Lista de Conteúdos - Terceira Etapa/2(Linguagem Dart)

1. Geral

1.1 `late`

- **Definição do Modificador `late`:**
  - O modificador `late` em Dart indica que uma variável não nula será inicializada posteriormente, mas antes de seu primeiro uso.
- **Exemplo de Uso:**

```
class Pessoa {
  late String nome;

  void inicializarNome(String nome) {
    this.nome = nome;
  }
}

void main() {
  var pessoa = Pessoa();
  pessoa.inicializarNome('João');
  print(pessoa.nome); // Saída: João
}
```

1.2 `final`

- **Definição do Modificador `final`:**
  - `final` impede que uma variável seja reatribuída após sua primeira inicialização, garantindo que o valor seja constante em tempo de execução.
- **Exemplo de Uso:**

```
void main() {
  final int idade = 25;
  // idade = 30; // Isso causará um erro de compilação
  print(idade); // Saída: 25
}
```

2. Orientação a Objeto

2.1 Classes

- **Definição de uma Classe:**
  - Uma classe é um molde para criar objetos, contendo propriedades e métodos que definem o comportamento desses objetos.
- **Estrutura Básica de uma Classe:**

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

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

void main() {
  var pessoa = Pessoa('Ana', 30);
  print('Nome: ${pessoa.nome}, Idade: ${pessoa.idade}'); // Saída: Nome: Ana, Idade: 30
}
```

2.2 Construtor

- **Definição e Uso do Construtor:**
  - Um construtor é usado para inicializar os valores das propriedades de um objeto quando ele é criado.
- **Tipos de Construtores:**
  - **Construtor Padrão:**

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

  // Construtor padrão
  Pessoa(this.nome, this.idade);
}

void main() {
  var pessoa1 = Pessoa('João', 25);
  print('Nome: ${pessoa1.nome}, Idade: ${pessoa1.idade}'); // Saída: Nome: João, Idade: 25
}
```

  - **Construtor Nomeado:**

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

  // Construtor padrão
  Pessoa(this.nome, this.idade);

  // Construtor nomeado
  Pessoa.idadeZero(this.nome) : idade = 0;
}

void main() {
  var pessoa2 = Pessoa.idadeZero('Maria');
  print('Nome: ${pessoa2.nome}, Idade: ${pessoa2.idade}'); // Saída: Nome: Maria, Idade: 0
}
```

2.3 Métodos

- **Definição e Uso de Métodos:**
  - Métodos são funções definidas dentro de uma classe, usadas para operar sobre os dados dessa classe.
- **Estrutura Básica de um Método:**

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

  Pessoa(this.nome, this.idade);

  // Método para exibir uma saudação
  void saudacao() {
    print('Olá, meu nome é $nome e eu tenho $idade anos.');
  }
}

void main() {
  var pessoa = Pessoa('Ana', 30);
  pessoa.saudacao(); // Saída: Olá, meu nome é Ana e eu tenho 30 anos.
}
```

- **Métodos com Retorno:**

```
class Calculadora {
  // Método que retorna a soma de dois números
  int soma(int a, int b) {
    return a + b;
  }
}

void main() {
  var calc = Calculadora();
  int resultado = calc.soma(10, 5);
```

```
  print('Resultado: $resultado'); // Saída: Resultado: 15
}
```

- **Métodos Estáticos:**

```
class Util {
  // Método estático para verificar se um número é par
  static bool ePar(int numero) {
    return numero % 2 == 0;
  }
}

void main() {
  bool resultado = Util.ePar(4);
  print('4 é par? $resultado'); // Saída: 4 é par? true
}
```

3. Pilares da Programação Orientada a Objetos

3.1 Abstração

- **Definição de Abstração:**
  - Abstração permite focar nas características essenciais de um objeto, ignorando detalhes de implementação.
- **Exemplos Práticos:**
  - **Classe Abstrata e Método Abstrato:**

```
abstract class Animal {
  // Método abstrato
  void fazerSom();

  // Método concreto
  void dormir() {
    print('Dormindo...');
  }
}

class Cachorro extends Animal {
  @override
  void fazerSom() {
    print('Latido');
  }
}

void main() {
  var dog = Cachorro();
  dog.fazerSom(); // Saída: Latido
  dog.dormir();   // Saída: Dormindo...
}
```

3.2 Encapsulamento

- **Definição de Encapsulamento:**
  - Encapsulamento protege os dados internos de um objeto, permitindo acesso controlado através de métodos.
- **Modificadores de Acesso:**
  - Dart utiliza _ para tornar atributos e métodos privados.
- **Exemplos Práticos:**
  - **Atributos Privados e Métodos Públicos:**

```
class ContaBancaria {
  double _saldo = 0;

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

```
  double get saldo => _saldo;
}

void main() {
  var conta = ContaBancaria();
  conta.depositar(100);
  print('Saldo: ${conta.saldo}'); // Saída: Saldo: 100.0
}
```

- **Getters e Setters:**

```
class Produto {
  String _nome;
  double _preco;

  Produto(this._nome, this._preco);

  String get nome => _nome;
  set nome(String value) => _nome = value;

  double get preco => _preco;
  set preco(double value) {
    if (value > 0) _preco = value;
  }
}

void main() {
  var produto = Produto('Celular', 800);
  print(produto.nome);  // Saída: Celular
  produto.preco = 900;
  print(produto.preco); // Saída: 900
}
```

3.3 Herança

- **Definição de Herança:**
  - Herança permite que uma classe herde atributos e métodos de outra classe, criando uma hierarquia.
- **Herança Simples:**

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

  Animal(this.nome, this.idade);

  void showOutput() {
    print('Nome: $nome, Idade: $idade');
  }
}

class Dog extends Animal {
  Dog(String nome, int idade) : super(nome, idade);

  @override
  void showOutput() {
    super.showOutput();
    print('Dog specific behavior');
  }
}

void main() {
  var dog = Dog('Rex', 5);
  dog.showOutput();
  // Saída:
  // Nome: Rex, Idade: 5
  // Dog specific behavior
}
```

- **Herança Múltipla (Mixin):**

```
mixin Cor {
  String cor = 'Branco';

  void mostrarCor() {
    print('Cor: $cor');
```

```
  }
}

class Animal {
  String nome;

  Animal(this.nome);
}

class Gato extends Animal with Cor {
  Gato(String nome) : super(nome);
}

void main() {
  var gato = Gato('Felix');
  gato.mostrarCor(); // Saída: Cor: Branco
}
```

3.4 Polimorfismo

- **Definição de Polimorfismo:**
  - Polimorfismo permite que métodos se comportem de maneira diferente com base no objeto que os invoca.
- **Sobrescrita de Métodos:**

```
class Veiculo {
  void mover() {
    print('Veículo se movendo');
  }
}

class Carro extends Veiculo {
  @override
  void mover() {
    print('Carro se movendo');
  }
}

void main() {
  Veiculo veiculo = Carro();
  veiculo.mover(); // Saída: Carro se movendo
}
```

- **Sobrecarga de Mét

odos:**

- **Simulação de Sobrecarga com Parâmetros Opcionais:**

```
class Carro {
  void acelerar([int velocidade = 0]) {
    if (velocidade > 0) {
      print('Carro acelerando a $velocidade km/h');
    } else {
      print('Carro acelerando');
    }
  }
}

void main() {
  var carro = Carro();
  carro.acelerar(100); // Saída: Carro acelerando a 100 km/h
  carro.acelerar();    // Saída: Carro acelerando
}
```

4. Conceitos Avançados

4.1 Classe Abstrata e Método Abstrato

- **Definição e Uso de Classes Abstratas:**
  - Classes abstratas não podem ser instanciadas diretamente e servem como base para outras classes.

- **Definição e Uso de Métodos Abstratos:**
  - Métodos abstratos não têm corpo e devem ser implementados por subclasses.
- **Exemplo Prático:**

```
abstract class Forma {
  double calcularArea();
}

class Circulo extends Forma {
  double raio;

  Circulo(this.raio);

  @override
  double calcularArea() {
    return 3.14 * raio * raio;
  }
}

void main() {
  var circulo = Circulo(5);
  print('Área do círculo: ${circulo.calcularArea()}'); // Saída: Área do círculo: 78.5
}
```

4.2 Mixin

- **Definição de Mixin:**
  - Mixins permitem a reutilização de código em várias classes sem usar herança direta.
- **Exemplo Prático:**

```
mixin Caminhar {
  void andar() {
    print('Andando...');
  }
}

class Animal {}

class Pato extends Animal with Caminhar {}

void main() {
  var pato = Pato();
  pato.andar(); // Saída: Andando...
}
```
