Apostila - Programação de Aplicativos Mobile

Tags	Desenvolvimento Mobile Flutter
Created	@April 24, 2023 7:59 PM
Uptaded	@August 8, 2023 6:09 AM

Dart

Sintaxe básica do Dart

`Variáveis e Tipos de Dados`

```
void main() {
  int age = 30;
  double pi = 3.14;
  bool isDartFun = true;
  String name = 'John';
  List<int> numbers = [1, 2, 3];
}
```

`Operadores Aritméticos e de Atribuição`

```
void main() {
  int a = 10, b = 5;
  int sum = a + b;
  int product = a * b;
  double quotient = a / b;
  int remainder = a % b;

  a += 3; // a = a + 3
  b *= 2; // b = b * 2
}
```

`Estrutura de Controle`

```
void main() {
  if (condition) {
    // Código a ser executado se a condição for verdadeira
  } else {
    // Código a ser executado se a condição for falsa
  }

  for (int i = 0; i < 5; i++) {
    // Código a ser repetido enquanto a condição for verdadeira
  }

  while (condition) {
    // Código a ser repetido enquanto a condição for verdadeira
  }

  switch (value) {
```

```
    case 1:
      // Código a ser executado se value for igual a 1
      break;
    case 2:
      // Código a ser executado se value for igual a 2
      break;
    default:
      // Código a ser executado se nenhum caso correspondente for encontrado
  }
}
```

`Funções`

```
int add(int a, int b) {
  return a + b;
}

void printGreeting(String name) {
  print('Hello, $name!');
}

void main() {
  int result = add(5, 3);
  printGreeting('Alice');
}
```

`Classe e Objetos`

```
class Person {
  String name;
  int age;

  Person(this.name, this.age);

  void sayHello() {
    print('Hello, my name is $name and I am $age years old.');
  }
}

void main() {
  var person = Person('John', 30);
  person.sayHello();
}
```

`Null Safety`

```
void main() {
  String name = 'John';
  String? nullableName = null;

  print(name.length); // Output: 4
  print(nullableName?.length); // Safe access, no error if nullableName is null
}
```

Segue acima a sintaxe básica do dart.. Ao longo da apostila, iremos ver com detalhes cada item, com mais exemplos e com explicações.

Conceitos de Orientação a Objetos

A Programação Orientada a Objetos (POO) é um paradigma de programação que se baseia em conceitos de objetos, classes, herança, polimorfismo e encapsulamento. Vamos explicar cada um desses conceitos em detalhes:

1. **Objeto**: um objeto é uma instância de uma classe, que possui atributos e métodos. Os atributos são as características do objeto, como nome, idade, cor, etc. Os métodos são as ações que o objeto pode executar, como mover, desenhar, calcular, etc. Em POO, os objetos são a base da programação.
2. **Classe**: uma classe é um modelo que define as características e comportamentos que os objetos desse tipo terão. A classe é uma espécie de plano de fundo que é utilizado para criar os objetos. Ela contém os atributos e os métodos que serão utilizados pelos objetos.
3. **Herança**: a herança é um mecanismo que permite criar uma nova classe a partir de uma classe existente. A nova classe é chamada de subclasse ou classe derivada, e a classe existente é chamada de superclasse ou classe base. A subclasse herda todos os atributos e métodos da superclasse e pode adicionar novos atributos e métodos ou modificar os existentes.
4. **Polimorfismo**: o polimorfismo é a capacidade de um objeto de uma classe derivada de ser usado como um objeto de sua superclasse. Isso significa que um objeto pode ser tratado de diferentes maneiras, dependendo do contexto em que é usado. O polimorfismo permite que diferentes objetos respondam de maneira diferente ao mesmo método.
5. **Encapsulamento**: o encapsulamento é um conceito que se refere à proteção dos dados e métodos de uma classe. Em POO, os dados e métodos são encapsulados em uma classe para evitar que sejam acessados de maneira não autorizada. A classe define o que é público e o que é privado.
6. **Abstração**: a abstração é um conceito que se refere à representação de um objeto ou conceito no mundo real de uma maneira simplificada. Em POO, a abstração é usada para criar modelos que representam objetos ou conceitos reais, tornando-os mais fáceis de entender e manipular.

Esses conceitos de POO são amplamente utilizados em muitas linguagens de programação, incluindo Dart, e são fundamentais para o desenvolvimento de software orientado a objetos eficiente e escalável.

Exemplo na prática:

Objeto:

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

  void caminhar() {
    print("$nome está caminhando...");
  }

  void falar() {
    print("$nome está falando...");
  }

  void comer() {
    print("$nome está comendo...");
  }
}
```

```
void main() {
  Pessoa pessoa1 = Pessoa();
  pessoa1.nome = "João";
  pessoa1.idade = 30;
  pessoa1.altura = 1.75;

  pessoa1.caminhar(); // imprime "João está caminhando..."
}
```

Classe:

```
class Carro {
  String marca;
  String modelo;
  int ano;
  double velocidadeAtual = 0.0;

  void acelerar(double velocidade) {
    velocidadeAtual += velocidade;
  }

  void frear() {
    velocidadeAtual = 0.0;
  }

  void ligar() {
    print("$marca $modelo ligado.");
  }

  void desligar() {
    print("$marca $modelo desligado.");
  }
}

void main() {
  Carro carro1 = Carro();
  carro1.marca = "Ford";
  carro1.modelo = "Mustang";
  carro1.ano = 2022;

  carro1.ligar(); // imprime "Ford Mustang ligado."
}
```

Herança:

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

  void emitirSom() {
    print("O animal está emitindo som.");
  }
}

class Cachorro extends Animal {
  void emitirSom() {
    print("Au Au");
  }
}

class Gato extends Animal {
  void emitirSom() {
    print("Miau");
  }
```

```
}

void main() {
  Animal animal1 = Animal();
  Cachorro cachorro1 = Cachorro();
  Gato gato1 = Gato();

  animal1.emitirSom(); // imprime "O animal está emitindo som."
  cachorro1.emitirSom(); // imprime "Au Au"
  gato1.emitirSom(); // imprime "Miau"
}
```

Polimorfismo:

```
class Animal {
  void emitirSom() {
    print("O animal está emitindo som.");
  }
}

class Cachorro extends Animal {
  void emitirSom() {
    print("Au Au");
  }
}

class Gato extends Animal {
  void emitirSom() {
    print("Miau");
  }
}

void somAnimal(Animal animal) {
  animal.emitirSom();
}

void main() {
  Animal animal1 = Animal();
  Cachorro cachorro1 = Cachorro();
  Gato gato1 = Gato();

  somAnimal(animal1); // imprime "O animal está emitindo som."
  somAnimal(cachorro1); // imprime "Au Au"
  somAnimal(gato1); // imprime "Miau"
}
```

Encapsulamento:

```
class ContaBancaria {
  double _saldo = 0.0;

  void sacar(double valor) {
    if (_saldo >= valor) {
      _saldo -= valor;
      print("Saque de R\$$valor realizado com sucesso.");
    } else {
      print("Saldo insuficiente.");
    }
  }

  void depositar(double valor) {
    _saldo += valor;
```

O que é uma Classe:

- Classes são os moldes que usamos para construir e representar coisas do mundo real. A partir desses moldes, podemos construir objetos específicos e com características semelhantes. Falando de forma mais técnica, criar uma classe é uma forma modularizada e produtiva de escrever código. Nelas, conseguimos representar as características de objetos através das Propriedades e suas ações através dos métodos.

O que são as Propriedades de uma Classe:

- Vimos que as propriedades de uma classe são as características (informações) que desejamos registrar sobre os objetos que serão gerados por essa classe. Aprendemos que algumas informações podem ser informadas já na criação da classe, mas outras precisarão vir externamente via Construtor.

O que é Construtor:

- Aprendemos também que Construtores são como aquele “check-list” de passos a serem tomados antes de começar uma viagem: é o método que será executado assim que um objeto dessa classe for criado. A sua principal tarefa normalmente é inicializar as Propriedades, mas os Construtores também podem executar ações iniciais que a classe possa demandar.

▼ Para saber mais: outros Construtores

Você sabia que é possível personalizar seus construtores e ter mais de um construtor por classe?

E, além disso, esses construtores podem ser nomeados para que a pessoa que for usar a classe escolha qual construtor faz mais sentido para ela? Sim! Tudo isso é possível! Existem mais tipos de construtores.

A forma de fazer um construtor que aprendemos em vídeo é o jeito mais rápido, simples e direto de criar um construtor em Dart, um que apenas inicializa as **propriedades** usando **parâmetros posicionais obrigatórios**:

```
class Fruta{
  String nome;
  String cor;

  Fruta(this.nome, this.cor);
}
```

Podemos criar um novo construtor (agora um construtor nomeado) se, **por exemplo**, quisermos usar apenas Parâmetros Nomeados! Basta seguir a seguinte estrutura:

```
class Fruta{
  String nome;
  String cor;

  Fruta(this.nome, this.cor);

  Fruta.nomeados({required this.nome, required this.cor});
}
```

Note que os construtores são independentes. Ou seja, na hora que você vai criar um objeto usando essa classe, você só vai poder usar um dos construtores. Dito isso, é importante notar que **todo construtor precisa preencher as propriedades que não podem ser nulas**, daí usamos o `required` no nosso construtor `.nomeados`.

No código abaixo, vemos como inicializar objetos diferentes usando cada um dos nossos construtores:

```
main() {
  Fruta laranja = Fruta("Laranja", "Verde"); // Criando o objeto com o construtor padrão
  Fruta uva = Fruta.nomeados(nome: "Uva", cor: "Roxo"); // Criando o objeto com o construtor nomeado
}
```

Além de inicializar as propriedades, também podemos executar algumas operações durante o construtor. Para fazer isso, basta criá-lo com uma estrutura similar a da criação de uma função:

```
class Fruta{
  String nome;
  String cor;

  Fruta(this.nome, this.cor);

  Fruta.nomeados({required this.nome, required this.cor});

  Fruta.minusculas(this.nome, this.cor){
    nome = nome.toLowerCase();
    cor = cor.toLowerCase();
  }
}
```

No construtor acima nomeado `minusculas`, após receber por parâmetro as informações que preencherão as propriedades, usamos o `toLowerCase()` em cada uma delas para que o conteúdo textual se torne minúsculo. Esse é só um exemplo, e as mais diversas manipulações podem ser feitas nesse tipo de construtor.

Pronto! Agora podemos fazer operações no nosso próprio construtor, e elas acontecerão durante a inicialização do objeto (caso ele seja inicializado usando esse construtor).

O que são Métodos:

- Por fim, vimos que os Métodos são como funções dentro de uma classe e determinam os comportamentos que os objetos que serão gerados por essa classe terão.

Orientação à Objetos: Getter e Setter

Encapsulamento

Muitas das vezes nós não queremos que alguns atributos dos nossos objetos ou classes não sejam acessíveis por fora, por questões de segurança, isso se chama encapsulamento.

Assim, basta colocar um “_” à frente do atributo que quer privar, ou seja, ele só poderá ser acessado de dentro da classe ou objeto:

> ***Cuidado**! Porém no Flutter você não torna a classe privada para o arquivo que você está, você teria que criar outro arquivo para que ela ficasse privada. O privado seria variáveis de escopo.*

```
int _idade;
```

Get

Então, para poder acessar estes atributos, utilizamos os Getters. Para declarar um *Getter* basta:

```
class Pessoa {
  int _idade;
  int get idade {
    return _idade;
  }
}
```

Assim, será possível acessar o atributo _idade usando a variável idade. Porém, com o *Getter* só podemos saber qual a idade, mas não podemos mudar tal atributo.

Podemos utilizar uma *arrow function* => também (é uma sintaxe mais curta):

```
class Pessoa {
  int _idade;
  int get idade => _idade;
}
```

Set

Já os Setter é para mudar o valor. Para usar um *Setter* fazemos assim:

```
class Pessoa {
  int _idade;
  double _altura;
  int get altura {
    return _altura;
  }
  set altura(double altura){
    if(altura > 0.0 && altura < 3.0){
      _altura = altura
    }
  }
}
```

No caso acima, estou colocando um limite para alterar a altura.

Aconselha-se usar somente para proteção.

Exemplo de POO

```
class Person {
  String name;
  int age;

  Person(this.name, this.age);

  void sayHello() {
    print("Olá, meu nome é $name e eu tenho $age anos.");
  }
}
```

Nesse exemplo, a classe `Person` tem dois atributos, `name` e `age`, que armazenam o nome e a idade de uma pessoa, respectivamente. Ela tem um construtor que recebe `name` e `age` como argumentos e define os valores dos atributos.

A classe `Person` também tem um método `sayHello()` que imprime uma mensagem de saudação na tela, usando os valores dos atributos `name` e `age`.

Aqui está um exemplo de como essa classe pode ser usada:

```
void main() {
  var person = Person("João", 30);
  person.sayHello();
}
```

Nesse exemplo, um novo objeto `Person` é criado com o nome "João" e idade de 30. Em seguida, o método `sayHello()` é chamado para exibir uma mensagem na tela usando os valores dos atributos `name` e `age`. A saída do programa será: "Olá, meu nome é João e eu tenho 30 anos."

Exercícios em Dart (2)

Estrutura Condicional

IF-ELSE

O if-else é uma estrutura de controle de fluxo em programação que permite executar um bloco de código se uma condição for verdadeira e outro bloco de código se a condição for falsa. Em Dart, o if-else é formado pelas palavras-chave "if", "else" e "else if".

A sintaxe básica do if-else é a seguinte:

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

Se a condição dentro dos parênteses for verdadeira, o código dentro do bloco "if" será executado. Caso contrário, o código dentro do bloco "else" será executado. É importante notar que o bloco "else" é opcional e pode ser omitido se não houver ação a ser tomada quando a condição for falsa.

Também é possível encadear várias condições com a palavra-chave "else if", como no exemplo a seguir:

```
if (condição1) {
  // código a ser executado se a condição1 for verdadeira
} else if (condição2) {
  // código a ser executado se a condição2 for verdadeira
} else {
  // código a ser executado se nenhuma das condições anteriores for verdadeira
}
```

Neste caso, a primeira condição que for verdadeira terá o seu respectivo bloco de código executado. Se nenhuma das condições for verdadeira, o código dentro do bloco "else" será executado.

O uso do if-else é essencial para a lógica de programação e permite tomar decisões com base em condições específicas. No entanto, **é importante ter cuidado para não aninhar muitos blocos de if-else, o que pode tornar o código difícil de ler e manter**. Além disso, é importante garantir que todas as condições possíveis sejam tratadas adequadamente.

Operador Ternário

o operador ternário é semelhante ao de outras linguagens de programação, incluindo C#. A sintaxe é a seguinte:

```
condicao ? valor_se_verdadeiro : valor_se_falso
```

Assim como em C#, a `condicao` é uma expressão booleana que é avaliada como verdadeira ou falsa. Se a condição for verdadeira, o valor `valor_se_verdadeiro` será retornado, caso contrário, o valor `valor_se_falso` será retornado.

Por exemplo, suponha que você queira verificar se um número é positivo ou negativo e, em seguida, imprimir uma mensagem na tela. Você poderia escrever o seguinte código usando o operador ternário:

```
int numero = -5;
String mensagem = numero >= 0 ? "O número é positivo" : "O número é negativo";
print(mensagem);
```

Nesse exemplo, a condição `numero >= 0` é avaliada como falsa porque `numero` é igual a -5. Portanto, o valor `mensagem` será definido como "O número é negativo" e será impresso na tela.

---

switch-case

O switch case é uma estrutura de controle de fluxo em programação que permite testar uma expressão em relação a vários valores possíveis e executar uma ação diferente para cada valor correspondente. Em Dart, o switch case é formado pelas palavras-chave "switch", "case", "default" e "break".

A sintaxe básica do switch case é a seguinte:

```
switch (expressão) {
  case valor1:
```

```
    // ação para o valor1
    break;
  case valor2:
    // ação para o valor2
    break;
  ...
  case valorN:
    // ação para o valorN
    break;
  default:
    // ação para o caso em que a expressão não corresponde a nenhum dos valores anteriores
}
```

A expressão é avaliada uma vez e o resultado é comparado com cada valor no caso. Se o valor correspondente for encontrado, a ação correspondente será executada e a palavra-chave "break" encerra a estrutura switch case. Se nenhum valor correspondente for encontrado, a ação no bloco "default" será executada.

É importante notar que a expressão pode ser qualquer valor ou variável que pode ser avaliada, incluindo números, strings e booleanos. Os valores nos casos também podem ser qualquer valor ou variável compatível com a expressão.

O uso do switch case pode tornar o código mais legível e fácil de manter, especialmente quando há muitas condições a serem testadas. No entanto, é importante ter cuidado para não repetir a lógica em vários casos e garantir que todos os valores possíveis sejam tratados adequadamente.

Exemplo1:

```
void main() {
  int opcao = 2;

  switch (opcao) {
    case 1:
      print('Você escolheu a opção 1.');
      break;
    case 2:
      print('Você escolheu a opção 2.');
      break;
    case 3:
      print('Você escolheu a opção 3.');
      break;
    default:
      print('Opção inválida!');
  }
}
```

Neste exemplo, declaramos uma variável `opcao` com o valor `2`. Em seguida, usamos a estrutura `switch` para verificar o valor da variável `opcao` e executar um bloco de código correspondente ao valor.

No caso de `opcao` ser igual a `1`, o programa imprime "Você escolheu a opção 1." No caso de `opcao` ser igual a `2`, o programa imprime "Você escolheu a opção 2." No caso de `opcao` ser igual a `3`, o programa imprime "Você escolheu a opção 3." Se `opcao` não corresponder a nenhum dos casos, o programa imprime "Opção inválida!".

É importante notar que, após cada bloco de código, usamos a palavra-chave `break` para sair da estrutura `switch`. Isso é necessário para evitar que o código execute os blocos subsequentes. Caso

contrário, todos os blocos de código subsequentes serão executados até encontrar uma declaração `break` ou até o final da estrutura `switch`.

Exemplo2: sem definição de valor na variável

```
void main() {
  int opcao;

  print('Escolha uma opção (1, 2 ou 3):');
  String opcaoInput = stdin.readLineSync()!;
  opcao = int.parse(opcaoInput);

  switch (opcao) {
    case 1:
      print('Você escolheu a opção 1.');
      break;
    case 2:
      print('Você escolheu a opção 2.');
      break;
    case 3:
      print('Você escolheu a opção 3.');
      break;
    default:
      print('Opção inválida!');
  }
}
```

Neste exemplo, declaramos uma variável `opcao` sem atribuir um valor a ela. Em seguida, solicitamos ao usuário que escolha uma opção e armazenamos a entrada em uma variável chamada `opcaoInput`. Em seguida, convertemos `opcaoInput` em um inteiro e atribuímos o valor a `opcao`. Finalmente, usamos a estrutura `switch` para verificar o valor de `opcao` e executar um bloco de código correspondente ao valor.

Observe que, ao usar uma variável não inicializada em uma estrutura `switch`, é importante garantir que ela tenha um valor atribuído a ela antes de usá-la no `switch`. Se não o fizermos, o compilador Dart emitirá um erro de tempo de compilação.

Laço de repetição

while

O laço while executa um bloco de código enquanto a condição especificada for verdadeira. A sintaxe básica é a seguinte:

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

Antes de executar o bloco de código, a condição é testada. Se a condição for verdadeira, o bloco de código é executado e a condição é testada novamente. Esse processo continua até que a condição seja falsa. É importante ter cuidado para não criar um loop infinito se a condição nunca se tornar falsa.

Exemplo:

```
void main() {
  var i = 0;
  while (i < 5) {
    print(i);
    i++;
  }
}
```

Neste exemplo, o loop é executado 5 vezes, pois a condição `i < 5` é verdadeira. A cada iteração, o valor de `i` é impresso no console e incrementado.

do-while

O laço do-while é semelhante ao while, mas executa o bloco de código pelo menos uma vez, independentemente da condição. A sintaxe básica é a seguinte:

```
do {
  // código a ser executado pelo menos uma vez
} while (condição);
```

O bloco de código é executado pelo menos uma vez e, em seguida, a condição é testada. Se a condição for verdadeira, o bloco de código será executado novamente e o processo continuará até que a condição seja falsa.

Exemplo:

```
void main() {
  var i = 0;
  do {
    print(i);
    i++;
  } while (i < 5);
}
```

Neste exemplo, o loop é executado 5 vezes, assim como no exemplo anterior. A diferença é que, neste caso, a primeira iteração é executada mesmo que a condição `i < 5`
seja falsa.

for

O laço for é usado para executar um bloco de código um número fixo de vezes. A sintaxe básica é a seguinte:

```
for (inicialização; condição; incremento/decremento) {
  // código a ser executado enquanto a condição for verdadeira
}
```

A inicialização é executada uma vez antes do bloco de código. Em seguida, a condição é testada antes de cada iteração do loop. Se a condição for verdadeira, o bloco de código será executado e, em seguida, o incremento ou decremento será executado antes da próxima iteração. Esse processo continua até que a condição seja falsa.

> 💡 Além desses três tipos principais de laços de repetição, muitas linguagens de programação têm outras variações e construções, como o for each, que é usado para iterar sobre uma coleção de elementos, e o break e continue, que permitem interromper ou continuar a execução do loop em determinadas condições.

Exemplo:

```
void main() {
  for (var i = 0; i < 5; i++) {
    print(i);
  }
}
```

Neste exemplo, o loop é executado 5 vezes, pois a condição `i < 5` é verdadeira. A cada iteração, o valor de `i` é impresso no console.

ENTRADA DE DADOS NO TERMINAL

`stdin.readLineSync()` é uma função em Dart que permite ler uma linha de texto do console de entrada padrão (stdin), ou seja, ler o que o usuário digitou no terminal.

A função `readLineSync()` pausa a execução do programa até que o usuário digite uma linha de texto e pressione a tecla "Enter" (ou "Return"), e em seguida retorna essa linha de texto como uma string. É importante notar que a função retorna `null` se o usuário digitar "Ctrl + D" (ou "Ctrl + Z" no Windows), indicando o fim da entrada.

Por exemplo, o seguinte código em Dart usa `stdin.readLineSync()` para solicitar ao usuário que digite seu nome e, em seguida, exibe uma mensagem personalizada com o nome fornecido:

```
import 'dart:io';

void main() {
  print("Qual é o seu nome?");
  String nome = stdin.readLineSync();
  print("Olá, $nome! Bem-vindo ao meu programa.");
}
```

Neste exemplo, a função `print()` exibe uma mensagem solicitando o nome do usuário. Em seguida, a função `stdin.readLineSync()` é usada para ler a linha de texto digitada pelo usuário e armazená-la na variável `nome`. Finalmente, a função `print()` é usada novamente para exibir uma mensagem personalizada com o nome fornecido.

Se o usuário digitar "João", por exemplo, a saída seria:

```
Qual é o seu nome?
João
Olá, João! Bem-vindo ao meu programa.
```

💡 Observe que, se o usuário digitar várias palavras em vez de apenas uma, `stdin.readLineSync()` retornará uma única string com todas as palavras e espaços entre elas. Se você quiser ler várias linhas de entrada do usuário, pode usar um laço de repetição (por exemplo, um while) para ler cada linha individualmente até que o usuário digite uma linha vazia.

Outro exemplo:

```
import 'dart:io';

void main() {
  print('Qual é o seu nome?');
  String nome = stdin.readLineSync();

  print('Quantos anos você tem?');
  String idadeInput = stdin.readLineSync();
  int idade = int.parse(idadeInput);

  print('Olá, $nome! Você tem $idade anos.');
}
```

Este programa solicita ao usuário que digite seu nome e, em seguida, armazena a entrada em uma variável chamada `nome`. Em seguida, solicita que o usuário digite sua idade e armazena a entrada em uma variável chamada `idadeInput`. Como `stdin.readLineSync()` retorna uma `String`, precisamos convertê-la para um `int` usando o método `int.parse()`. Em seguida, o programa imprime uma mensagem de saudação personalizada com o nome e a idade do usuário.

Como fazer para ler um número ou texto no dart

Para ler um número ou texto em Dart, você pode usar a classe `stdin` do pacote `dart:io`. Essa classe permite que você leia dados do usuário a partir do console.

Para ler um número, você pode utilizar o método `int.parse()` ou `double.parse()` para converter uma string em um número inteiro ou de ponto flutuante, respectivamente. Veja um exemplo:

```
import 'dart:io';

void main() {
  stdout.write('Digite um número inteiro: ');
  int numInt = int.parse(stdin.readLineSync()!);
  print('O número digitado foi: $numInt');

  stdout.write('Digite um número de ponto flutuante: ');
  double numDouble = double.parse(stdin.readLineSync()!);
  print('O número digitado foi: $numDouble');
}
```

No exemplo acima, o método `stdout.write()` é utilizado para exibir uma mensagem no console, pedindo que o usuário digite um número inteiro e um número de ponto flutuante. Em seguida, o método `stdin.readLineSync()` é utilizado para ler a entrada do usuário como uma string, que é convertida para um número inteiro ou de ponto flutuante utilizando os métodos `int.parse()` ou `double.parse()`, respectivamente.

Para ler um texto, basta utilizar o método `stdin.readLineSync()`, que lê uma linha de texto do console e retorna uma string. Veja um exemplo:

```
import 'dart:io';

void main() {
  stdout.write('Digite seu nome: ');
  String nome = stdin.readLineSync()!;
  print('Olá, $nome!');
}
```

No exemplo acima, o método `stdout.write()` é utilizado para exibir uma mensagem no console, pedindo que o usuário digite seu nome. Em seguida, o método `stdin.readLineSync()` é utilizado para ler a entrada do usuário como uma string e armazená-la na variável `nome`. A string é então interpolada na mensagem de boas-vindas utilizando o sinal de cifrão `$`.

> o operador `!` em Dart é usado apenas para indicar que um valor não é nulo. Ele é conhecido como operador *"not-null assertion"* ou *"bang operator"*.

> No caso de números, não é necessário utilizar o operador `!`, já que um número não pode ser nulo em Dart. Portanto, você pode simplesmente usar `int.parse()` ou `double.parse()` sem o `!`.

> No caso de textos, é importante utilizar o operador `!` junto com o método `stdin.readLineSync()`, pois esse método pode retornar `null` se não houver nenhuma entrada do usuário. No entanto, se você tem certeza de que a entrada não pode ser nula, pode usar o operador de asserção de não nulo para informar ao compilador que a entrada não é nula.

CONST e FINAL no Dart

`const`:

- `const` é usado para declarar constantes que são conhecidas em tempo de compilação.
- A palavra-chave `const` é usada para criar constantes em tempo de compilação, o que significa que seus valores são definidos em tempo de compilação e não podem ser alterados em tempo de execução.
- As constantes `const` são resolvidas durante a compilação e incorporadas diretamente no código onde são usadas, o que pode resultar em otimizações de desempenho.
- As constantes `const` só podem ser declaradas usando tipos primitivos de dados, literais e objetos criados com construtores `const`.
- As constantes `const` são úteis quando você deseja garantir que o valor de uma variável não seja alterado após sua inicialização.

Exemplo de uso de `const`:

```
void main() {
  const int number = 42;
  const String message = 'Hello, world!';
  const List<int> numbers = [1, 2, 3];

  // Isso resultaria em um erro porque as constantes não podem ser modificadas.
  // number = 10;
}
```

EXPLICAÇÃO DO CÓDIGO

1. `const int number = 42;`
   Nesta linha, uma constante inteira chamada `number` é declarada e inicializada com o valor `42`. A palavra-chave `const` indica que essa variável é uma constante e seu valor é conhecido em tempo de compilação. Como o valor é definido em tempo de compilação, ele será incorporado diretamente no código, tornando-o eficiente. Note que, como é uma constante, não pode ser alterado após a atribuição.
2. `const String message = 'Hello, world!';`
   Nesta linha, uma constante de string chamada `message` é declarada e inicializada com o valor `'Hello, world!'`. Assim como no caso anterior, a palavra-chave `const` indica que essa variável é uma constante de tempo de compilação. Ela também é imutável e não pode ser alterada após a atribuição.
3. `const List<int> numbers = [1, 2, 3];`
   Nesta linha, uma constante de lista chamada `numbers` é declarada e inicializada com os valores `[1, 2, 3]`. Novamente, a palavra-chave `const` indica que essa variável é uma constante e seus valores são conhecidos em tempo de compilação. Isso significa que a lista `[1, 2, 3]` é criada em tempo de compilação e seus elementos são definidos em tempo de compilação.

`final`:

- `final` é usado para declarar constantes cujos valores são definidos em tempo de execução.
- A palavra-chave `final` é usada para criar constantes cujos valores são determinados em tempo de execução (ou seja, após a compilação).
- As constantes `final` só podem ser declaradas uma vez e não podem ser alteradas após a sua atribuição, o que significa que, uma vez atribuído um valor, ele não pode ser modificado novamente.
- As constantes `final` são úteis quando você deseja criar uma variável cujo valor é definido dinamicamente, mas não pode ser alterado posteriormente.

Exemplo de uso de `final`:

```
void main() {
  final int number = 42;
  final String name = 'John';

  // Isso resultaria em um erro porque as constantes finais não podem ser modificadas após a atribuição.
  // number = 10;
}
```

EXPLICAÇÃO DO CÓDIGO

1. `void main() { ... }`
   Aqui, temos a função `main()`, que é o ponto de entrada de qualquer programa Dart. Todo código executável começa a partir desta função.
2. `final int number = 42;`
   Nesta linha, uma variável chamada `number` é declarada e inicializada com o valor `42`. A palavra-chave `final` indica que essa variável é uma constante em tempo de execução, o que significa que seu valor é determinado em tempo de execução e não pode ser alterado após a atribuição. Uma vez atribuído um valor a uma variável `final`, ele permanecerá o mesmo durante toda a execução do programa.
3. `final String name = 'John';`
   Nesta linha, uma variável chamada `name` é declarada e inicializada com o valor `'John'`. Da mesma forma que a variável `number`, a variável `name` é uma constante em tempo de execução e não pode ser alterada após a atribuição.

OO com Dart

A linguagem de programação Dart suporta programação orientada a objetos (POO), e é possível criar classes, objetos e herança utilizando seus recursos.

Para criar uma classe em Dart, utilize a palavra-chave `class` seguida do nome da classe e de chaves, que irão conter os membros da classe. Por exemplo:

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

  Pessoa(this.nome, this.idade);

  void aniversario() {
    idade++;
  }
}
```

Este exemplo define uma classe chamada `Pessoa`, com dois membros `nome` e `idade`. O construtor é definido utilizando a sintaxe `Pessoa(this.nome, this.idade)`, que atribui os valores passados para as variáveis `nome` e `idade`. O método `aniversario()` incrementa a idade da pessoa em um ano.

Para criar um objeto da classe `Pessoa`, utilize o operador `new` seguido do nome da classe e dos parâmetros do construtor, como no exemplo a seguir:

```
Pessoa pessoa1 = new Pessoa("João", 30);
```

Este exemplo cria um objeto da classe `Pessoa` com nome "João" e idade 30.

Para herdar de uma classe em Dart, utilize a palavra-chave `extends`. Por exemplo:

```
class Aluno extends Pessoa {
  String curso;

  Aluno(String nome, int idade, this.curso) : super(nome, idade);
}
```

Este exemplo define uma classe `Aluno` que herda de `Pessoa`. Além das variáveis `nome` e `idade`, a classe `Aluno` possui a variável `curso`. O construtor da classe `Aluno` chama o construtor da classe `Pessoa` utilizando a sintaxe `: super(nome, idade)`.

Estas são apenas algumas das características de POO em Dart. Há muito mais recursos e funcionalidades para explorar.

Outras funcionalidades de OO

Algumas outras funcionalidades da programação orientada a objetos em Dart incluem:

1. Modificadores de acesso: Dart possui modificadores de acesso `public`, `private` e `protected` para controlar o acesso aos membros de uma classe. Os membros `public` são acessíveis a partir de qualquer lugar do código, enquanto os membros `private` são acessíveis somente dentro da classe e os membros `protected` são acessíveis dentro da classe e em suas subclasses.
2. Métodos getters e setters: em Dart, é possível definir métodos getters e setters para acessar e modificar variáveis privadas de uma classe. Por exemplo:

```
class Pessoa {
  String _nome;

  String get nome => _nome;

  set nome(String nome) {
    _nome = nome;
  }
}
```

Este exemplo define uma classe `Pessoa` com uma variável privada `_nome`. Os métodos getter e setter são definidos utilizando a sintaxe `get nome => _nome` e `set nome(String nome) { _nome = nome; }`, respectivamente. Desta forma, é possível acessar e modificar a variável `_nome` utilizando a sintaxe `pessoa.nome` e `pessoa.nome = "João"`.

3. Classes abstratas: em Dart, é possível definir classes abstratas que não podem ser instanciadas diretamente, mas servem como modelos para outras classes. As classes abstratas são definidas utilizando a palavra-chave `abstract` antes do nome da classe. Por exemplo:

```
abstract class Animal {
  void fazerBarulho();
}

class Cachorro extends Animal {
  void fazerBarulho() {
    print("Au au!");
  }
}

class Gato extends Animal {
  void fazerBarulho() {
```

```
    print("Miau!");
  }
}
```

Este exemplo define uma classe abstrata `Animal` com um método `fazerBarulho()`. As classes `Cachorro` e `Gato` herdam de `Animal` e implementam o método `fazerBarulho()`, cada uma com sua própria implementação.

4. Mixins: em Dart, é possível definir mixins, que são classes que contêm métodos que podem ser reutilizados em outras classes. Para definir um mixin, utiliza-se a palavra-chave `mixin` antes do nome da classe. Por exemplo:

```
mixin Voador {
  void voar() {
    print("Estou voando!");
  }
}

class Pato extends Animal with Voador {
  void fazerBarulho() {
    print("Quack!");
  }
}

class Aviao with Voador {
  void voar() {
    print("Estou voando como um avião!");
  }
}
```

Este exemplo define um mixin `Voador` com um método `voar()`. A classe `Pato` herda de `Animal` e utiliza o mixin `Voador` para implementar o método `voar()`. A classe `Aviao` utiliza o mixin `Voador` para implementar seu próprio método `voar()`. Note que é possível sobrescrever métodos do mixin em uma classe que o utiliza.

Exemplo

```
void main() {
  // Chamando a função com parâmetros e recebendo o resultado
  int resultado = soma(10, 20);
  print(resultado); // 30

  // Chamando a função sem parâmetros e sem retorno
  imprimeMensagem();
}

// Função que recebe dois parâmetros inteiros e retorna a soma
int soma(int a, int b) {
  return a + b;
}

// Função que não recebe parâmetros nem retorna valor
void imprimeMensagem() {
  print("Olá, mundo!");
}
```

Nesse exemplo, estamos criando duas funções. A primeira função é chamada `soma` e recebe dois parâmetros inteiros `a` e `b`. Dentro da função, estamos retornando a soma desses dois valores. Essa

função tem um tipo de retorno definido como `int`, indicando que ela vai retornar um valor inteiro.

A segunda função é chamada `imprimeMensagem` e não recebe parâmetros nem retorna valor. Dentro da função, estamos imprimindo uma mensagem na tela.

Na função `main`, estamos chamando as duas funções diferentes. Na primeira chamada, estamos passando dois valores inteiros como parâmetros para a função `soma` e armazenando o resultado em uma variável `resultado`. Em seguida, estamos imprimindo o valor dessa variável na tela.

Na segunda chamada, estamos simplesmente chamando a função `imprimeMensagem` sem passar nenhum parâmetro. Como essa função não tem valor de retorno, não precisamos armazenar o resultado em nenhuma variável.

Funções

Uma função em Dart é um bloco de código que realiza uma tarefa específica e pode ser reutilizado em diferentes partes do programa. As funções têm um nome, uma lista de parâmetros (opcionais) e um corpo, que contém as instruções que são executadas quando a função é chamada.

Para definir uma função em Dart, usamos a palavra-chave `void` seguida pelo nome da função e seus parâmetros (se houver). Exemplo:

```
void imprimirMensagem(String mensagem) {
  print(mensagem);
}
```

Nesse exemplo, estamos definindo uma função chamada `imprimirMensagem` que recebe um parâmetro `mensagem` do tipo `String` e imprime a mensagem no console.

Para chamar uma função em Dart, usamos seu nome seguido pelos parâmetros (se houver). Exemplo:

```
imprimirMensagem("Olá, mundo!");
```

Esse código chama a função `imprimirMensagem` e passa a mensagem "Olá, mundo!" como argumento.

As funções em Dart também podem ter um tipo de retorno, que indica o tipo de valor que a função retorna. Por exemplo:

```
int soma(int a, int b) {
  return a + b;
}
```

Nesse exemplo, estamos definindo uma função chamada `soma` que recebe dois parâmetros do tipo `int` e retorna a soma desses valores como um inteiro. Para indicar o tipo de retorno, usamos o nome da função seguido por dois pontos e o tipo. Depois, usamos a palavra-chave `return` para retornar o valor calculado.

Podemos chamar essa função da seguinte forma:

```
int resultado = soma(2, 3);
print(resultado); // imprime 5
```

Esse código chama a função `soma` com os argumentos 2 e 3, recebe o resultado da soma e o armazena na variável `resultado`, que é impressa no console.

Listas

`List` é uma estrutura de dados em Dart que permite armazenar e manipular coleções de elementos em sequência. As listas podem conter elementos de qualquer tipo, como números, strings, objetos, funções, etc. O tamanho de uma lista pode ser alterado dinamicamente, ou seja, podemos adicionar, remover ou substituir elementos em qualquer posição da lista.

Para criar uma lista em Dart, podemos usar a sintaxe de lista literal, que consiste em colocar os elementos entre colchetes `[]`, separados por vírgulas. Exemplo:

```
var minhaLista = [1, 2, 3, 4, 5];
```

Também é possível criar uma lista vazia usando a classe `List` e especificando o tamanho da lista no construtor. Exemplo:

```
var listaVazia = List<int>(5);
```

Nesse caso, estamos criando uma lista vazia com capacidade para 5 elementos do tipo inteiro.

Algumas operações comuns em listas incluem:

- Acessar elementos: podemos acessar um elemento da lista usando o índice correspondente, que começa em 0. Exemplo: `minhaLista[0]` retorna o primeiro elemento da lista, que é o valor 1.
- Adicionar elementos: podemos adicionar um elemento ao final da lista usando o método `add()`. Exemplo: `minhaLista.add(6)` adiciona o valor 6 ao final da lista.
- Remover elementos: podemos remover um elemento da lista usando o método `remove()`. Exemplo: `minhaLista.remove(3)` remove o valor 3 da lista, se ele estiver presente.
- Substituir elementos: podemos substituir um elemento da lista usando o índice correspondente e o operador de atribuição `=`. Exemplo: `minhaLista[2] = 7` substitui o terceiro elemento da lista pelo valor 7.
- Iterar sobre elementos: podemos percorrer todos os elementos da lista usando um laço `for`. Exemplo:

```
for (var elemento in minhaLista) {
  print(elemento);
}
```

Esse código imprime todos os elementos da lista `minhaLista` em sequência.

Existem muitas outras operações que podem ser feitas com listas em Dart, como ordenação, filtro, mapeamento, concatenação, etc. Para aprender mais sobre listas e outras estruturas de dados em Dart, consulte a documentação oficial em **https://dart.dev/guides/libraries.**

Exemplo

```
void main() {
  // Criando uma lista de inteiros
  List<int> numeros = [1, 2, 3, 4, 5];

  // Adicionando um elemento ao final da lista
  numeros.add(6);

  // Imprimindo a lista
  print(numeros); // [1, 2, 3, 4, 5, 6]

  // Acessando um elemento pelo índice
  print(numeros[3]); // 4

  // Alterando um elemento pelo índice
  numeros[2] = 10;

  // Imprimindo a lista novamente
  print(numeros); // [1, 2, 10, 4, 5, 6]

  // Percorrendo a lista com um loop for
  for (int i = 0; i < numeros.length; i++) {
    print(numeros[i]);
  }

  // Percorrendo a lista com um loop for-in
  for (int numero in numeros) {
    print(numero);
  }
}
```

Nesse exemplo, estamos criando uma lista de inteiros chamada `numeros` com os valores de 1 a 6. Depois, adicionamos um novo número ao final da lista usando o método `add`.

Em seguida, imprimimos a lista inteira usando `print` e acessamos um elemento específico pelo seu índice (no caso, o quarto elemento, que tem índice 3).

Depois, alteramos o terceiro elemento da lista para o valor 10 e imprimimos a lista novamente.

Em seguida, usamos um loop for e um loop for-in para percorrer a lista e imprimir cada elemento.

Tipos de dados

As listas e mapas são tipos de dados em Dart que permitem armazenar e manipular conjuntos de valores.

List

Uma lista é uma coleção ordenada de objetos, que pode conter objetos de diferentes tipos. Para criar uma lista, podemos usar a classe `List<T>`, onde `T`
é o tipo dos objetos que queremos armazenar na lista. Aqui estão alguns exemplos:

```
// Criar uma lista vazia de strings
List<String> listaVazia = [];

// Criar uma lista com valores iniciais de inteiros
List<int> listaNumeros = [1, 2, 3, 4, 5];

// Adicionar um elemento ao final da lista
listaNumeros.add(6);

// Remover um elemento da lista
listaNumeros.remove(3);

// Acessar um elemento em uma posição específica da lista
print(listaNumeros[0]); // 1

// Iterar sobre os elementos da lista usando um for loop
for (var numero in listaNumeros) {
  print(numero);
}

// Iterar sobre os elementos da lista usando forEach()
listaNumeros.forEach((numero) {
  print(numero);
});
```

Mapas

Um mapa é uma coleção de pares de chave-valor, onde cada valor é identificado por uma chave exclusiva. Para criar um mapa, podemos usar a classe `Map<K, V>`
, onde `K` é o tipo das chaves e `V` é o tipo dos valores que queremos armazenar. Aqui estão alguns exemplos:

```
// Criar um mapa vazio de strings para inteiros
Map<String, int> mapaVazio = {};

// Criar um mapa com valores iniciais de strings para strings
Map<String, String> mapaNomes = {
  'Alice': 'Programadora',
  'Bob': 'Designer',
  'Carol': 'Gerente de projeto'
};

// Adicionar um novo par chave-valor ao mapa
mapaNomes['Dave'] = 'Analista de dados';

// Remover um par chave-valor do mapa
mapaNomes.remove('Bob');

// Acessar um valor usando uma chave específica
print(mapaNomes['Alice']); // 'Programadora'

// Iterar sobre as chaves e valores do mapa usando forEach()
mapaNomes.forEach((chave, valor) {
  print('$chave é $valor');
});
```

Em resumo, as listas são usadas para armazenar uma coleção ordenada de objetos, enquanto os mapas são usados para armazenar uma coleção de pares chave-valor. Ambos os tipos de dados são úteis para organizar e manipular conjuntos de dados em Dart.

Exception e Null Safety

Em Dart, lidar com exceções (exceptions) e utilizar o recurso de null safety são aspectos importantes para tornar o código mais robusto e seguro. Vamos abordar cada um deles:

`Lidando com Exceptions:`

As exceções são usadas para lidar com situações excepcionais que podem ocorrer durante a execução do programa. Dart oferece a possibilidade de capturar e tratar exceções usando blocos `try`, `catch`, `on`, `finally`.

- Bloco `try`: O código suscetível a lançar exceções é colocado dentro do bloco `try`.
- Bloco `catch`: Se ocorrer uma exceção dentro do bloco `try`, o código dentro do bloco `catch` é executado, permitindo que você trate a exceção de forma adequada.
- Bloco `on`: Você pode especificar o tipo de exceção que deseja capturar usando a cláusula `on`.
- Bloco `finally`: O código dentro do bloco `finally` é sempre executado, independentemente de ocorrer ou não uma exceção. É usado para limpar recursos, como fechar arquivos ou conexões de banco de dados.

Exemplo:

```
void main() {
  try {
    int result = 10 ~/ 0; // Divisão por zero lança uma exceção
    print('Resultado: $result');
  } catch (e) {
    print('Exceção capturada: $e');
  } finally {
    print('Finalizado.');
  }
}
```

`Null Safety:`

O Null Safety é um recurso introduzido no Dart 2.12 para evitar erros frequentes relacionados a valores nulos (null) durante a execução do programa. Com o Null Safety, você deve explicitamente informar quando uma variável pode conter um valor nulo usando o operador `?`.

- Variáveis não nulas: Variáveis que não podem conter valores nulos devem ser declaradas sem o operador `?`.
- Variáveis possivelmente nulas: Variáveis que podem conter valores nulos devem ser declaradas com o operador `?`.

Exemplo:

```
void main() {
  String name = 'John'; // Variável não nula
  String? nullableName = null; // Variável possivelmente nula

  print(name.length); // Nenhum problema, name não é nulo.

  // Se nullableName for nulo, o código abaixo resultará em um erro em tempo de compilação.
  print(nullableName.length);
}
```

Flutter

A estrutura do Flutter é baseada em alguns conceitos principais:

1. Widgets:
   O Flutter utiliza widgets como a unidade básica de construção da interface do usuário. Widgets são componentes visuais que descrevem o que deve ser renderizado na tela. Existem dois tipos principais de widgets: Stateless (sem estado) e Stateful (com estado). Os widgets Stateful podem armazenar dados e atualizar a interface do usuário quando esses dados mudam.
2. MaterialApp:
   MaterialApp é um widget que define a estrutura básica de um aplicativo Flutter. Ele oferece suporte a navegação, temas e muito mais. É um wrapper para vários outros widgets fornecidos pelo Flutter, como Scaffold, Navigator, AppBar etc.
3. Scaffold:
   Scaffold é um widget que implementa a estrutura básica de um aplicativo Material Design. Ele fornece uma barra de aplicativos (AppBar), um botão de menu de navegação, uma área de conteúdo principal (body), uma gaveta de navegação (drawer), entre outros recursos.
4. Hot Reload:
   Uma das características mais poderosas do Flutter é o Hot Reload, que permite que os desenvolvedores visualizem imediatamente as mudanças feitas no código-fonte em tempo de execução, sem perder o estado atual do aplicativo. Isso acelera significativamente o processo de desenvolvimento.
5. Packages:
   O Flutter tem uma vasta comunidade de desenvolvedores que criam e compartilham pacotes (packages) com funcionalidades prontas para uso. Esses pacotes podem ser facilmente integrados ao seu aplicativo para adicionar recursos adicionais.
6. Pubspec.yaml:
   O arquivo "pubspec.yaml" é um arquivo de configuração onde você especifica as dependências (packages) do seu projeto, além de outras configurações, como nome do aplicativo, descrição, versão, entre outros.
7. Estrutura de diretórios típica:
   Um projeto Flutter geralmente tem uma estrutura de diretórios organizada, comumente incluindo pastas como "lib" (para o código-fonte do aplicativo), "assets" (para recursos como imagens e fontes) e "test" (para testes automatizados).
8. Gerenciamento de estado:
   O Flutter oferece várias opções para o gerenciamento de estado, como o próprio gerenciamento de estado interno do widget, o uso de pacotes de gerenciamento de estado (por exemplo, Provider, Bloc, MobX), ou até mesmo o uso do gerenciamento de estado fornecido pelo Firebase.

Esses são apenas alguns dos conceitos básicos da estrutura do Flutter. Com o Flutter, você pode criar aplicativos bonitos, rápidos e fluidos em várias plataformas, proporcionando uma experiência de usuário consistente. É uma ferramenta poderosa e popular para desenvolvedores de aplicativos móveis e também é amplamente adotada pela comunidade de desenvolvedores.

O Flutter possui uma grande variedade de widgets que podem ser usados para construir interfaces de usuário complexas e interativas. Vou explicar alguns dos principais widgets do Flutter e dar exemplos

de como eles podem ser utilizados:

Container:
O widget `Container` é um widget que permite personalizar o layout, a cor, a borda e outras propriedades de seus filhos.

Exemplo:

```
Container(
  width: 100,
  height: 100,
  color: Colors.blue,
  child: Center(
    child: Text('Hello Flutter', style: TextStyle(color: Colors.white)),
  ),
)
```

Column:
O widget `Column` é um widget que organiza seus filhos em uma coluna vertical.

Exemplo:

```
Column(
  children: [
    Text('Item 1'),
    Text('Item 2'),
    Text('Item 3'),
  ],
)
```

Row:
O widget `Row` é um widget que organiza seus filhos em uma linha horizontal.

Exemplo:

```
Row(
  children: [
    Icon(Icons.email),
    Text('example@email.com'),
  ],
)
```

ListView:
O widget `ListView` é um widget que cria uma lista rolável de seus filhos. Existem várias variantes do `ListView`, como `ListView.builder` e `ListView.separated`, que são úteis para lidar com grandes listas de itens.

Exemplo:

```
ListView(
  children: [
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    ListTile(title: Text('Item 3')),
  ],
)
```

AppBar:
O widget `AppBar` é um widget que representa a barra superior do aplicativo. É comumente usado junto com o `Scaffold` para criar uma estrutura típica de aplicativos.

Exemplo:

```
Scaffold(
  appBar: AppBar(title: Text('Meu App')),
  body: Center(child: Text('Conteúdo do aplicativo')),
)
```

TextField:
O widget `TextField` é um widget que permite ao usuário inserir texto.

Exemplo:

```
TextField(
  decoration: InputDecoration(
    labelText: 'Digite seu nome',
  ),
)
```

O `ElevatedButton` é um botão de superfície elevada que representa uma ação a ser executada quando é pressionado.

Aqui está o exemplo atualizado usando o `ElevatedButton`:

```
ElevatedButton(
  onPressed: () {
    // Ação a ser executada ao pressionar o botão
    print('Botão pressionado');
  },
  child: Text('Pressione-me'),
)
```

Image:
O widget `Image` é usado para exibir imagens no Flutter. Ele pode ser usado para exibir imagens locais ou remotas.

Exemplo:

```
Image.network('https://example.com/image.png')
```

Esses são apenas alguns dos principais widgets disponíveis no Flutter. Há muitos outros widgets disponíveis que podem ser usados para criar interfaces de usuário ricas e complexas. A documentação oficial do Flutter é uma excelente fonte para explorar e aprender sobre todos os widgets disponíveis: **https://flutter.dev/docs/development/ui/widgets**

TextField:
O `TextField` é um widget usado para entrada de texto. Ele permite que os usuários insiram dados, como texto, números ou senhas.

Exemplo:

```
TextField(
  decoration: InputDecoration(
    labelText: 'Nome',
    hintText: 'Digite seu nome',
  ),
)
```

Explicação TextFiel:

- `decoration`: É usado para personalizar a aparência do campo de texto.
- `labelText`: Define um rótulo para o campo de texto.
- `hintText`: Exibe um texto de sugestão quando o campo está vazio.

DropdownButton:
O `DropdownButton` cria um menu suspenso onde os usuários podem selecionar uma opção de uma lista.

Exemplo:

```
DropdownButton<String>(
  value: selectedOption,
  onChanged: (newValue) {
    setState(() {
      selectedOption = newValue;
    });
  },
  items: <String>['Opção 1', 'Opção 2', 'Opção 3'].map((String value) {
    return DropdownMenuItem<String>(
      value: value,
      child: Text(value),
    );
  }).toList(),
)
```

RadioListTile:
O `RadioListTile` cria uma lista de opções de rádio, onde o usuário pode selecionar uma opção única.

Exemplo:

```
RadioListTile<int>(
  value: 1,
  groupValue: selectedValue,
  onChanged: (value) {
    setState(() {
      selectedValue = value;
    });
  },
  title: Text('Opção 1'),
)
```

CheckboxListTile:
O `CheckboxListTile` cria uma lista de caixas de seleção, onde o usuário pode selecionar várias opções.

Exemplo:

```
CheckboxListTile(
  value: isChecked,
  onChanged: (value) {
    setState(() {
      isChecked = value;
    });
  },
  title: Text('Aceito os termos e condições'),
)
```

Para mais detalhes, acesse: https://medium.com/flutter-comunidade-br/flutter-descomplica-lista-de-checkbox-1e0dca525cb

SwitchListTile:
O `SwitchListTile` cria uma lista de interruptores que podem ser alternados entre os estados "ligado" e "desligado".

Exemplo:

```
SwitchListTile(
  value: isSwitched,
  onChanged: (value) {
    setState(() { // é usado para atualizar o estado do widget quando o usuário faz uma ação.
      isSwitched = value;
    });
  },
  title: Text('Ativar notificações'),
)
```

StatelessWidget:

- Um `StatelessWidget` é um widget que não possui estado interno mutável. Isso significa que seus atributos (propriedades) são fixos durante toda a vida útil do widget.
- Quando um `StatelessWidget` é construído, ele gera uma representação visual (interface do usuário) com base em seus atributos, mas essa representação não pode ser alterada após a construção.
- É usado para widgets que não precisam reagir a mudanças de estado e têm uma representação visual estática.
- Um exemplo comum de `StatelessWidget` é um botão que não muda visualmente após a criação.

Exemplo de `StatelessWidget`:

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

class MeuBotao extends StatelessWidget {
  final String texto;
  final VoidCallback onPressed;

  MeuBotao({required this.texto, required this.onPressed});

  @override
  Widget build(BuildContext context) {
```

```
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(texto),
    );
  }
}
```

StatefulWidget:

- Um `StatefulWidget` é um widget que pode ter estado interno mutável. Isso permite que o widget reaja a mudanças de estado e atualize sua representação visual.
- A classe `StatefulWidget` possui uma classe associada chamada `State`, que armazena o estado mutável do widget.
- O método `build()` do `StatefulWidget` é chamado sempre que o estado do widget é atualizado, permitindo que a interface do usuário seja reconstruída.
- É usado para widgets que precisam reagir a eventos, alterações de dados ou interações do usuário.

Exemplo de `StatefulWidget`:

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

class Contador extends StatefulWidget {
  @override
  _ContadorState createState() => _ContadorState();
}

class _ContadorState extends State<Contador> {
  int _contador = 0;

  void _incrementarContador() {
    setState(() {
      _contador++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Contagem: $_contador'),
        ElevatedButton(
          onPressed: _incrementarContador,
          child: Text('Incrementar'),
        ),
      ],
    );
  }
}
```

Em resumo:

- Use `StatelessWidget` quando você deseja criar um widget cuja representação visual é estática e não muda ao longo do tempo.
- Use `StatefulWidget` quando você precisa criar um widget que reage a eventos, atualizações de estado ou interações do usuário e pode ter uma representação visual dinâmica.

💡 É importante escolher o tipo certo de widget com base na natureza das suas necessidades e na interatividade da interface do usuário que você está construindo. Muitas vezes, você encontrará situações em que precisa usar ambos os tipos de widgets em conjunto para criar interfaces de usuário completas e interativas.

REPOSITÓRIOS FLUTTER E DART NO GITHUB

https://github.com/gui-alonso/flutter_level1

https://github.com/gui-alonso/revisao-dart

Elaborado por: Professor Guilherme