0.1 Antes de Começar

- A linguagem Dart é orientada a objetos.
- O estilo é baseado em C.
- A linguagem é tipada, havendo opção de tipagem estática e dinâmica.
- Os comentários seguem o padrão do Java (single e multiline). Uma sintaxe de comentários para documentação também existe.

```
// Isto é um comentário

/*
   Um comentário de
   múltiplas linhas
*/

/// Comentário de documentação

/**
  Comentário de documentação
  com múltiplas linhas
*/
```

- A linguagem descreve tanto o comportamento quanto a interface gráfica dos aplicativos.
- Uma função main é o ponto de partida do código.

```
void main() {
  // Escreva algum código
}
```

0.2 Widgets

- Widgets são elementos da GUI. A sua interface gráfica será uma hierarquia deles.

```
// Exemplo de um widget Simples
Text("Hello!")

// Widget mais complexo
RaisedButton(
  onPress : function() {
    // O código vem aqui
  },
  child : Text("Click me!")
)
```

```
// Aqui temos vários widgets
Center(
  child : Container(
    child : Row(
      Text("Child 1"),
      Text("Child 2"),
      RaisedButton(
        onPress : function() {
          // O código vem aqui
        },
        child : Text("Click me")
) ) ) )
```

0.3 Tipos

Tipos Básicos

- Em Dart, declarar o tipo é uma opção.

```
String name = "Ulisses";
int idade = 25;
double altura = 1.74;
bool casado = true;
```

- Outra opção é declarar var.

```
var name = "Ulisses";
var idade = 25;
var altura = 1.74;
var casado = true;
```

- Para obter o tipo da variável em tempo de execução, use o comando runtimeType.
- Para saber se uma variável é de um dado tipo, use o comando is.

```
var altura = 1.74;
print(altura is int); // false

print(altura.runtimeType);// double
```

- Em Dart, você poderá mudar o tipo de uma variável declarada como dinâmica (dynamic).

```
dynamic altura = 1.74;
print(altura.runtimeType);// double
```

```
altura = "gigante";
print(altura.runtimeType);//
 ↪  String

altura = true;
print(altura.runtimeType);// bool
```

- A palavra reservada const declara constantes conhecidas em tempo de compilação.
- A palavra reservada final declara constantes cujo valor só pode ser associado uma vez (em tempo de compilação ou execução).
- Em ambos, a instanciação do valor é obrigatória.

```
// Um valor deve ser atribuído
// em tempo de compilação
const altura = 1.74;

// O valor da constante abaixo
// pode ser atribuído em tempo
// de execução.
final agora = DateTime.now();
```

- As conversões para string usam toString, as conversões para tipos numéricos usam parse.

```
int idade = 25;
double altura = 1.74;
String si = idade.toString();
String sa = altura.toString();

int id1 = int.parse(si);
double al1 = double.parse(sa);
print("Idade: $id1, Altura:
 ↪  $al1");
// Idade = 25, Altura = 1.74
```

Tipos Estruturados

- Os principais são sequências (List), dicionários (Map) e conjuntos (Set). Para cada tipo, existe uma gama de funções.

- **Sequências**

```
/* List: sequência de valores
 ↪  indexáveis
 pela posição. Podemos mudar
 ↪  valores
```

```
 existentes e acrescentar novos.
 ↪  */
var seq = ["a", "e", "i", 1, 2];
String k = seq[2]; // k recebe "i"
print(seq.runtimeType);//JSArray<Objec

seq.add(3);
print(seq); //[a, e, i, 1, 2, 3]
print(seq.indexOf("e")); // 1

// Podemos iterar com o método
 ↪  forEach
 seq.forEach(print);
/*
a
e
i
1
2
3
*/
```

- **Dicionários**

```
// Map: Pares "chave : valor"
```

```
var dic = {
    "key"   : "value",
    1       : "one",
    3.14    : "pi",
    "flag"  : true
  };
print(dic);
/* {key: value, 1: one,
   3.14: pi, flag: true} */

var x = dic["key"]; // Acessos
print(dic.runtimeType);
//JsLinkedHashMap<Object,Object>

// Acrescentando novos elementos
dic[2]      = "dois";
dic["dois"] = 2;
print(dic);
/*
{key: value, 1: one, 3.14: pi,
flag: true, 2: dois, dois: 2}
*/

// Podemos iterar com a função
 ↪  forEach
```

```
dic.forEach( (key, val) {
    print("C: $key, V: $val");
});
/*
C: key, V: value
C: 1, V: one
C: 3.14, V: pi
C: flag, V: true
C: 2, V: dois
C: dois, V: 2
*/
```

```
var docentes = Map<String, int>();
docentes["Ulisses"] = 5;
docentes["Meira"] = 3;
docentes["Marco"] = 1;
docentes["Gisele"] = 4;

print(docentes);
/*
{Ulisses: 5, Meira: 3,
Marco: 1, Gisele: 4}
*/
```

```
docentes.remove("Marco");
print(docentes);
/*
{Ulisses: 5, Meira: 3, Gisele: 4}
*/

print(docentes.keys);
//(Ulisses, Meira, Gisele)

print(docentes.values);  //(5, 3,
 ↪  4)
```

```
Map discentes = { };
print(discentes.isEmpty); // true

discentes["Bernini"] = 2;
discentes["Gislaine"] = 3;

discentes.forEach((k, v) {
    print( k +" discente #" +
      v.toString());
});
/*
Bernini discente #2
```

```
Gislaine discente #3
*/
```

- **Conjuntos**

```
/*
 Set: itens não ordenados dentro do
conjunto e não há elemento repetido
*/

Set docentes = Set();
docentes.addAll([ "Ulisses",
    "Meira", "Leon", "Ulisses"]);
docentes.add("Ana Estela");
docentes.remove("Meira");

print(docentes);
// {Ulisses, Leon, Ana Estela}

print(docentes.contains("Ulisses"));
// true

print(docentes.containsAll(
    [ "Meira",
```

```
        "Ana Estela" ]
)); // false

/* A linha a seguir gera ERRO */
print(docentes[0]);
// Não é possível indexar Set
```

0.4 Operadores

- Operadores Aritméticos

```
double a = 23.0;
double b = 7.0;

a  +  b; // Adição :          30.0
a  -  b; // Subtração:        16.0
a  *  b; // Multiplicação:    161.0
a  /  b; // Divisão: 3.2857142857
a  ~/ b; // Divisão Inteira: 3
a  %  b; // Resto da Divisão: 2

/*
 Abaixo, o valor impresso é 23,
 mas o valor é incrementado logo
 em seguida
*/
print(a++); // 23
print(a);   // 24

/*
 Abaixo, o valor impresso é 25,
 pois o incremento ocorreu antes
```

```
 de retornar o valor da expressão
*/
print(++a); // 25
print(a);   // 25
```

- Operadores Aritméticos de Atribuição

```
double a = 23.0;

a += 1; // 24
a -= 1; // 23
a *= 2; // 46
a /= 2; // 23
```

- Operadores de Comparação

```
double a = 23.0;
double b = 7.0;

a  <  b; // Menor que:          false
a <=  b; // Menor ou igual:    false
a ==  b; // igual:             false
a  >  b; // Maior que          true
```

```
a >=  b; // Maior ou igual  true
a !=  b; // Diferente       true
```

- Operadores Lógicos

```
bool a = true;
bool b = false;

// Operadores Lógicos
a   &&   b;  // false
a   ||   b;  // true
!   a;       // false
```

0.5 Comandos Condicionais

- O comando if só aceita resultados booleanos. A ideia comum em outras linguagens de que existe um "contexto" booleano não é válida. Ou seja, o inteiro 1 não será considerado verdadeiro e nem o inteiro 0 será considerado falso.

```
var professor = "Ulisses";
if (professor=="Ulisses" ||
 ↪  professor=="Meira") {
```

```
  print ("Professor FT/Unicamp");
} else if (professor=="Zanoni") {
  print ("Professor IC/Unicamp");
} else {
  print("Não sei quem é");
}
```

- O comando switch pode lidar com tipos não booleanos, desde que os objetos comparados sejam do mesmo tipo (subclasses não são permitidas) e as classes não sobrescrevam o operador ==.
- É comum o comando switch ser usado com enumerate. Neste caso, um erro será gerado se faltar cláusula para algum dos elementos no enumerate.

```
enum Disciplinas {SI700, SI202,
 ↪  SI101, SI100}
var disciplina =
 ↪  Disciplinas.SI700;

switch(disciplina) {
  case Disciplinas.SI700 :
     print("Ambos os semestres");
     break;
```

```
    case Disciplinas.SI202 :
        print("Segundo semestre");
        break;
    case Disciplinas.SI101 :
    case Disciplinas.SI100 :
        print("Primeiro semestre");
        break;
}
// Ambos os semestres
```

- Existem operadores ? e ?? para gerar comandos condicionais com apenas uma linha de código.

```
// Condição ternária:
bool a = true;
int b = 1;
int c = 2;
var d;

/* Se a for verdadeiro, então
retorna b, caso contrário c. */
print(a? b : c); // 1

/* Se d for não nulo retorna d,
```

```
caso contrário b */
print(d ?? b); // 1
```

0.6 Laços de Repetição

- Os laços de repetição são muito semelhantes ao que encontramos na linguagem Java. Vamos exemplificar os principais usos dos laços for, while e do ... while.

- **Comando while**

```
int count  = 0;
while (count < 4) {
  print("Count = $count");
  count = count + 1;
}
/*
Count = 0
Count = 1
Count = 2
Count = 3
*/
```

- **Comando do ... while**

```
int count  = 0;
do {
  print("Count = $count");
  count = count + 1;
  if (count == 2) {
    break;
  }
} while (count < 5);
/*
Count = 0
Count = 1
*/
```

- **Comando for**

```
var soma = 0;
for (var i =1; i <= 10; i++){
  soma += i;
}
print(soma); // 55

// Iteração sobre iterators
var numeros = [1, 2, 3, 4, 5];
```

```
for (var  num in numeros){
  soma += num;
}
print(soma); // 70
```

0.7 Funções

Forma Básica

- Possuem nome, corpo onde se coloca códico, lista de parâmentros de entrada e um tipo de retorno.

```
// Esta função retorn null
void hello_world() {
  print("Hello World!");
}
// Esta função recebe um parâmetro
void hello_user(user) {
  print("Hello $user");
}
void main() {
  hello_world(); // Hello World!
  hello_user("Ulisses");//Hello
   ↪  Ulisses
```

```
}
```

Retorno de Funções

- A declaração de tipo de retorno é opcional. Se o programador não declarar uma cláusula return, então a função retorna null.

```
findArea(int altura, int largura){
  return altura * largura;
}

main() {
  int x = findArea(2,4);
  print(x); // 8
}
```

Funções como Expressões

- Sintaxe mais simples com o operador => para apenas um comando.

```
int findArea(int altura, int
 ↪  largura) =>
  altura * largura;
```

```
main() {
  int x = findArea(2,4);
  print(x); // 8
}
```

Funções como Variáveis

- Funções em Dart podem ser passadas como parâmetros e podem ser atribuídas a variáveis.

```
// Função que soma
int soma(a, b) {
  return a+b;
}
/* Função  depende de outra
  passada como parâmetro. */
doSomething(param_a, param_b,
 ↪  funcao){
  return funcao(param_a, param_b);
}

void main() {
  var x = doSomething(2, 5, soma);
```

```
  print(x);
}
```

Parâmetros Opcionais Posicionais

- Parâmetros podem ser declarados opcionais com colchetes. A ordem dos parâmetros define como os valores passados pelo usuário serão atribuidos.
- Se o usuário decidir não passar valor naquela posição, então o valor null será usado por padrão.

```
/*
Argumentos posicionais opcionais:
devem ocorrer após os obrigatórios
*/
void hello_familia(user, [esposa])
 ↪  {
  print("Hello $user e $esposa");
}

// Função Principal
void main() {
  // Argumentos opcionais
   ↪  posicionais
```

```
  hello_familia("Ulisses");
  hello_familia("Ulisses",
   ↪  "Danielle");
}
/*
Hello Ulisses e null
Hello Ulisses e Danielle
*/
```

Parâmetros Opcionais Nomeados

- Uma chave ao redor de um parâmetro também indica que são opcionais. Entretanto, neste caso, o programador deverá fornecer o nome do parâmetro ao qual gostaria de fornecer um valor.

- Note especialmente que a ordem em que os parâmetros são declarados na assinatura da função não mais importa.

```
// Argumentos opcionais nomeados
void hello_amigos(String user,
     {String esposa, String
      ↪  amiga}){
 print("Hello $user, $esposa e
  ↪  $amiga");
}

void main() {
  // Argumentos opcionais nomeados
  hello_amigos("Ulisses");

   ↪  hello_amigos("Ulisses",esposa:"Da
  hello_amigos("Ulisses",
   ↪  esposa:"Dani",
```

```
                    amiga:"Pri");
  hello_amigos("Ulisses",
   ↪  amiga:"Pri",
                    esposa:"Dani");
}
/*
Hello Ulisses, null e null
Hello Ulisses, Dani e null
Hello Ulisses, Dani e Pri
Hello Ulisses, Dani e Pri */
```

Parâmetros com Valores Default

- Nos casos dos parâmetros opcionais vistos acima, um valor null é atribuído quando nada é fornecido na invocação. Entretanto, a própria função pode ter um valor default para esses casos.
- O valor default é informado no momento da declaração do parâmetro opcional dentro das chaves.

```
// Argumentos default
void hello_todos(String user,
     {String esposa = "Dani",
      String amiga = "Pri",
```

```
        String cachorro = "Snoop",
        String gato = "Nini" } ) {
    print("Hello $user, $esposa,
     ↪  $amiga, $cachorro, $gato");
}

void main() {
    // Invocando com argumentos
     ↪  default
    hello_todos("Ulisses");
    hello_todos("FT", amiga:"Mari");
    hello_todos("FT",
     ↪  cachorro:"Boró");
    hello_todos("FT", gato:"Mingau");
}
/*
Hello Ulisses, Dani, Pri, Snoop,
 ↪  Nini
Hello FT, Dani, Mari, Snoop, Nini
Hello FT, Dani, Pri, Boró, Nini
Hello FT, Dani, Pri, Snoop, Mingau
*/
```

0.8 Tratamento de Exceções

- Durante a execução do programa, erros podem fazer o nosso aplicativo abortar. É função do programador tratar os erros para não causarem prejuízos.
- As palavras reservadas try, on, catch e finally.
- O bloco try deve ser usado para delimitar onde o erro irá ocorrer.
- O bloco on especifica um tipo de erro.

```
int x = 12;
int y = 0;
try {
    int res = x ~/ y;
    print("Resultado: $res");
} on
 ↪  IntegerDivisionByZeroException{
    print('Divisão por zero');
}
// Divisão por zero
```

- O bloco catch é usado quando uma instância de uma classe de exceção é necessária.

```
int x = 12;
int y = 0;
try {
    int res = x ~/ y;
    print("Resultado: $res");
} catch(e) {
    print(e);
}
// IntegerDivisionByZeroException
```

- A união de on e catch também é possível.

```
int x = 12;
int y = 0;
try {
    int res = x ~/ y;
    print("Resultado: $res");
} on
 ↪  IntegerDivisionByZeroException
 ↪  catch(e) {
    print(e);
}
// IntegerDivisionByZeroException
```

- O bloco finally contém código que deve ser usado independente da ocorrência de uma exceção, após try/on/catch.

```
int x = 12;
int y = 0;
try {
   int res = x ~/ y;
   print("Resultado: $res");
} on
 ↪  IntegerDivisionByZeroException
 ↪  {
   print('Divisão por zero');
} finally {
   print('Bloco finally invocado');
}
 /*
Divisão por zero
Bloco finally invocado
*/
```

Lançando Exceções

- Em várias situações, um ponto do código encontra uma situação anormal, mas não pode tratar a

situação, então lançará uma exceção para que seja possível tratar o problema em outro local.

- O comando throw serve para isso, para iniciar uma exceção que se não for tratada ocasionará a saída abrubta do programa.

```
main() {
  try {
    setIdade(-2);
  }
  catch(e) {
    print('Error:
     ↪  ${e.getMessage()}');
  }
}
void setIdade(int age) {
   if(age<0) {
      throw new AgeException();
   }
}
class AgeException implements
 ↪  Exception{
  String errorMessage() {
    return 'Idade negativa';
```

```
    }
}
```

0.9 Orientação a Objetos em Dart

Classes

- Em Dart, as classes são criadas com class.

```
class Professor { }
```

Atributos

- As classes podem ter atributos. Se você não instanciar os atributos, eles terão o valor null.

```
class Professor {
  String nome;
  int idade;
}
```

- Atributos podem ser estáticos, o que significa que você pode utilizá-los sem instanciar objetos.

```
class Professor {
  static String vinculo =
   ↪  "Unicamp";
}
```

Métodos

- Classes podem definir seu próprio comportamento por meio de métodos. Esses métodos também podem ser estáticos.

```
class Professor {
  String nome;
  String sobrenome;

  String nomeProfessor(){
    return "$nome $sobrenome";
  }

  static String getVinculo(){
    return "Unicamp";
  }
}
```

Construtores

- Construtores geram instâncias. Caso você não defina um, Dart assume que existe um construtor que não recebe parâmetros.

- Existem vários tipos de construtor em classe, você pode definir um construtor com o mesmo nome da classe e sem nenhum parâmetro, isso será chamado de construtor default.

```
main() {
  Professor p = new Professor();
  print(p.nome); // Ulisses
}
class Professor {
  String nome;
  Professor(){
    this.nome = "Ulisses";
  }
}
```

- Também é possível definir um construtor com parâmetros, para que seja possível instanciar um objeto passando alguns valores de inicialização.

- Em ambos os casos, o construtor sempre tem o mesmo nome da classe e não possui identificador de tipo de retorno.

```
class Professor {
  String nome;
  String sobrenome;

  Professor(String n, String sn) {
    nome = n;
    sobrenome = sn;
  }
  String displayName() {
    return "$nome $sobrenome";
  }
}
```

- Um construtor que apenas atribui valores a atributos é um padrão muito comum. Por isso, foi criado um atalho. A palavra reservada this referencia a instância atual.

```
class Professor {
 String nome;
 String sobrenome;
```

```
  ↪   Professor(this.nome,this.sobrenome
}
```

- Em Dart, pode existir apenas um construtor default ou um construtor com parâmetros.
- Necessitadas outras formas de instanciar objetos, é preciso criar construtores nomeados.

```
Professor p = new
 ↪  Professor.meuProprioConstrutor();
print(p.nome); // Ulisses

class Professor {
 String nome;
 Professor(this.nome);
 Professor.meuProprioConstrutor(){
   nome = "Ulisses";
}}
```

Herança

- Herança é feita com a palavra reservada extends.
- A palavra reservada super referencia a classe mãe. No início do construtor, deve haver uma chamada para um construtor da classe mãe. A mesma será feita de forma implícita para o construtor default caso o usuário não a declare.
- No caso de a classe mãe não ter um construtor default, então a classe filha deverá invocar super com os parâmetros necessários.

```
main(){
  Professor p = new Professor(
    "Ulisses", "Dias"
  );
  /*
    Nova pessoa
    Novo professor
    */
  Subst s = new Subst();
  /*
    Nova pessoa
    Novo professor
```

```
    Novo Substituto
    */

  print(p.displayName());
  // Dr. Ulisses Dias
  print(s.displayName());
  // Dr. Alan Tal
}

class Pessoa {
  String nome;
  String sobrenome;

  Pessoa(){
    print("Nova pessoa");
  }
  String displayName() {
    return "Dr. $nome $sobrenome";
  }
}

class Professor extends Pessoa {
  //Professor(String n, String
   ↪  sn):super(){
  Professor(String n, String sn){
```

```
    print("Novo professor");
    this.nome = n;
    this.sobrenome = sn;
  }
}

class Subst extends Professor {
  Subst() : super("Alan","Tal"){
    print("Novo Substituto");
  }
  Subst.myConst(
    String n,
    String sn): super(n, sn);
}
```

- A mesma regra de chamada de um construtor da classe mãe vale para construtores com nome. Nesse caso, uma chamada super foi feita em Subst.myConst.

- Uma classe filha não pode acessar o construtor da classe mãe com uma chamada a super() no corpo das funções, por isso colocamos após o ":" na sintaxe vista nos códigos.

- Você pode usar o construtor com nome da classe mãe ao invés do default ou do parametrizado. Neste caso, apenas o construtor com nome da classe mãe será invocado na instanciação da classe filha.

```
class Pessoa {
  String nome;
  String sobrenome;
  Pessoa.build(this.nome,
               this.sobrenome);
  String displayName() {
    return "$nome $sobrenome";
  }
}
class Professor extends Pessoa {
  Professor(n,
   ↪  sn):super.build(n,sn);
  String displayName() {
    return "Dr. $nome $sobrenome";
  }
}
```

Visibilidade (Getters e Setters)

- Em Dart, todos os membros de uma classe são públicos a não ser que comecem com um *underscore*, o que os tornam privados dentro do arquivo `.dart`.
- Métodos **getters** são criados automaticamente para todos os atributos públicos e métodos **setters** são criados para todos os atributos públicos não marcados como final.
- No caso dos membros privados, você deverá criar um getter e um setter para serem acessíveis de fora do arquivo.
- As palavras reservadas set e get servem para criarmos **getters** e **setters**.

```
class Professor {
 String _nome;
 String _sobre;

  ↪  Professor(this._nome,this._sobre)

 String get nome => "$_nome
  ↪  $_sobre";
 set nome(n) {
```

```
    List strings = n.split(" ");
    _nome =strings[0];
    _sobre=strings.sublist(1).join("
     ↪  ");
  }
}

main() {
    Professor p = new
     ↪  Professor("Ulisses",

                                    ↪  "Dias
    print(p.nome); // Ulisses Dias
    p.nome = "Danielle Dias";
    print(p.nome); // Danielle Dias
}
```

Classes Abstratas

- Classes abstratas não serão instanciadas.
- Métodos sem corpo são abstratos.

```
/*
    Classe abstrata, não pode ser
```

```
  instanciada.
*/
abstract class Professor {
  /* Método abstrato, precisa ser
  sobrescrito na classe filha */
  verificaID();

  /* Método concreto, não precisa
  ser sobrescritno na classe filha
 ↪  */
  verificaNome(){
    print("Ulisses Dias");
  }
}
```

Interfaces

- Em Dart, não temos uma palavra reservada para declarar interfaces como acontece com Java. Nesse caso, qualquer classe possui uma interface própria que pode ser implementada por outras classes.

- A palavra reservada implements serve para dizer que uma classe irá implementar a interface própria de outra classe.

- Nesse caso, nenhum dos métodos serão herdados, mas todos deverão ser reescritos. Uma consequência disso é que você não poderá invocar os métodos da superclasse por meio de super.
- Uma classe pode implementar mais de uma superclasse.

```
int main(){
  Pessoa p = new Professor();
  Funcionario f = new Professor();
}
class Pessoa {
  void nasce(){
  }
  void cresce(){
  }
  void morre(){
  }
}
class Funcionario{
  void trabalha(){
  }
}
class Professor implements
 ↪  Pessoa,Funcionario {
```

```
    /* A implementação dos métodos
       a seguir é mandatória.*/
    void nasce(){
    }
    void cresce(){
    }
    void morre(){
    }
    void trabalha(){
    }
}
```

Contexto Estático

- A palavra reservada static serve para gerar contexto estático e pode ser aplicada tanto a métodos quanto a atributos.
- Membros estáticos são armazenados em memória apenas uma vez, independente do número de instâncias da classe.
- Membros estáticos só podem ser acessados usando o nome da classe.

- Instâncias de uma classe não podem acessar os métodos estáticos diretamente, o que difere do que ocorre com Java. Isso criar uma separação entre o contexto da instância e o contexto da classe.
- Atributos estáticos podem ser mudados inadvertidamente em vários lugares do código, por isso são normalmente declarados como const. Isso evita que sejam usados como variáveis globais, mas como constantes globais.
- De dentro de um método estático, você não poderá acessar os membros de instância da mesma classe, apenas os membros estáticos.

```
int main(){
  /*Podemos acessar membros
   ↪  estáticos
    sem precisar de instâncias */
  print(Professor.pi);

  // Instanciando professor
  Professor p = new Professor();

  /* Não podemos acessar membros
```

```
    estáticos com as instâncias*/
    //print(p.pi);

    //As chamadas a seguir são
     ↪  possíveis
    Professor.nasce();
    p.cresce();

}
class Professor  {
    // Atributo estático
    static double pi = 3.14;

    // Atributo de instância
    double altura = 1.74;

    // Método estático
     static void nasce(){
      print(pi);

        // Não podemos acessar altura
        // print(altura);
    }

    // Método de instância
```

```
  void cresce(){
    /* Podemos acessar as variáveis
    estáticas e de instâncias.  */
    print(pi);
    print(Professor.pi);
    print(altura);
  }
}
```

0.10 Introdução a Widgets

- Existem centenas de widgets em Flutter, sendo impossível e desnecessário conhecer todos. Vamos conhecer os mais usados em algumas categorias.

Top-Level Widgets

- A escolha impacta no padrão a ser seguido.
- De um modo, o widget **MaterialApp** é o mais usado para usar o padrão **MaterialDesign** do google. Ele funcionará em android e iOS.
- **MaterialApp**: possui alguns atributos (propriedades) comumente configurados.
  - **title**: usado para que o SO identifique o aplicativo para o usuário pelo nome.
  - **theme**: usa um objeto **ThemeData** para especificar cores e outras propriedades relacionadas ao *look and feel* do aplicativo.
  - **home**: tela que o usuário visualizará por primeiro e ocupará inicialmente o centro do aplicativo. Em geral, coloque um Scaffold nessa posição.

```
MaterialApp(
  title: 'Flutter Demo',
  theme: ThemeData(
    primarySwatch: Colors.blue,
    visualDensity:
     ↪  VisualDensity.adaptivePlatformD
  ),
  home: MyHomePage(title: 'Page'),
);
```

Gerenciadores de Layout Principais

Widgets nesta categoria ajudam a organizar a interface e a estrutura do aplicativo. Os mais comuns são: **Scaffold**, **Center**, **Row**, e **Container**.

Scaffold

Implementa a estrutura básica da tela, e gerencia elementos como: barra de navegação, *drawers*, barra de botões na parte inferior, botão flutuante, ...

- Existem vários scaffolds, independente da sua escolha, eles deverão ter um widget filho especificado usando a propriedade body.

- Em **body**, você adicionará apenas um widget filho que aparecerá no centro da tela principal.

```
Scaffold(
  appBar: AppBar(
    title: Text(widget.title),
  ),
  body: Center(),
  floatingActionButton:
   ↪  FloatingActionButton(
    onPressed: () {},
    tooltip: 'Increment',
    child: Icon(Icons.add),
  ),
);
```

Center

Possui apenas um filho e centralizará esse filho na tela. O widget **Center** será tão grande quanto possível (*match_parent*).

```
Center(
  child:Column()
)
```

Row e Column

Dois widgets com propriedades similares, exceto pelo fato de o primeiro posicionar os filhos horizontalmente e o segundo posicionar os filhos verticalmente.

- Em conjunto, esses widgets permitem organizar o layout da tela como um grid (tabela).
- Podem ter vários filhos no atributo **children**.
- **mainAxisAlignment** permite definir o alinhamento dos filhos. Podemos atribuir **mainAxisAlignment.center** para centralizar os filhos.
- sobre **Row**, não haverá *scroll* na horizontal, pois é um erro de *design* se ocuparmos mais espaço à esquerda ou à direita da tela.
- No caso de faltar espaço abaixo da tela, é possível adicionar uma barra de *scroll* com o widget **SingleChildScrollView** como pai de **column**.
- No caso de as widgets ocuparem menos espaço que a tela, você poderá redistribuir o excedente entre as views com a widget **Expanded**.
- Você pode colocar **Row** dentro de **Column** e vice-versa para gerar estruturas complexas.

```
Row(
  children: [
    Text("Child1"),
    Expanded(child:
     ↪  Text("Child2")),
    Text("Child3")
  ],
  mainAxisAlignment:
       MainAxisAlignment.center,
)
```

Container

Combina o que outros widgets têm a oferecer em um único pacote. Por exemplo, ele pode substituir **Padding** para adicionar margens.

```
/*
  Abaixo temos um exemplo de
  Padding.
*/
Padding(
  padding: EdgeInsets.all(20.0),
  child: Text("Child2")
```

```
)

/*
  Abaixo um código usando
  Container que gera o
  mesmo resultado.
*/
Container(
  padding: EdgeInsets.all(20.0),
  child: Text("Child2")
),
```

Container também pode substituir o pacote **Transform** em transformações simples como rotações, translações, mudanças de escala, ...

- Note a seguir que ao usar o atributo **transform** do **Container** utilizamos multiplicação de matrizes, o que é mais difícil do que simplesmente dizer que queremos dobrar o tamanho.
- Note ao reproduzir o exemplo abaixo que o texto não será reposicionado. Uma funcionalidade que pode atrapalhar no caso de textos, mas que ajudaria muito no caso de imagens independentes.

```
/*
  Abaixo um exemplo de mudança
  de escala.
*/
Transform.scale(
  scale: 2,
  child: Text("Child2")
)
/*
  Abaixo o mesmo exemplo com
  o Container. Note que inclusive
  mantivemos o Padding do exemplo
  anterior
*/
Container(
  transform: new
   ↪  Matrix4.identity()..scale(2.0),
  padding: EdgeInsets.all(20.0),
  child: Text("Child2")
)
```

Outros Widgets Específicos

Vamos adicionar agora alguns widgets que lhe ajudarão muito em situações específicas.

ConstrainedBox

Adiciona restrições como tamanho mínimo ou máximo. As propriedades são **minWidth**, **minHeight**, **maxWidth**, **maxHeight**.

```
ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 60
  ),
  child: Text("Child2")
)
```

FittedBox

Muda a escala e reposiciona os objetos relativamente ao próprio FittedBox. Parece mais natural de usar do que o **Transform** visto anteriormente.

- Normalmente, o **FittedBox** é usado em conjunto com o **ConstrainedBox**.
- O código a seguir muda o tamanho e reposiciona o texto para manter centralizado. Além disso, não é

preciso definir dimensões ou razão de aumento de escala, o aumento respeitará o *aspect ratio*.

```
ConstrainedBox(
  constraints:BoxConstraints(
    minWidth: 200.0
  ),
  child: FittedBox(
    fit: BoxFit.fill,
    child: Text("Child2")
  )
)
```

RotatedBox

Rotaciona objetos de uma forma simples. Entretanto, é bastante limitado no que pode fazer.

- A seguir, **quarterTurns** é o número de rotações de 90 graus no sentido horário. Se você precisar de graus arbitrários, então use **Transform**.

```
RotatedBox(
  quarterTurns: 3,
  child:Text("Child2")
)
```

SizedBox

Força os filhos a terem largura e altura específicos. No entanto, este widget não implica em mudança de escala de qualquer tipo, apenas reserva o espaço.

Por padrão, o conteúdo dentro do **SizedBox** será alinhado com o canto superior esquerdo.

```
SizedBox(
  width: 200, height: 400,
  child: Text("Child2")
)
```

Widgets para Efeitos Decorativos

- Alguns widgets adicionam efeitos decorativos à interface gráfica. Não vamos aprender a desenvolver interfaces bonitas, vamos apenas listar alguns itens que realçam os principais conteúdos.
- Os itens que veremos são **Divider** e **CardLayout**.

  Divider

- **Divider**: coloca uma reta horizontal, o que pode criar divisões lógicas.

- A seguir, **Divider** foi acoplado com o **Expander** para ocupar toda a largura da tela.

```
child: Column(
  children: [
    Text("Texto 1"),
    Expanded(
      child: Divider()
    ),
    Text("Texto 2")
  ]
)
```

Card

Caixa com cantos arredondados e com um sombreado nas laterais. As propriedades são:

- child: único filho que **Card** pode ter.
- color: cor de fundo.
- elevation: altera a sombra.
- shape: muda o arredondamento dos cantos.

```
Card(
 child: Column(children: [
     Text("InnerChild1"),
     Divider(),
     Text("InnerChild2")
 ], mainAxisSize:
  ↪  MainAxisSize.min),
 color: Colors.blueAccent,
 elevation: 50,
 shape: RoundedRectangleBorder(

   ↪  borderRadius:BorderRadius.circula
 )
)
```

Widgets de Navegação

- Alguns widgets facilitam navegação utilizando padrões difundidos em aplicativos para dispositivos móveis: **TabBar**, **BottomNavigationBar**, **NavigationDrawer**, ...

TabBar

implementa abas que ficam na parte superior da janela principal. A cada momento, apenas uma aba pode estar visível.

- O equivalente do **TabBar** no iOS é o **CupertinoTabBar**.
- Uma **TabBarView** é uma pilha de telas (ou views) em que uma está visível a cada momento.
- A maneira de tornar uma tela visível é interagindo com a **TabBar** criada na **AppBar**.
- Para que a navegação funcione, deve existir um **DefaultTabController**, normalmente como filho de **MaterialApp** e pai de **Scaffold**. Não esqueça desse detalhe.
- O número de abas existente deve ser colocado no parâmetro **length** do **DefaultTabController**.

```
MaterialApp(
 title: "Meu primeiro TabBar",
 home: DefaultTabController(
  length: 3,

  child: Scaffold(

   body: TabBarView(children: [
    Center(child: Text("Anúncio")),
    Center(child:
     ↪  Text("Aniversário")),
    Center(child: Text("Tempo"))
   ]),
   appBar: AppBar(
    title: Text("Meu Primeiro
     ↪  TabBar"),
    bottom: TabBar(tabs: [
     Tab(icon:
      ↪  Icon(Icons.announcement)),
     Tab(icon: Icon(Icons.cake)),
     Tab(icon: Icon(Icons.cloud))
   ])),
  ),
 ),
```

```
);
```

BottomNavigationBar

Permite colocar uma barra de botões na parte inferior da tela, o que permite uma limitada forma de navegação.

- Diferente do TabBar, o **BottomNavigationBar** precisará ser construído dentro de um **StatefulWidget**. Isso ocorre porque o **BottomNavigationBar** não é realmente um layout de navegação, mas pode ser implementado como se fosse.
- No código a seguir, vamos assumir que temos na classe **State** associada a um **StatefulWidget** uma declaração de atributo var _currentPage = 0;
- Além disso, temos um array de Text widgets que contém três widgets:

```
class _MyApp extends State {
  var _currentPage = 0;
  var _pages = [
    Text("Page 1 - Anúncios"),
    Text("Page 2 - Aniversários"),
    Text("Page 3 - Previsão do
    ↪  Tempo")
```

```
  ];
  ...
```

- Em **Scaffold**, devemos colocar a nossa declaração de **BottomNavigationBar**. Os atributos são:
  - **items**: é onde adicionamos os botões, o que permite incluir um texto e adicionar um ícone.
  - **currentIndex**: informamos a tela mostrada no momento.
  - **onTab**: informamos o comportamento quando o botão for clicado.

```
class _MyApp extends State {
  ...

home : Scaffold(
 body: Center(child :
  ↪  _pages.elementAt(_currentPage)),
 bottomNavigationBar:
  ↪  BottomNavigationBar(
  items : [
    BottomNavigationBarItem(
      icon :
       ↪  Icon(Icons.announcement),
```

```
        title : Text("Anúncios")
      ),
      BottomNavigationBarItem(
        icon : Icon(Icons.cake),
        title : Text("Aniversários")
      ),
      BottomNavigationBarItem(
        icon : Icon(Icons.cloud),
        title : Text("Previsão do
        ↪  Tempo")
      ),
    ],
    currentIndex : _currentPage,
    fixedColor : Colors.red,
    onTap:(int inIndex){
      setState(() {
          _currentPage = inIndex;
      });
    }
    )
)

...
}
```

0.11 Widgets de Entrada e Saída

Temos usado apenas **Text** como placeholder na tela, então vamos conhecer outras widgets que farão parte da Interface Gráfica.

Text

- Não temos usado muito os estilos de formatação do **Text**, mas existem alguns. O código a seguir é auto-explicativo.

```
Text("Prof. Ulisses Martins Dias",
  style: TextStyle(
    fontSize: 40,
    fontWeight: FontWeight.bold,
    color: Colors.blue,
    letterSpacing: 2, // Padrão 1
  )
)
```

Image

- Essa widget serve para adicionar uma imagem na tela.

- Existem vários construtores para **Image**, dependendo de onde a imagem será buscada. Vamos falar apenas de um deles por enquanto.

```
Image.asset('assets/images/name.jpeg',
  height: 300, fit:
   ↪  BoxFit.fitHeight)
```

- Para que a linha acima funcione, você deverá primeiro criar a pasta **assets** seguida da criação da pasta **images** dentro dela. Feito isso, colocar o arquivo de imagem.

- O próximo passo será avisar ao Flutter que temos essa pasta de imagens. Isso deve ser feito no arquivo **pubspec.yaml**. Será preciso adicionar a linha a seguir:

```
assets :
    - assets/images/
```

- Para que as imagens apareçam na tela de forma agradável uma série de tentativa e erro deve ser feita para se obter o ajuste correto.

- É possível colocar **Image** dentro de um **Container** para permitir usar o atributo **decoration**.
- Note o uso de **BoxDecoration** para criar um contorno com cantos arredondados ao redor da imagem. Várias outras estilizações podem ser feitas com esse widget e talvez seja o que você tentará por primeiro.
- Usamos **ClipRRect** para arredondar a borda da imagem. Com o **BoxDecoration** criamos um contorno com borda arredondada. Não seria agradável se a imagem não acompanhasse o contorno.
- A combinação do **BoxDecoration** com **ClipRRect** gera uma interface não baseada apenas em retângulos.

```
Container(
  /*
  Maneira clássica de adicionar uma
↪
  borda. Vamos simplesmente
↪   circular
  as bordas do container.
  */
  decoration: BoxDecoration(
```

```
    // Fazendo a borda circular.
    borderRadius:
     ↪  BorderRadius.circular(15),

    // Colocando borda em todos os
     ↪  lados.
    border: Border.all(
      color: Colors.black,
      width: 4.0,
    ),
  ),
  // Arredondando também a imagem.
  child: ClipRRect(
   borderRadius:
    ↪  BorderRadius.circular(10),
   child: Image.asset(
    'assets/images/ulisses.jpeg',
    height: 300, fit:
     ↪  BoxFit.fitHeight),
  ),
);
```

Form

- Um widget opcional, mas que possui algumas utilidades importantes, então usaremos como se fosse mandatório. Todos os campos de entrada serão filhos do nosso **Form** widget.
- A razão para usar **Form** é que oferece funcionalidades de salvar dados do formulário, resetar e validar o conteúdo.
- A propriedade **key** do formulário permite associar uma chave global (**GlobalKey**) para identificar o form, permitindo que a validação seja invocada em um passo posterior.

```
Form(
  key: _formKey,
  child: Column(
    children: <Widget>[
      // Adicionar TextFormFields e
      // RaisedButton.
    ]
  )
);
```

TextFormField

- Sempre que precisarmos obter informações dos usuários, criaremos uma classe simples que terá o mesmo número de atributos que os campos dos formulários. Dessa forma, um objeto da classe consolidará tudo o que foi inserido.

- Pediremos duas informações do usuário: login e senha. Nesse caso, uma classe que consolidaria essa informação seria:

```
class LoginData {
  String username = "";
  String password = "";

  doSomething() {
    print("Username: $username");
    print("Password: $password");
  }
}
```

- Um **TextFormField** irá criar uma tela onde o usuário poderá inserir informações. Alguns atributos são importantes:

- **keyboardType**: permite dizer o que pretendemos receber do usuário e o sistema colocará um teclado com teclas apropriadas. Por exemplo, datas, urls, ...
- **validator**: permite adicionar uma função para validar a entrada do usuário. Se a entrada for inválida, o validator retornará uma string contendo a mensagem de erro. Caso contrário, o validator retornará **null**.
- **onSave**: permite criar uma função que informa como os dados serão salvos. No nosso caso, salvaremos em um objeto da classe **LoginData** vista anteriormente.

```
TextFormField(
  keyboardType:
   ↪  TextInputType.emailAddress,
  validator: (String inValue) {
    if (inValue.length == 0) {
      return "Please enter
       ↪  username";
    }
    return null;
  },
```

```
  onSaved: (String inValue) {
    loginData.username = inValue;
  },
  decoration: InputDecoration(
    hintText: "asdf@asdf.com.br",
    labelText: "E-Mail address")
)
```

- Como segundo exemplo, a seguir criaremos um TextFormField para guardar a senha do usuário.
- Note o parâmetro **obscureText** sendo usado para colocar asterisco no lugar das teclas digitadas pelo usuário.
- O **validator** agora proibe senhas muito curtas.

```
TextFormField(
  obscureText: true,
  validator: (String inValue) {
   if (inValue.length < 10) {
    return "Mínimo 10 letras";
   }
   return null;
  },
```

```
    onSaved: (String inValue) {
      loginData.password = inValue;
    },
    decoration: InputDecoration(
      hintText: "Senha",
      labelText: "Senha")
)
```

RaisedButton

- Um formulário não estará completo sem um botão. Quando o usuário pressionar o botão, usaremos a referência que temos do formulário (a **GlobalKey** que mencionamos anteriormente para invocar os métodos **validate** e **save**.
- A invocação desses métodos irá desencadear uma chamada para os métodos adicionados nos parâmetros **validator** e **onSave**, respectivamente.

```
RaisedButton(
  child: Text("Log In!"),
  onPressed: () {
    /*
     A chamada abaixo invoca todos
 ↪   os
```

```
    validators. O valor "true" será
↪
    retornado se todos os
↪   validators
    retornarem nulo.
  */
   if
    ↪  (formKey.currentState.validate()
     formKey.currentState.save();
   }
  }
)
```

CheckBox

- Agora que entendemos como um formulário básico funciona, não teremos problemas em adicionar outros elementos para entrada de dados. Vamos começar com uma classe que irá consolidar todos os valores do formulário.

```
class OtherData {
  var checkboxValue = false;
  var switchValue = false;
  var sliderValue = .3;
```

```
  var radioValue = 1;

  doSomething() {
    print("CheckBox:
     ↪  $checkboxValue");
    print("Switch: $switchValue");
    print("Slider: $sliderValue");
    print("Radio: $radioValue");
  }
}
```

- Feito isso, vamos assumir que temos um objeto **otherData** em alum lugar visível do código. A seguir, um código de referência. Note o uso dos parâmetros **value** e **onChanged**.

```
Checkbox(
  value: otherData.checkboxValue,
  onChanged: (bool inValue) {
    setState(() {
        otherData.checkboxValue =
         ↪  inValue;
    });
})
```

- Caso você coloque a **CheckBox** dentro de uma **ListView** ou de um **BottomNavigationBar**, será preciso colocar algum pai da **CheckBox** na widget tree como **StatefulBuilder**. Caso contrário, os elementos não serão mostrados corretamente na interface gráfica.

- Um outro detalhe é que um **CheckBox** não possui um rótulo de texto, o que seria comum nesse tipo de componente. Nesse caso, você teria que criar um você mesmo colocando o **CheckBox** e um widget **Text** dentro de um **Row**.

Switch

- Um **Switch**, assim como o **CupertinoSwitch** é um CheckBox com um visual diferente. Ele se parece com um botão de ligar e desligar que vemos em aparelhos eletrônicos.

- Um ponto a se notar é que se **onChange** for nulo, então o **Switch** aparecerá desabilitado na tela e não receberá interação do usuário.

```
Switch(
  value: otherData.switchValue,
```

```
    onChanged: (bool inValue) {
      setState(() {
          otherData.switchValue =
           ↪  inValue;
      });
})
```

Slider

- Um widget que mostra uma linha e uma maçaneta (círculo no centro da linha) para que você possa mover para alguma posição. Isso permitirá definir um valor dentro de um intervalo.

- As propriedades **min** e **max** são usadas para gerenciar o intervalo, e a propriedade **value** define o valor atual.

- **onChange** é necessário para modificar o valor quando a maçaneta é movida. Outras propriedades podem ser úteis em casos específicos, como **onChangeStart** e **onChangeEnd** que permitem adicionar comportamento quando o usuário começa a mover o **Slider** e quando termina.

```
Slider(
  min: 0,
  max: 20,
  value: otherData.sliderValue,
  onChanged: (inValue) {
    setState(() =>
    ↪  otherData.sliderValue =
    ↪  inValue);
})
```

RadioButton

- Uma barra de botões circulares semelhante aos rádios de carros antigos. Quando um botão é pressionado, todos os outros pulam de volta.
- Um **RadioButton** é muito semelhante ao CheckBox, exceto pelo fato de que as opções são excludentes dentro de um mesmo grupo que você deve definir dentro da propriedade **groupValue**.
- **RadioButtons** não aparecem sozinhos, vários devem existir. Em especial, cada um dos **RadioButton** deve ter uma propriedade **value** diferente dentro de um **groupValue**.

```
Row(children: [
    Radio(
      value: 1,
      groupValue:
       ↪  otherData.radioValue,
      onChanged: (int inValue) {
       setState(() {
        otherData.radioValue =
         ↪  inValue;
       });
    }),
    Text("Opção 1")
]),
Row(children: [
    Radio(
      value: 2,
      groupValue:
       ↪  otherData.radioValue,
      onChanged: (int inValue) {
       setState(() {
        otherData.radioValue =
         ↪  inValue;
       });
    }),
```

```
    Text("Opção 2")
]),
Row(children: [
    Radio(
      value: 3,
      groupValue:
       ↪  otherData.radioValue,
      onChanged: (int inValue) {
       setState(() {
        otherData.radioValue =
         ↪  inValue;
       });
    }),
    Text("Opção 3")
])
```

Tooltip

- Uma maneira conveniente de mostrar ao usuário alguma informação não relacionada diretamente com a tela. O **Tooltip** mostra uma descrição de algum outro widget quando algum evento ocorre. Por exemplo, o clique longo por padrão.
- Alguns widgets como **Icons** já possuem uma propriedade **tooltip** com a funcionalidade, mas você pode

sempre criar um **Tooltip** quando essa propriedade não existir.

- Coloque mensagens informativas nos **Tooltips**, pensando principalmente naqueles como problemas de acessibilidade.

- Um **Tooltip** aparecerá normalmente logo abaixo do widget a que se refere, mas você pode usar **verticalOffset** para determinar a distância entre o **Tooltip** e o widget alvo.

```
Tooltip(
  message: "Nada bom sairá deste
   ↪  botão",
  child: RaisedButton(
    child: Text("Log In!"),
    onPressed: () {
      otherData.doSomething();
  }),
)
```

Dialog

- Um **Dialog** é um elemento popup que informa alguma coisa ao usuário ou pede alguma informação pontual. Os mais comuns são **SimpleDialog** e o **AlertDialog**. Abaixo um exemplo deste último.

```
AlertDialog(
  title: Text("Resposta
   ↪  Requerida"),
  content: Text("Você aceitaria?"),
  actions: [
    FlatButton("Sim"),
    FlatButton("Não"),
```

```
  ],
  elevation: 24.0,
  backgroundColor: Colors.blue,
  shape: CircleBorder()
)
```

- Note que um **AlertDialog** pode pedir informações dos usuários, normalmente uma resposta binária como a que vemos no código.
- Para mostrar o **AlertDialog** na tela, usamos a função **showDialog**, que recebe como parâmetros um **context** e um **builder** para retornar o **dialog** que queremos renderizar.
- O programador pode decidir se o usuário poderá fechar o **dialog** clicando fora da janela com o parâmetro booleano **barrierDismissible**.

```
showDialog(
  context: context,
  builder: (_) => AlertDialog(),
  barrierDismissible: false,
);
```

- Os dois blocos de códigos acima estavam mais interessados na legibilidade. Entretanto, você não será capaz de reproduzir um **AlertDialog** só com a descrição deles. Por isso, temos o código mais completo a seguir.

- Note especialmente o uso do método **pop** no objeto do tipo **Navigator**. Isso serve para remover a janela popup tão logo o usuário clique em alguma das opções.

```
RaisedButton(
 child: Text("Não Clique"),
 onPressed: () {
  return showDialog(
   context: context,
   builder: (BuildContext context)
    ↪  {
     return AlertDialog(
      title: Text("Foi avisado"),
      content: Text("Quer
       ↪  clicar?"),
      actions: [
       FlatButton(
        child: Text("Sim"),
```

```
            onPressed: () {
              // Faça algo

                ↪  Navigator.of(context).pop
            },
          ),
          FlatButton(
            child: Text("Não"),
            onPressed: () {
              // Faça algo

                ↪  Navigator.of(context).po
            },
          ),
        ],
        elevation: 24.0);
    },
    barrierDismissible: true);
})
```

SnackBar

- Componente que mostra uma mensagem no rodapé da tela por um período de tempo. O usuário poderá clicar para fazer a mensagem sumir.

- Para mostrar uma mensagem no **SnackBar**, usamos uma chamada para **Scaffold.of** para obter uma referência ao **Scaffold**, que por sua vez possui um método chamado **showSnackBar**.
- A propriedade **action** do **SnackBar** é opcional, mas quando colocada apresenta um texto clicável. Tente não colocar comportamentos muito importantes neste código, pois não é o tipo de coisa que a maioria dos usuários verá.

```
RaisedButton(
  onPressed: () {

    ↪  Scaffold.of(context).showSnackB
     SnackBar(
       backgroundColor:
        ↪  Colors.red,
       duration: Duration(seconds:
        ↪  5),
       content: Text("Obrigado"),
       action: SnackBarAction(
         label: "Volte Sempre!",
         onPressed: () {
           print("Funcionou");
```

```
      }));
    },
    child: Text("Clique Neste")
)
```

BottomSheet

- Uma mistura de **Dialog** com **SnackBar**. O **BottomSheet** permite criar uma tela customizável da maneira que o programador bem entender.
- Da maneira que criaremos a seguir, a tela será persistente e só desaparecerá quando o usuário clicar em algum dos botões. O usuário poderá, no entanto, interagir com o restante do aplicativo.
- Se você quiser bloquear o restante do aplicativo, então você precisará utilizar um **ModalBottomSheet**.

```
RaisedButton(
 onPressed: () {
  showBottomSheet(
   context: context,
   builder: (BuildContext context)
    ↪ {
```

```
      return Row(
       children: [
        Expanded(
         child: Column(
          children: [
           Text("Professor
            ↪  favorito:"),
           FlatButton(
            child: Text("Guilherme"),
            onPressed: () {

              ↪  Navigator.of(context).po
           }),
           FlatButton(
            child: Text("Celmar"),
            onPressed: () {

              ↪  Navigator.of(context).po
           }),
           FlatButton(
            child: Text("Ulisses"),
            onPressed: () {

              ↪  Navigator.of(context).po
           }),
```

```
            ],
            mainAxisSize:
             ↪  MainAxisSize.min,
          ),
        )
      ],
    );
  },
 );
},
child: Text("Clique Neste")
)
```

0.12 Animações

- Usuários esperam que os aplicativos tenham algum tipo de animação após transições de tela. Vários widgets já possuem alguma animação que de tão comuns quase não notamos.

- Você pode criar suas próprias animações caso necessite. Entretanto, evite usar demais, pois pode ter o efeito contrário ao que você imagina. Vamos ver os métodos mais tradicionais.

- Não esqueça de utilizar o **setState** sempre que fizer uma mudança de configuração para engatilhar as animações. Isso nos obriga a dizer que você usará **StatefulWidgets**.

AnimatedContainer

- Use quando a sua animação for simples, basta dizer as propriedades de início e de fim, o widget se encarregará de fazer as interpolações e criar o efeito de animação.

- No código a seguir, criamos um **AnimatedContainer** com as propriedades de **cor**, **largura** e **altura** configuradas como **amarelo**, **200** e **200**, respectivamente.

- Quando o usuário clicar no botão, essas propriedades mudarão para **vermelho**, **400** e **400**. A duração da animação seria de 1 segundo, conforme indicado na propriedade **duration**.

```
var _color = Colors.yellow;
var _height = 200.0;
var _width = 200.0;
```

```
child: Column(
  mainAxisAlignment:
   ↪  MainAxisAlignment.center,
  children: [
    AnimatedContainer(
      duration: Duration(seconds:
       ↪  1),
      color: _color,
      width: _width,
      height: _height),
    RaisedButton(
      child: Text("Animate!"),
      onPressed: () {
        setState(() {
            _color = Colors.red;
            _height = 400.0;
            _width = 400.0;
        });
    })
])
```

AnimatedCrossFade

- Como o próprio nome indica, faz com que um widget desapareça gradualmente enquanto outro apa-

rece.

- Nesse caso, você deverá passar dois filhos para o widget nas propriedades **firstChild** e **secondChild**.
- A propriedade **crossFadeState** recebe qual dos dois filhos deve ser mostrado em um determinado momento. Observe no código a seguir o uso do operador "**?**" para criar um comando condicional de uma única linha.
- A propriedade **duration** informa o tempo da animação, do mesmo modo como foi feito no AnimatedContainer.

```
bool crossFadeFirst = true;

AnimatedCrossFade(
  duration: Duration(seconds: 2),
  firstChild: FlutterLogo(
    style:
     ↪  FlutterLogoStyle.horizontal,
    size: 100.0),
  secondChild: FlutterLogo(
    style:
     ↪  FlutterLogoStyle.stacked,
```

```
      size: 100.0),
    crossFadeState: crossFadeFirst
    ? CrossFadeState.showFirst
    : CrossFadeState.showSecond,
),

RaisedButton(
    child: Text("Animate!"),
    onPressed: () {
      setState(() {
          crossFadeFirst = false;
      });
})
```

AnimatedDefaultTextStyle

Opção para animação de textos. Funciona de modo similar aos anteriores. Será assumido não haver necessidade de explicar muito o código abaixo.

```
var _fontSize = 10;
var _color = Colors.yellow;

AnimatedDefaultTextStyle(
  duration: const Duration(seconds:
  ↪  1),
  style: TextStyle(
    color: _color,
    fontSize: _fontSize),
  child: Text("Ulisses Martins
  ↪  Dias")),
RaisedButton(
  child: Text("Animate!"),
  onPressed: () {
    setState(() {
        _color = Colors.red;
        _fontSize = 40;
    });
})
```

Além dos widgets mencionados nesta seção, recomenda-se que você pesquise na internet os seguintes: AnimatedOpacity, AnimatedPositioned, PositionedTransition, SlideTransition, AnimatedSize, ScaleTransition, SizeTransition e RotationTransition. Esses widgets não diferem muito do que já vimos.

0.13 Visualização de Dados

Se você tem que mostrar vários dados aos usuários em um aplicativo, então terá que organizá-los de alguma forma para que a visualização seja agradável na tela pequena de um celular. Nesses momentos que os widgets vistos nesta seção serão utilizados.

Table

- A forma mais simples de visualizar os dados é preenchendo uma planilha. Nesse contexto, o widget **table** é útil e funciona de modo parecido com o que você faria para web.

- Você irá simplesmente definir as bordas da tabela (ou manter o padrão sem bordas) para depois adicionar as linhas usando **TableRow**. Dentro das linhas, você pode colocar qualquer widget que você quiser.

```
Table(
  border: TableBorder(
    top: BorderSide(width: 2),
    bottom: BorderSide(width: 2),
    left: BorderSide(width: 2),
    right: BorderSide(width: 2)),
  children: [
    TableRow(
      decoration: BoxDecoration(
        borderRadius:
        ↪  BorderRadius.circular(15),
        border: Border.all(
          color: Colors.black,
          width: 1.0,
      )),
      children: [
        Center(
          child: Padding(
            padding:
            ↪  EdgeInsets.all(10),
            ↪  child: Text("1"))),
        Center(
          child: Padding(
```

```
                padding:
                 ↪  EdgeInsets.all(10),
                child:
                 ↪  Icon(Icons.cake))),
          Center(
            child: Padding(
              padding:
               ↪  EdgeInsets.all(10),
               ↪  child: Text("3")))
    ])
])
```

DataTable

- Um outro padrão que você já deve ter visto várias e várias vezes. Um **DataTable** requer que você informe quais são as colunas com um **DataColumn** e que depois comece a informar as linhas, uma após a outra, com **DataRow**.

- Você pode adicionar o parâmetro **sortColumnIndex** para indicar por qual coluna a tabela está ordenada. Note que isso é só um indicador, não passando de perfumaria, você é quem deve garantir que os dados estão realmente ordenados.

```
DataTable(sortColumnIndex: 1,
 ↪  columns: [
    DataColumn(label:
     ↪  Text("Nome")),
    DataColumn(label:
     ↪  Text("Sobrenome"))
  ],
  rows: [
    DataRow(cells: [
        DataCell(Text("Ulisses")),
        DataCell(Text("Dias"))
      ]
    ),
    DataRow(
      cells: [
        DataCell(Text("Guilheme")),
        DataCell(Text("Coelho"))
      ]
    ),
  ]
)
```

ListView

- Talvez o widget mais onipresente em aplicativos móveis. É tão comum que quase não notamos a presença dele em vários aplicativos que usamos com frequência.
- Em sua forma mais simples, você declarará o widget **ListView** e depois colocará como filho uma série de **ListTiles**. Entretanto, note que os filhos de uma **ListView** podem ser qualquer coisa que você desejar.
- De forma resumida, um **ListTile** é um widget que possui altura fixa e pode conter texto e ícones antes (*leading*) ou depois (*trailing*) do texto.
- Uma **ListView** pode fazer scroll na vertical ou na horizontal, dependendo do que você colocar na propriedade **scrollDirection**.
- Caso você precise fazer paginação na sua **ListView**, existe a **PageView**, uma ótima opção para mostrar vários elementos na tela sem perder tanto em desempenho.

```
ListView(children: [
    ListTile(
```

```
        leading: Icon(Icons.gif),
        title: Text("1"),
        onTap: () {
          print("Colocando na
           ↪  linha");
        },
        trailing: Icon(Icons.pets),
      ),
      ListTile(
        leading:Icon(Icons.book),
        title: Text("2")),
      ListTile(
        leading: Icon(Icons.call),
        title: Text("3")),
      ListTile(
        leading: Icon(Icons.dns),
        title: Text("4")),
      ListTile(
        leading: Icon(Icons.cake),
        title: Text("5"))
]);
```

- No caso de termos muitos elementos para colocar na **ListView**, é interessante usar um construtor que

garantar um pouco mais de eficiência.

- No exemplo a seguir, queremos criar uma **ListView** com **1000** termos. Não queremos colocar todos esses termos na memória, então usaremos um construtor que vai garantir a criação apenas daqueles visíveis ao usuário.

- Caso o usuário faça uma rolagem para outro ponto da **ListView**, o construtor vai tratar de gerar novos dados para popular a **ListView**.

```
var items = [];
for (var i = 0; i < 1000; i++) {
  items.add("Item $i");
}

ListView.builder(
  itemBuilder: (context, index) {
    return ListTile(
      title: Text(items[index]
      )
    );
  }
)
```

ListView + Drawer = Navegação

- Um uso muito comum de uma **ListView** é o pareamento feito com o **Drawer** do widget **Scaffold**. Isso permite criar o menu lateral tão comum em aplicativos.

- Note o uso de **DrawerHeader** como primeiro filho da **ListView**. Isso permite criar um espaço no canto superior esquerdo do **Drawer** para adicionar informações do usuário, por exemplo.

- Note também a invocação de **Navigator.pop(context);** para fechar a tela do **Drawer** quando o primeiro item do menu for clicado. Você deverá adicionar essa invocação no parâmetro **onTap** de todos os **ListTiles**.

```
ListView(children: [
    DrawerHeader(
      child: Text('Drawer Header'),
      decoration: BoxDecoration(
        color: Colors.blue,
      ),
    ),
    ListTile(
```

```
        leading: Icon(Icons.gif),
        title: Text("1"),
        onTap: () {
          Navigator.pop(context);
        },
        trailing: Icon(Icons.pets),
      ),
      ListTile(
        leading:Icon(Icons.book),
        title: Text("2")),
      ListTile(
        leading: Icon(Icons.call),
        title: Text("3")),
      ListTile(
        leading: Icon(Icons.dns),
        title: Text("4")),
      ListTile(
        leading: Icon(Icons.cake),
        title: Text("5"))
]);
```

0.14 Paralelismo

- Em programas Android nativos, existem pelo menos três tipos de threads: **RenderThread**, **MainThread** e **OutrasThreads**.

- A **MainThread** é aquela que gerencia os cliques dos botões, a interação com a tela, as chamadas do ciclo de vida, e assim por diante. Em geral, não colocamos nada muito pesado na **MainThread** sob o risco de congelar a interface gráfica.

- Nesse caso, em programação para android nativo, usamos threads em paralelo para operações longas, para operações que precisam executar em paralelo (como músicas, por exemplo), para operações que usam rede (internet, download, ...), operações em arquivos e operações em bancos de dados.

- Em android nativo, fazemos o uso das classe **Thread**, **AsyncTask**, **Service**, ou da interface **Runnable**.

- Em flutter, todos os códigos executarão na **MainThread**. Nesse caso, um código muito lento pode ter o efeito de bloquear a interface gráfica piorando a

experiência do usuário, então como podemos fazer para que isso não aconteça?

- A solução gira em torno das seguintes palavrinhas: **Future**, **Await** e **Async**. A resposta do flutter consiste em adicionar à sintaxe essas palavras reservadas que indicam que algumas operações não deverão bloquear a **MainThread**.

- **Future**: um tipo que permite "**prometer**" a entrega de um objeto no futuro. Quando invocamos uma função que retorna Future, não podemos associar o valor prometido imediatamente, porque o valor é só uma promessa. Se quisermos imprimir o valor, e não a promessa, usamos **await**.

```
operacaoLonga() {
  Future<String> result =
   ↪  Future.delayed(
    Duration(seconds: 5), () {
      return "Mensagem aguardada";
    }
  );
  return result;
}
```

- **Await**: indica que queremos aguardar que um promessa se concretize antes de prosseguir com o có-

digo. Nesse caso, indica que queremos pagar o preço de uma operação longa para ter o objeto, e não uma promessa apenas.

- A função onde a palavra reservada **await** está ficará bloqueada aguardando a concretização da promessa. Entretanto, não é sensato que ela bloqueie toda a **MainThread**. Nesse caso, deverá ser colocada em paralelo.

```
var aux = await operacaoLonga();
```

- **Async**: a palavra reservada "async" deverá ser usada para inform que aquela função (notadamente a que usa o await) não irá bloquear a MainThread, o código que invocou a função com async irá prosseguir sem aguardar o término dela.

```
esperaOperacaoLonga() async {
  var aux = await operacaoLonga();
  print("O valor aguardado é:
   ↪  $aux");
}
```

- Outra forma de esperar a concretização de um promessa consiste em utilizar a API "then" do objeto Future. Esse método permite que passemos uma função por parâmetro que será executada quando Future se concretizar no objeto que está destinado a ser.

```
var aux = operacaoLonga();
aux.then((conc) {
    print("Valor: $conc");
});
```

0.15 Banco de Dados

- Utilizaremos o banco de dados **SQLite**, que possui uma série de características interessantes:

  Self-contained : suporte mínimo de biliotecas externas.

  Serverless : o banco na verdade lê e escreve diretamento de um arquivo no disco.

  Zero-configuration : não exige instalação nem configuração, basta criar um arquivo **.sqlite** e começar a usar.

Transacional : as mudanças acarretadas por uma transação são todas concretizadas, ou nenhuma é concretizada.

- O plugin que utilizaremos para nos comunicar com o **SQLite** se chama **SQFLite**, note a presença do "F".
- Esse plugin permite acessar bancos de dados **SQLite** tanto em aplicativos android quanto IOS.
- Esse plugin permite salvar objetos **Map<String,dynamic>** no banco de dados e também retornar objetos **Map<String, dynamic>**.
- Para utilizar o plugin, precisamos primeiramente declarar no arquivo **pubspec.yaml**. Em geral, para usar bancos de dados colocamos três plugins: **sqflite**, **path_provider**, **intl**.

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: 1.3.1
  path_provider: 1.6.21
  intl: 0.16.1
```

- O **path_provider** serve para podermos acessar o diretório particular do aplicativo (podemos acessar a memória externa também no caso de android) e alocaros o banco de dados em algum lugar.
- O **intl** possui diversas funções muito úteis para formatação de datas e horas.

Classe Singleton

- Para centralizar o acesso ao banco, criaremos uma classe **singleton** que instanciará apenas um único objeto. Esse efeito pode ser feito declarando uma classe com um construtor privado e colocando a única instância dessa classe como um atributo estático.

```
class DatabaseHelper {
  static DatabaseHelper helper =
   ↪  DatabaseHelper._createInstance();

  DatabaseHelper._createInstance();
}
```

- Dentro dessa classe Singleton iremos ter apenas uma instância de **Database**, um objeto que nos permite

manipular o banco de dados. Essa instância será inicializada com uma chamada a **openDatabase**, um método **async**.

- A função **openDatabase** recebe como parâmetro uma função que irá criar fornecer uma primeira string **SQL** para executar a criação das tabelas.

```
static Database _db;

Future<Database> get database
 ↪  async {
  if (_db == null) {
    _db = await initDB();
  }
  return _db;
}

Future<Database> initDB() async {
  Directory dir = await
   ↪  getApplicationDocumentsDirectory(
  String path = dir.path +
   ↪  "notes.db";
  var notesDB = await
   ↪  openDatabase(
```

```
    path,
    version: 1,
    onCreate: _createDb
  );
  return notesDB;
}

void _createDb(Database db,
               int newVersion)
                ↪  async {
  await db.execute("CREATE TABLE
   ↪  ...");
}
```

- Feito isso, a classe pode ainda fornecer uma série de métodos para **inserir**, **atualizar**, **excluir** e **consultar** o banco de dados. Isso pode ser feito invocando vários dos métodos presentes em **Database**:

    1. **rawQuery** ou **query**;
    2. **rawInsert** ou **insert**;
    3. **rawUpdate** ou **update**;
    4. **rawDelete** ou **delete**.

```
// Query
getNoteMapList() async {
  Database db = await
   ↪  this.database;
  var result = await db.query(
    noteTable,
    orderBy: "$colPriority ASC"
  );
  return result;
}

// Insert:
insertNote(Note note) async {
  Database db = await
   ↪  this.database;
  var result = await db.insert(
    noteTable, note.toMap());
  // var result = await
   ↪  db.RawInsert();
  return result;
}

updateNote(Note note) async {
  var db = await this.database;
```

```
    var result = await db.update(
      noteTable,
      note.toMap(),
      where: "$colId = ?",
      whereArgs: [note.id]);

    //var result = await
     ↪  db.rawUpdate();
    return result;
}

deleteNote(int id) async {
    var db = await this.database;
    int result =
    await db.rawDelete("DELETE FROM
     ↪  $noteTable WHERE $colId =
     ↪  $id");

    //int result = await db.delete();
    return result;
}

getCount() async {
```

```
  Database db = await
   ↪  this.database;

  List<Map<String, dynamic>> x =
   ↪  await
       db.rawQuery("SELECT COUNT
        ↪  (*) FROM $noteTable");

  int result =
   ↪  Sqflite.firstIntValue(x);
  return result;
}
```