red,  
      ),  
    ),  
  ],  
)
```



**Figura 19:** Exemplo de uso do flex no Flexible.

## ● Widgets de Layout - Flexible

- Utilize o Flexible quando colocar um **ListView** dentro de um **Column**!
- Veremos um exemplo prático mais a frente;

## ● Widgets de Layout - Center

- O próximo widget que veremos, possui a funcionalidade de centralizar widgets dentro dele;
- Tal classe se chama **Center**;
- Utilizaremos apenas o atributo **child**, semelhante ao do Container;

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Text('Hello World'),
    ),
    appBar: ...,
  );
}
```

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

**Exercícios**

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

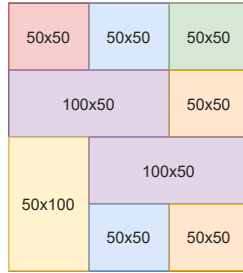
Exercícios

Compilação e Instalação

## ● Exercícios

1. Crie uma aplicação Flutter contendo uma Scaffold que possua uma AppBar contendo um título centralizado escrito “Exercício 1” e um **body** contendo um Column exibindo os números de 0 a 10;
2. Altere o item anterior para utilizar um ListView que exiba uma lista “valores”, sendo essa lista um atributo da classe. Utilize o construtor **ListView.builder()** para isso.
3. A partir do item anterior, crie uma função **adiciona()** que adiciona um valor na lista “valores” e redesenha a tela. Adicione um **FloatingActionButton** ao **Scaffold** contendo a função **adiciona()**.

4. Continuando o item anterior, adicione um TextField acima da ListView (utilize um Column no body do Scaffold) e altere a função **adiciona()** para que ela salve o valor digitado do usuário na lista de valores, limpando o TextField ao final.
5. Replique o layout definido na Figura 20 em uma nova tela.



**Figura 20:** Exercício 5.



## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

**Diálogos e Snackbar**

Padrão MVC

Exercícios

Navegação de Telas

Exercícios

Compilação e Instalação

## ● Diálogos

- Na aula de Dart vimos como lançar exceções e tratá-las;
- Porém, vimos apenas como exibir as exceções no console;
- Sabemos que o console não irá a aparecer para o usuário final;
- Tendo em vista isso, precisamos de uma forma mais interessante de apresentar erros para o usuário;
- Para isso, vamos utilizar caixas de diálogos;
- Não só servem para exibir exceções, mas possuem outros propósitos também;
- Veremos apenas alguns exemplos básicos;

## ● Diálogos

- Para exibirmos uma caixa de diálogo, vamos utilizar o método **showDialog()**;
- Tal método recebe como parâmetro um **BuildContext** e uma função builder que dado um **BuildContext** deve retornar um **Widget**;
- Para o widget, vamos retornar um **AlertDialog()**;
- Utilizaremos alguns atributos da classe **AlertDialog**:
  - **Widget title**: apresenta um titulo na caixa de diálogo;
  - **Widget content**: apresenta uma mensagem no corpo da caixa de diálogo;
  - **List<Widget> actions**: uma lista de widgets, normalmente adiciona-se um botão de confirmação para fechar a caixa de diálogo;

## ● Diálogos

- Para utilizar um diálogo, vamos criar um método **\_dialogo()** em nossa classe Tela;
- Dentro desse método, vamos invocar o método **showDialog()**;
- Passando o **BuildContext** e instanciando dentro da função **build** um novo **AlertDialog**:

```
_dialogo() {  
  showDialog(  
    context: context,  
    builder: (BuildContext context) {  
      return AlertDialog();  
    },  
  );  
}
```

## ● Diálogos

- Dentro do construtor do **AlertDialog**, vamos adicionar um **Text** dentro do atributo **title** e do **content**;
- No **actions** vamos adicionar um **ElevatedButton** que irá chamar o método **Navigator.pop()** dentro de sua função **onPressed**:

```
return AlertDialog(  
  title: Text('Titulo do erro'),  
  content: Text('Corpo'),  
  actions: [  
    ElevatedButton(  
      onPressed: () {  
        Navigator.pop(context);  
      },  
      child: Text('Confirmar'),  
    ),  
  ],  
);
```

- Podemos generalizar essa função e pedir as String's do título e do corpo como parâmetro da função **dialogo()**:

```
_dialogo(String titulo, String mensagem){  
    ...  
}
```

- Alterando assim dentro do construtor do **AlertDialog** para utilizar esses dois parâmetros:

```
return AlertDialog(  
  title: Text(titulo),  
  content: Text(corpo),  
  actions: [  
    ElevatedButton(  
      onPressed: () {  
        Navigator.pop(context);  
      },  
      child: Text('Confirmar'),  
    ),  
  ],  
);
```

- Para fins de exemplo, vamos criar uma função na classe Tela que apenas lança uma exceção do tipo Exception:

```
_lanca() {  
    throw Exception('Algum erro ocorreu');  
}
```



## ● Diálogos

- Para testar o exemplo, vamos dentro do **body** do **Scaffold** retonar um botão centralizado que chama a função **\_lanca()**;
- Como a função lança uma exceção, precisamos colocá-la dentro de um **try/catch**;
- E ao invés de exibirmos a exceção no console como fizemos nas aulas anteriores, vamos chamar o método **\_dialogo()**:

```
body: Center(  
  child: ElevatedButton(  
    onPressed: () {  
      try {  
        _lanca();  
      } on Exception catch (e) {  
        _dialogo('Erro', 'Um erro ocorreu!');  
      }  
    },  
    child: Text('Pressione'),  
  ),  
)
```

## ● Diálogos

- Com isso finalizamos o nosso exemplo de diálogos;
- Agora as exceções e erros podem ser tratadas chegando ao usuário final de forma mais “agradável”;

## ● SnackBar

- Uma outra forma de mostrar “mensagens agradáveis” para o usuário é usando **SnackBar’s**;
- Seu uso é comum quando estamos confirmando uma ação para o usuário;
- Utilizaremos a classe **SnackBar**;
- Tal classe possui alguns métodos que utilizaremos:
  - **Widget content**: uma mensagem (normalmente um Text) exibindo alguma informação ao usuário;
  - **SnackBarAction action**: um “botão” que pode conter uma mensagem (atributo label) e uma ação a ser executada quando pressionado (atributo onPressed);
- Para exibir uma SnackBar vamos utilizar o método **ScaffoldMessenger.of(context).showSnackBar()** da classe **Scaffold**;

## ● SnackBar

- A fins de exemplo, vamos criar uma função **\_\_snackbar()** que utilizaremos dentro de um botão;
- Tal função irá exibir uma SnackBar na parte inferior da tela;
- Faremos a função com mensagens pré-fixas, mas você pode alterar a função caso deseje, tal como fizemos no exemplo do AlertDialog para torná-lo genérico;
- Nossa função irá criar um objeto **SnackBar** e após irá passá-lo como parâmetro para o método **ScaffoldMessenger.of(context).showSnackBar()**:

```
__snackbar() {  
  SnackBar snackBar = SnackBar();  
  ScaffoldMessenger.of(context).showSnackBar(snackBar);  
}
```

## ● Snackbar

- Agora, dentro do construtor da classe **SnackBar** vamos passar como **content** um **Text** com uma mensagem “Ação confirmada”;
- No atributo **action** vamos passar um **SnackBarAction** com um **label** “ok”;
- Na função **onPressed** deixaremos ela como vazia:

```
SnackBar(  
  content: Text('Acao confirmada'),  
  action: SnackBarAction(  
    label: 'ok',  
    onPressed: () {},  
  ),  
);
```

- Para testarmos, vamos adicionar no **body** um botão centralizado que chame a função **`_snackbar()`**:

```
body: Center(  
  child: ElevatedButton(  
    onPressed: () {  
      _snackbar();  
    },  
    child: Text('Pressione'),  
  ),  
)
```

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

**Padrão MVC**

Exercícios

Navegação de Telas

Exercícios

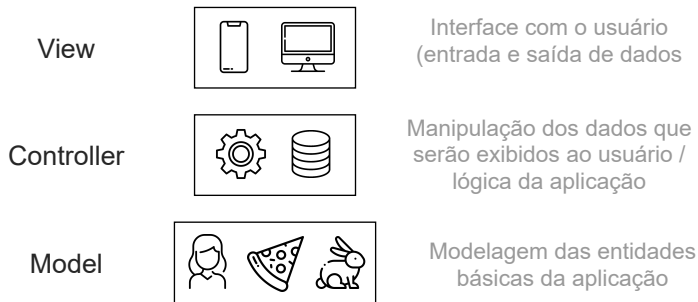
Compilação e Instalação

## ● Padrão MVC

- Falamos brevemente sobre o padrão MVC em slides anteriores;
- Agora veremos em um exemplo prático como aplicá-lo;
- O padrão consiste basicamente em dividir as classes em três tipos:
  - **Model**: classes que representam entidades da aplicação;
  - **View**: classes que representam componentes visuais da aplicação;
  - **Controller**: classes que controlam a lógica de negócio, separando-a da View.



## ● Padrão MVC



**Figura 21:** Organização do MVC.

## ● Padrão MVC

- No pacote **models**, ou melhor, na pasta **models**, colocaremos nossas entidades básicas;
- Por exemplo:
  - Usuario;
  - Produto;
  - etc.

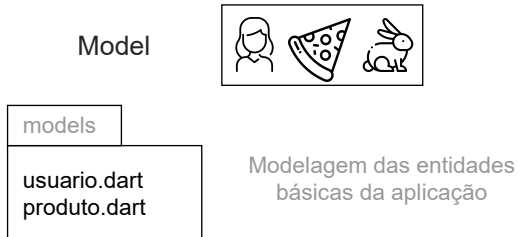
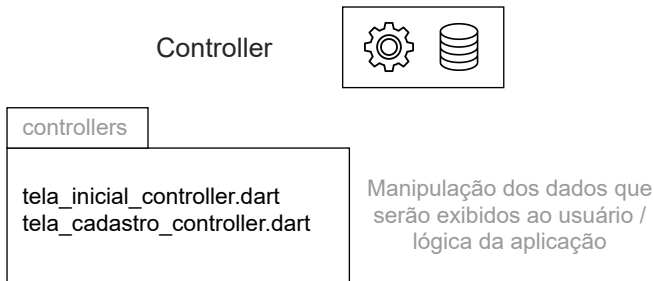


Figura 22: Organização do MVC.

## ● Padrão MVC

- Já no **controllers**, incluíremos as lógicas da nossa aplicação;
- Imagine por exemplo o aplicativo do iFood;
- O controller poderia ser o carrinho de uma loja;
- Onde a lista de produtos ficaria até que o usuário finalize a compra;



**Figura 23:** Organização do MVC.

## ● Padrão MVC

- Por fim, a **view** irá conter os componentes do Flutter:
  - Telas;
  - Widgets;
  - etc.

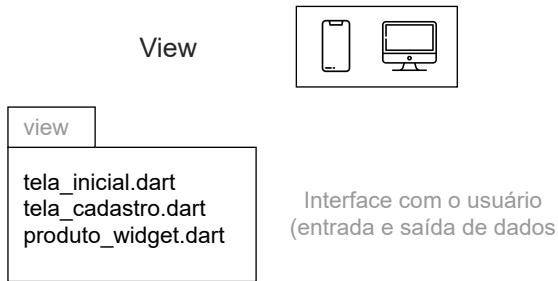


Figura 24: Organização do MVC.

## ● Padrão MVC

- Vamos construir um exemplo de uma lista de contatos usando o MVC;
- Nela será possível ver os contatos cadastradas e também inserir novos contatos;

## ● Padrão MVC - Model

- Começaremos definindo nosso modelo, a classe Contato;
- Para isso, vamos criar uma pasta **models** e dentro dela a classe **Contato** no arquivo **contato.dart**;
- Tal classe terá um nome e um número de telefone;
- Além de um construtor:

```
class Contato {  
  String nome;  
  String telefone;  
  
  Contato({this.nome = "", this.telefone = ""});  
}
```

## ● Padrão MVC - Controller

- Agora, para criarmos a lógica de negócio da aplicação, vamos criar a classe **ListaTelefonicaController**, criaremos tal classe dentro de uma pasta chamada **controllers**;
- Lá, vamos criar um atributo que será a lista de contatos a ser desenhada no aplicativo;
- Faremos esse atributo privado e com um get;
- É nessa classe que iremos armazenar os dados, logo, deixaremos um **TextEditingController** para o nome do contato e outro para o telefone;

## ● Padrão MVC - Controller

```
class ListaTelefonicaController {  
    List<Contato> _contatos = [  
        Contato(nome: 'Vinicius', telefone: '99999999'),  
    ];  
  
    TextEditingController caixaNome = TextEditingController();  
    TextEditingController caixaTelefone = TextEditingController();  
  
    List<Contato> get contatos => _contatos;  
}
```

- Note que já deixamos um contato salvo na lista;



## ● Padrão MVC - Controller

- Agora que temos o controller que armazena os dados, vamos criar um método **adicionarContato()** que irá instanciar um contato usando os dados dos **TextEditingController's** e após irá limpá-los, adicionando a lista de contatos ao final:

```
class ListaTelefonicaController {  
  ...  
  adicionarContato() {  
    Contato contato = Contato(  
      nome: caixaNome.text,  
      telefone: caixaTelefone.text,  
    );  
    caixaNome.clear();  
    caixaTelefone.clear();  
    _contatos.add(contato);  
  }  
}
```

## ● Padrão MVC - Controller

- Por fim, faremos apenas um método **removerContato()** do qual passaremos um inteiro e ele irá remover da lista o contato na posição passada:

```
class ListaTelefonicaController {  
    ...  
    adicionarContato() {  
        Contato contato = Contato(  
            nome: caixaNome.text,  
            telefone: caixaTelefone.text,  
        );  
        caixaNome.clear();  
        caixaTelefone.clear();  
        _contatos.add(contato);  
    }  
  
    removerContato(int pos) => _contatos.removeAt(pos);  
}
```

## ● Padrão MVC - View

- Agora podemos implementar nossa interface gráfica;
- Para as classes da View, vamos criar um pacote chamado **views**;
- Nela, vamos criar uma classe **TelaInicial** que estende a classe **StatefulWidget**;
- Não se esqueça de adicionar essa classe no MaterialApp que não iremos mostrar a criação visto que já vimos em slides anteriores;
- No método **build()** vamos retornar um Scaffold com uma AppBar e um título escrito “Lista Telefonica”:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Lista Telefonica'),
      centerTitle: true,
    ),
  );
}
```

## ● Padrão MVC - View

- Em seguida, para tornar mais organizado o código, vamos criar um método **`_body()`** que irá retornar todo o conteúdo do **`body`**;
- Chamaremos esse método no **`_body()`**:

```
class _TelaInicialState extends State<TelaInicial> {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: ...,  
      body: _body(),  
    );  
  }  
  
  _body() {}  
}
```

## ● Padrão MVC - View

- Precisaremos de uma instancia do nosso **controller** aqui, para isso, vamos criar um atributo **controller**:

```
class _TelaInicialState extends State<TelaInicial> {  
  
  ListaTelefonicaController controller = ListaTelefonicaController();  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: ...,  
      body: _body(),  
    );  
  }  
  
  _body() {}  
}
```

## ● Padrão MVC - View

- Vamos dividir o **body** em duas regiões, uma para fazer o *input* dos dados do usuário e outra para apresentar a lista de contatos;
- Na Figura 25 é possível ver o resultado que iremos construir;

The image shows a mobile application interface titled "Lista Telefonica". It features two main sections outlined in red. The top section, labeled with a red "1", contains two text input fields: "Digite o nome:" and "Digite o telefone:". Below these fields is a blue button labeled "Cadastrar". The bottom section, labeled with a red "2", displays a list of contacts. The first contact is visible, showing "Nome: Vinicius" and "Telefone: 99999999". There is a close button (an 'x' icon) in the top right corner of the list area.

**Figura 25:** Resultado a ser construído.

## ● Padrão MVC - View

- Começaremos definindo um método **\_input()** do qual iremos construir a parte com as duas **TextField** e o **ElevatedButton** (a região 1 da Figura 25);
- Para isso, vamos retornar um **Column** e acrescentar cinco **Container's** dentro;

```
_input() {  
  return Column(  
    children: [  
      Container(),  
      Container(),  
      Container(),  
      Container(),  
      Container(),  
    ],  
  );  
}
```

## ● Padrão MVC - View

- No primeiro **Container**, vamos colocar um Text com margem de 20 em cima e um **Text** contendo a mensagem “Digite o nome:”:

```
Container(  
  margin: EdgeInsets.only(top: 20),  
  child: Text('Digite o nome:'),  
),
```



## ● Padrão MVC - View

- No segundo, vamos colocar um **TextField** e uma margem de 20 na esquerda e na direita;
- Lembrando de colocar o controller referente ao campo nome que é um atributo da classe **ListaTelefonicaController**:

```
Container(  
  child: TextField(  
    controller: controller.caixaNome,  
  ),  
  margin: EdgeInsets.symmetric(horizontal: 20),  
),
```

## ● Padrão MVC - View

- Para o terceiro, vamos colocar um **Text** contendo a mensagem “Digite o telefone:” e uma margem de 20 no topo:

```
Container(  
  margin: EdgeInsets.only(top: 20),  
  child: Text('Digite o telefone:'),  
),
```

## ● Padrão MVC - View

- No quarto, iremos colocar outro **TextField** contendo o controller que está na classe **ListaTelefonicaController**;
- Além de uma margem na esquerda e na direita de 20:

```
Container(  
  margin: EdgeInsets.symmetric(horizontal: 20),  
  child: TextField(  
    controller: controller.caixaTelefone,  
  ),  
),
```

## ● Padrão MVC - View

- Por fim, faremos o quinto, que contém o **ElevatedButton**;
- Adicionaremos uma margem de 20 no topo e na função **onPressed** do botão, vamos redesenhar a tela com o **setState()** chamando o método **adicionarContato()** de nosso controller:

```
Container(  
  margin: EdgeInsets.only(top: 20),  
  child: ElevatedButton(  
    onPressed: () {  
      setState(() {  
        controller.adicionarContato();  
      });  
    },  
    child: Text('Cadastrar'),  
  ),  
),
```

## ● Padrão MVC - View

- Agora, precisamos apenas desenhar a lista de objetos que está no **controller**;
- Para isso, vamos utilizar um widget chamado [ListTile](#);
- Cada **ListTile** irá representar um contato da lista;
- Usaremos os atributos:
  - **Widget title**: o título do objeto;
  - **Widget subtitle**: o subtítulo do objeto;
  - **Widget trailing**: um widget a direita;
- No **title** colocaremos o nome do contato;
- No **subtitle** vamos colocar o seu telefone;
- E no **trailing** iremos colocar um botão que irá excluir o contato da lista;

## ● Padrão MVC - View

- Utilizaremos um **ListView** para desenhar todos os elementos da lista que está no **controller**;
- Vamos então implementar o método **\_contatos()**;
- Começaremos retornando um **ListView**:

```
_contatos() {  
  return ListView.builder(  
    itemBuilder: (context, index) {  
    },  
    itemCount: controller.contatos.length,  
  );  
}
```

- Lembrando que o atributo **itemCount** é número de elementos a serem desenhados, ou seja, o tamanho da lista do **controller**.

## ● Padrão MVC - View

- Agora, dentro do **itemBuilder** podemos retornar um **ListTile**, utilizando os atributos que comentamos nos slides anteriores:

```
itemBuilder: (context, index) {  
  return ListTile(  
    title: Text('Nome: ${controller.contatos[index].nome}'),  
    subtitle: Text('Telefone: ${controller.contatos[index].telefone}'),  
    trailing: IconButton(  
      icon: Icon(Icons.clear),  
      onPressed: () {  
        setState(() {  
          controller.removerContato(index);  
        });  
      }),  
  );  
}
```

- Note que a função passada para o **onPressed** redesenha a tela usando o método **removerContato()** que implementamos no **controller** dessa tela;

## ● Padrão MVC

- Com isso, terminamos o exemplo de uma aplicação usando o padrão MVC;
- Note como fica nítido a divisão entre as funcionalidades dos pacotes;
- Cada classe é responsável por algo, assim, a lógica de negócio, fica no **controller**, enquanto que a lógica da interface gráfica fica na **view**;



## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

**Exercícios**

Navegação de Telas

Exercícios

Compilação e Instalação

## ● Exercícios - Diálogos e SnackBar

1. Crie uma tela contendo dois botões (utilize a classe que quiser para um botão), um deles deve exibir uma caixa de diálogo contendo o número de vezes que a caixa de diálogo foi aberta e o outro contendo a data atual (utilize o construtor da classe `DateTime.now()`);
2. Adapte o código dos exemplos, para que quando o usuário remova um objeto, uma SnackBar apareça confirmando que o objeto foi removido;
3. Adapte o código dos exemplos, para que quando o usuário digitar um nome contendo algum número, uma exceção seja lançada pelo **controller** e a **view** trate essa exceção mostrando uma caixa de diálogo informando que o nome não pode conter números;

## ● Exercícios - MVC

4. Crie um aplicativo usando o padrão MVC para uma **lista de tarefas**. Cada tarefa deve possuir um **nome** e um **status** de concluído ou não. Crie funções no **controller** para que seja possível adicionar uma tarefa e remover uma tarefa. Na view, exiba a lista de tarefas usando o **ListTile** e o **ListView**, mostre o nome da tarefa e no **trailing** dois íconos de sua escolha, um para quando a tarefa estiver concluída e outro para quando a tarefa estiver pendente;
5. Continuando o item anterior, coloque o **ListTile** dentro de um widget [InkWell](#), utilize apenas o **child** (um Widget) e o **onTap** (uma função igual ao onPressed de botões), para que ao o usuário tocar na tarefa, ela mude altere seu status;

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

**Navegação de Telas**

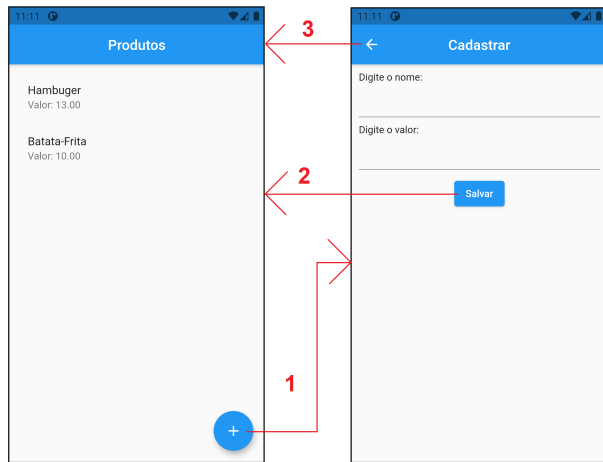
Exercícios

Compilação e Instalação

## ● Navegação de Telas

- Agora veremos o último tópico de Flutter nativo desse curso: a navegação de telas;
- É comum em aplicações, sejam elas, mobile, web ou desktop, a existência de múltiplas telas;
- Das quais, quando o usuário executa uma ação, por exemplo, o pressionar de um botão;
- A aplicação saia da tela atual e avance para outra;
- Na Figura 26 veremos um exemplo contendo três casos.

## ● Navegação de Telas



**Figura 26:** Exemplo de navegação entre telas.

## ● Navegação de Telas

- Note que na Figura 26 temos três setas:
  1. Navegação de ida para uma segunda tela;
  2. Navegação de retorno para uma tela inicial, após efetuar alguma operação;
  3. Navegação de retorno para uma tela inicial, sem realizar nada;
- O Flutter possui uma classe que utilizaremos para navegar de uma tela para outra, tal classe chama-se **Navigator**;
- Com a classe **Navigator** é possível ir de uma tela para outra e até mesmo trazer dados consigo;

## ● Navegação de Telas

- Utilizaremos alguns dos seus métodos estáticos:
  - **Future<T?> push(BuildContext context, Route<T> route):** navega para uma tela, deixando a tela atual em *stand by*;
  - **Future<T?> pushReplacement(BuildContext context, Route<T> newRoute):** navega para uma nova tela, descartando a tela antiga;
  - **void pop(BuildContext context, [T result]):** retorna para uma tela “aguardando em *stand by*”, é possível trazer um valor para a tela de retorno;
- Utilizaremos a classe [MaterialPageRoute](#) como implementação da classe **Route**;



## ● Navegação de Telas

- Basicamente iremos utilizar um dos dois métodos **push()** para **irmos** para outra tela, e o método **pop()** para **retornarmos**;
- Na Figura 26, fora utilizado o método **push()** e não o **pushReplacement()**;
- Pois quando utilizamos o método **push()**, assume-se que o método **pop()** poderá ser usado a qualquer momento para retornar para a outra tela;
- Também ao usar o método **push()**, caso a outra tela possua um **Scaffold**, um botão de retorno será criado no **leading** dele;

## ● Navegação de Telas

- Como citado anteriormente, vamos utilizar a classe **MaterialPageRoute** para fazermos a navegação;
- Seu construtor pede apenas um parâmetro **builder**:
  - **MaterialPageRoute(WidgetBuilder builder)**: recebe uma função que possui um **BuildContext** como parâmetro e retorna um **Widget**, isto é, a tela ao qual iremos navegar;
- Veremos alguns exemplos agora;

## ● Navegação de Telas

- Criaremos duas telas, uma classe **Tela1** e outra classe **Tela2**, ambas contendo um **Scaffold** com uma **AppBar** e um título:

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

class Tela1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Tela 1'),
        centerTitle: true,
      ),
    );
  }
}
```

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

class Tela2 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Tela 2'),
        centerTitle: true,
      ),
    );
  }
}
```

## ● Navegação de Telas

- Faremos com que a classe que contenha o **MaterialApp** instancie um objeto da **Tela1**:

```
return MaterialApp(  
  debugShowCheckedModeBanner: false,  
  home: Tela1(),  
);
```

- Agora, vamos voltar a **Tela1** e adicionar um FloatingActionButton;

## ● Navegação de Telas

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Tela 1'),
      centerTitle: true,
    ),
    floatingActionButton: FloatingActionButton(
      child: Icon(Icons.add),
      onPressed: () {},
    ),
  );
}
```

- Criaremos agora um método **\_navega()**, que irá avançar para a Tela2;

## ● Navegação de Telas

- Como a classe **Tela1** estende a classe **StatelessWidget**, precisamos passar o **BuildContext** como parâmetro para o método (caso ela estendesse a classe **StatefulWidget** não seria necessário pois o **BuildContext** é visível a todos nessa classe);
- Agora, chamaremos o método **push()** da classe **Navigator**, passando o **context** e instanciando um novo objeto da classe **MaterialPageRoute**;

```
_navega(BuildContext context) {  
  Navigator.push(context, MaterialPageRoute(builder: (context) {  
    return Tela2();  
  }));  
}
```

- Note que o **builder** da classe **MaterialPageRoute** irá retornar uma instancia da **Tela2**, esta que será utilizada pelo **Navigator** para avançar para outra tela;

## ● Navegação de Telas

- Como a classe **Tela1** estende a classe **StatelessWidget**, precisamos passar o **BuildContext** como parâmetro para o método (caso ela estendesse a classe **StatefulWidget** não seria necessário pois o **BuildContext** é visível a todos nessa classe);
- Agora, chamaremos o método **push()** da classe **Navigator**, passando o **context** e instanciando um novo objeto da classe **MaterialPageRoute**;

```
_navega(BuildContext context) {  
  Navigator.push(context, MaterialPageRoute(builder: (context) {  
    return Tela2();  
  }));  
}
```

## ● Navegação de Telas

- Agora, basta colocarmos a função **`_navega()`** no atributo **`onPressed`** do botão:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Tela 1'),
      centerTitle: true,
    ),
    floatingActionButton: FloatingActionButton(
      child: Icon(Icons.add),
      onPressed: () {
        _navega(context);
      },
    ),
  );
}
```



## ● Navegação de Telas

- Com isso, já é possível ir e voltar da **Tela2**;
- Agora, veremos como trazer objetos da classe **Tela2**, para a classe **Tela1**, quando voltarmos;
- Note que a utilização do método **pushReplacement()** é exatamente igual (teste você mesmo), a única diferença é que não é possível retornar com o **pop()**;

## ● Navegação de Telas

- Para utilizarmos o método **pop()**, vamos criar uma botão centralizado no **body** do **Scaffold** da **Tela2** contendo uma função que chama o método **pop()**:

```
body: Center(  
  child: ElevatedButton(  
    child: Text('Retornar'),  
    onPressed: () {  
      Navigator.pop(context);  
    },  
  ),  
),
```

## ● Navegação de Telas

- O que fizemos agora, foi exatamente replicar o botão de retorno que há na **AppBar** da **Tela2**;
- Vamos retornar um objeto para a **Tela2**;
- Retornaremos uma String, que será exibida centralizada na **Tela1**;

## ● Navegação de Telas

- Para isso, precisamos retornar a **Tela1** e transformá-la em uma **StatefulWidget**;
- Há um atalho para isso com a extensão do Flutter no VSCode (ver slide 26);
- Tendo convertido, vamos criar um atributo String chamado **mensagem**:

```
class _Tela1State extends State<Tela1> {  
  String? mensagem;  
  @override  
  Widget build(BuildContext context) {  
    return ...  
  }  
  
  _navega(BuildContext context) {  
    ...  
  }  
}
```

## ● Navegação de Telas

- Agora, utilizando o **operador ternário** visto na aula de Dart, faremos uma verificação no **body** para sabermos se o valor da mensagem existe;
- Caso exista, vamos exibir ela centralizada no **body**;
- Caso não, vamos exibir um **Container** vazio:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: ...,
    body: mensagem != null
      ? Center(
        child: Text(mensagem),
      )
      : Container(),
    floatingActionButton: ...,
  );
}
```

## ● Navegação de Telas

- Porém, agora precisamos modificar o método **\_navega()**, para salvar o valor de retorno em nosso atributo **mensagem**;
- Lembrando que o método **Navigator.push()** retorna um **Future<T>**;
- A priori sabemos que esse retorno é uma mensagem (vamos fazer ele ser nos próximos slides);
- Logo, podemos usar o método **then()** já que o retorno é um **Future**, para quando o resultado for retornado, salvarmos-o no atributo **mensagem** e redesenhamos a tela:

```
_navega(BuildContext context) {  
  Navigator.push(context, MaterialPageRoute(builder: (context) {  
    return Tela2();  
  })).then((value) {  
    setState(() {  
      mensagem = value;  
    });  
  });  
}
```

## ● Navegação de Telas

- Por fim, vamos tornar o exemplo mais iterativo e adicionar um **TextField** na classe **Tela2**;
- Quando o usuário pressionar o botão, o valor digitado será retornado para a **Tela1**;
- Para isso, vamos envolver o **ElevatedButton** da classe **Tela2** com um **Column**;
- Precisaremos também de um **TextEditingController** como atributo da classe **Tela2**;
- Este, colocaremos dentro do **TextField** que iremos colocar em cima do **ElevatedButton**;

## ● Navegação de Telas

```
class Tela2 extends StatelessWidget {  
  TextEditingController caixa = TextEditingController();  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: ...,  
      body: Column(  
        children: [  
          TextField(controller: caixa),  
          ElevatedButton(  
            child: Text('Retornar'),  
            onPressed: () {  
              Navigator.pop(context, caixa.text);  
            },  
          ),  
        ],  
      ),  
    );  
  }  
}
```



## ● Navegação de Telas

- Repare que o texto da **TextEditingController** é passado como parâmetro no método **pop()** juntamente com o **BuildContext**;
- É interessante comentar também que o método **pop()**, quando não passado nenhum parâmetro além do **context**, ele retorna **null**;
- Lembre-se sempre de tratar tal caso, pois haverá um erro de compilação caso você não verifique se o retorno é **null** antes de utilizá-lo.

## ● Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

**Exercícios**

Compilação e Instalação

1. Crie duas classes **TelaLogin** e **TelaInicial**, a classe **TelaLogin** deve possuir duas caixas de texto (login e senha) e um botão, a ação do botão deve navegar para a **TelaInicial** apenas se o usuário digitar o usuário e a senha correta. Utilize o método **pushReplacement()** para avançar da **TelaLogin** para a **TelaInicial**.
2. Modifique o código do item anterior para incluir um botão de **logout** na classe **TelaInicial**, tal botão deve levar o usuário de volta para a **TelaLogin**;

## Seções

Introdução

Widgets, StatelessWidget e StatefulWidget

Widgets Básicos

Exercícios

Widgets de Layout

Exercícios

Diálogos e Snackbar

Padrão MVC

Exercícios

Navegação de Telas

Exercícios

Compilação e Instalação

## ● Compilação e Instalação

- Agora que vimos como construir aplicações completas utilizando o Flutter, veremos como compilá-las (gerar um **.apk**) e instalá-las em outros dispositivos;
- Como o foco do curso é o Android, veremos a compilação apenas para o Android;
- Podemos compilar o aplicativo de duas formas:
  - **APK:** gera um **Android Package** que pode ser instalado em algum dispositivo Android (Smartphone, Tablet, Watch, TV, etc);
  - **App Bundle:** gera um compilado para ser publicado na **Google Play**;
- Não veremos nesse curso como publicar um aplicativo na **Google Play**, apenas como compilá-lo;

## ● Compilação e Instalação - Compilando

- A geração do APK também pode ser feita de duas formas:
  - Específico por arquitetura;
  - Geral;
- Faremos a compilação geral, visto que não falamos sobre as arquiteturas disponíveis para Android;
- Para gerar tal compilado, basta executar o comando **flutter build apk** no Terminal;
- É necessário estar na raiz do diretório, para isso, no VSCode clique em **Terminal** e em **New Terminal** com o seu projeto aberto;

## ● Compilação e Instalação

- A instalação do aplicativo é possível ser feita de duas formas:
  - Enviando o **.apk** para o dispositivo a ser instalado (via cabo ou via whatsapp, email, etc);
  - Conectar o celular no computador e instalar via comando;
- Veremos as duas formas;

## ● Compilação e Instalação - Instalando

- Começaremos pela instalação conectando o celular ao computador;
- Para isso, basta apenas conectar o celular no computador e executar o comando **flutter install**;
- É necessário estar no mesmo diretório do projeto.
- Já para instalar em um celular remotamente, primeiro precisamos encontrar o **.apk**;
- Para isso vamos acessar a pasta: **build > app > outputs > apk > release**;
- Isso, óbvio, após termos executado o comando para compilar visto nos slides anteriores;




## ● Compilação e Instalação - Instalando

- Em seguida, basta enviar o arquivo **app-release.apk** para o seu dispositivo;
- Via e-mail, whatsapp, etc;
- Apenas atente-se que o aplicativo com o qual você receber/abrir o **.apk** deve permitir a instalação de fontes desconhecidas;
- Cada versão do Android possui a sua forma de alterar isso, porém, no geral, basta acessar:
- **Configurações > App e Notificação > WhatsApp > Avançado > Instalar apps desconhecidos.**
- **Porém, cuidado com essa configuração pois pode sujeitar seu dispositivo a instalação de softwares maliciosos!**

## Referencias

 CODE, D. **Dart & Flutter support for Visual Studio Code**. 2021. Disponível em: <<https://dartcode.org>>.

 FLUTTER. **Flutter documentation**. 2021. Disponível em: <<https://flutter.dev/docs>>.

Obrigado :)  
Vinicius Takeo Friedrich Kuwaki  
vtkwki@gmail.com  
github.com/takeofriedrich

**NEMOBIS**

