



# PROGRAMAÇÃO II

## AULA 2

Prof. Elton Masaharu Sato

## CONVERSAINICIAL

Como vimos anteriormente, o Flutter é um framework de desenvolvimento que utiliza uma linguagem de programação chamada Dart. Nesta etapa, vamos aprender o básico de programação com Dart. Os leitores que já conhecem uma linguagem de programação orientada a objetos vão identificar similaridades na forma como o código é escrito, já que o Dart também é orientado ao objeto.

Figura 1 – Dart



Fonte: Dart, 2022.

## TEMA 1 – DART: O QUE É?





Veremos como funciona o Dart, linguagem de programação do Flutter, e com quais objetivos ele foi desenvolvido.

### 1.1 OBJETIVOS

Assim como o Flutter foi desenvolvido buscando solucionar problemas específicos, o Dart foi projetado e planejado com alguns objetivos em mente. Um dos pontos principais do Dart é a sua otimização para UI (User Interface), ou "interface do usuário". Vejamos algumas das características que o ajudam a atingir esse objetivo:

- **Funções assíncronas e espera:** permite que operações lentas sem tempo determinado possam ser marcadas, para outras funções, considerando a sua completude. Vamos imaginar um programa que mostra para um cliente uma lista de compras. Para que essa funcionalidade rode corretamente, são necessárias duas etapas: a primeira é buscar a lista de compras do cliente na base de dados, e a segunda é imprimir a lista na tela. Mas como a busca da lista online é lenta, se a segunda parte do código não esperar pela lista, ela escreverá na tela tudo o que sabe sobre a lista: nada.
- **Funções para trabalhar com coleções:** Dart tem uma variedade de funções prontas para trabalhar com coleções ou agrupamentos de dados. Isso facilita a programação de códigos que operam com múltiplos dados personalizados ao usuário. O Dart tem uma operação chamada de "collection if", que cria coleções baseados em uma condição "if". Assim, podemos criar uma coleção de dados que só existem se uma condição estiver ativa. Usando a lista de compras do cliente como exemplo, podemos criar uma lista de produtos de compra que só existem na lista quando os produtos estão em promoção.
- **Linguagem de alto nível:** quanto maior o nível da linguagem, mais próxima da linguagem humana. Em uma linguagem como Assembly, temos um exemplo de linguagem de baixíssimo nível, com uma versão com letras romanas do código de 0 e 1 da máquina. Linguagens de alto nível criam produtos mais complexos, com menos linhas de código.

Figura 2 – Diferença entre algumas linguagens

 Dart	<pre>class Segment {   int links = 4;   toString() =&gt; "I have \$links links"; }</pre>
 Kotlin	<pre>class Segment {   var links: Int = 4   override fun toString()= "I have \$links links" }</pre>
 Swift	<pre>class Segment: CustomStringConvertible {   var links: Int = 4   public var description: String { return     "I have \(links) links" } }</pre>
 TypeScript	<pre>class Segment {   links: number = 4   public toString = () : string =&gt; { return     `I have \${this.links} links` }; }</pre>

Fonte: Dart, 2022.

A Figura 2 traz exemplos de códigos que fazem a mesma coisa, mas com escrita em linguagens diferentes. O código da figura gera uma String, uma sequência de caracteres, que informa a quantidade de links que a aplicação apresenta.

Outro objetivo do Dart é proporcionar um ambiente de desenvolvimento altamente produtivo. Para isso, o Dart utiliza os seguintes recursos:

- **Hot reload:** traduzindo do inglês ao pé da letra, temos a ideia de “recarregamento quente”. Em informática, o termo “hot/quente” pode significar que algo está em funcionamento, sendo energizado/ligado, diferentemente do “cold/frio”, que significa que algo está desligado. Uma das características mais atrativas do Dart é a capacidade de alterar o código e analisar os seus efeitos na aplicação, sem precisar abrir e fechar a aplicação – ou seja, o desenvolvedor pode alterar o código e carregar esse novo código na aplicação enquanto ela está em funcionamento.
- **Ferramentas de análise de código:** o Dart tem ferramentas que permitem que o código seja analisado, para que os erros possam ser apontados antes mesmo da execução de uma linha sequer de código. Assim como demonstra a Figura 3, na própria tela do código, o Dart aponta possíveis erros que podem ser corrigidos, com sugestões para deixar o código mais limpo e de fácil manutenção.

Figura 3 – Exemplo de análise estáticas



Fonte: Dart, 2022.



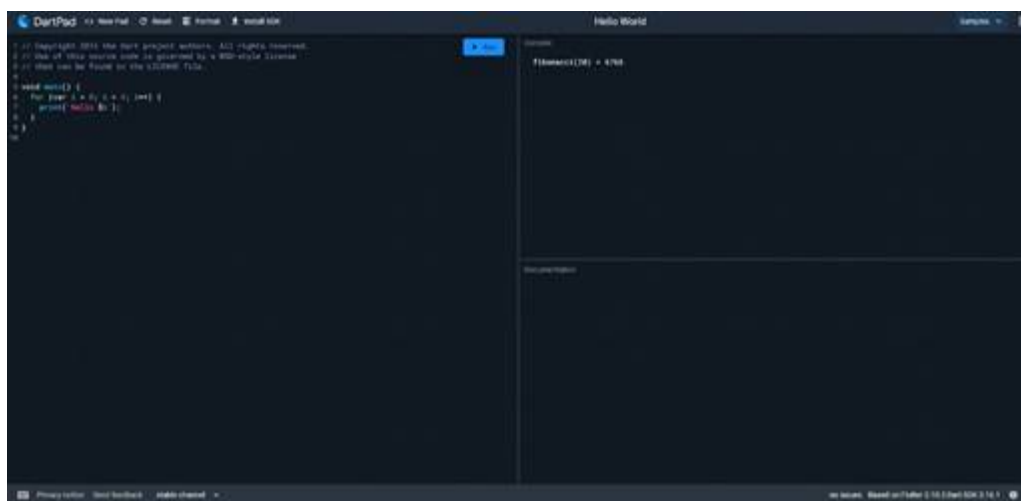
desenvolvimento, executado em tempo real para a utilização de ferramentas como o debugger de código e o hot reload.

- **Dart Web:** ferramenta que permite que o Dart funcione em aplicações web via JavaScript. O código Dart é compilado em JavaScript, e então operado em qualquer navegador com suporte a JavaScript.

## 1.2 DARTPAD

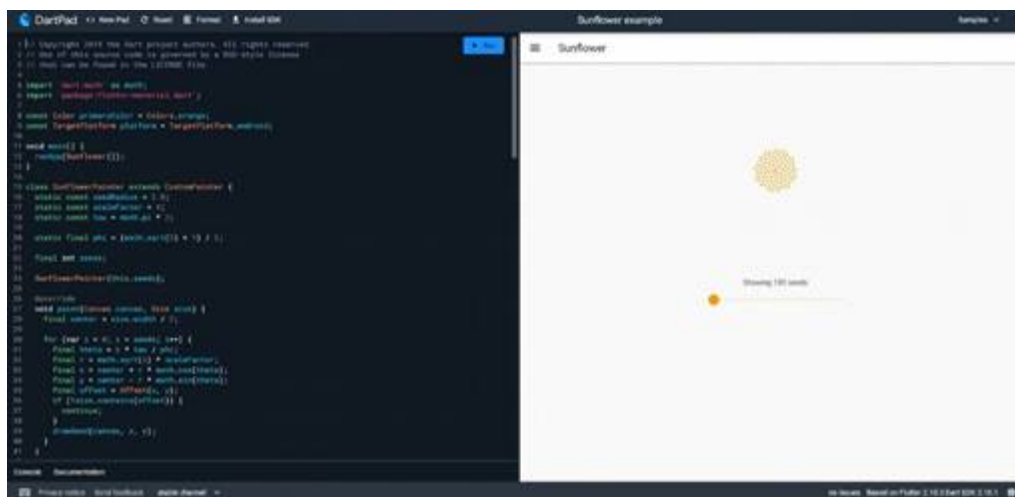
DartPad é um ótimo compilador online para a realização de pequenos testes de código Dart, sem um computador que rode editores e emuladores mais pesados. Opensource, o Dartpad funciona em todos os navegadores modernos, podendo ser embarcado em páginas HTML. O link para acesso ao DartPad é: <http://www.dartpad.dev/> (Acesso em: 17 maio 2022).

Figura 6 – Tela do DartPad com exemplo Hello World



Fonte: Dart, 2022.

Figura 7 – Tela do DartPad com exemplo de Flutter



Fonte: Dart, 2022.

As figuras 6 e 7 apresentam códigos de exemplo que podem ser encontrados na página. O Dartpad é bastante versátil para testes rápidos, já que não exige muito do computador; porém, funcionalidades não funcionam no Dartpad. O Dartpad não tem suporte para algumas bibliotecas de funções, como o `dart.io`, que apresenta as funções de entrada e saída do usuário, além de não permitir a estruturação de projeto.

## TEMA 2 – DART: OPERADORES E VARIÁVEIS

Neste tópico, vamos nos familiarizar com os operadores básicos e variáveis do Dart. Os leitores que já têm conhecimento em outras linguagens de programação vão entender que a base costuma ser bastante similar entre as linguagens, porém sempre existem diferenças. Quando o leitor tem mais familiaridade com a linguagem, poderá utilizar essa seção como revisão de sintaxes.

### Saiba mais

Caso esteja interessado em um tour mais longo e completo, o site do Dart oferece um guia em inglês: <https://dart.dev/guides/language/language-tour>. Acesso em: 17 maio 2022.

## 2.1 OPERADORES

Os operadores usados são bastante comuns na grande maioria das linguagens de programação, mas é sempre bom revisar com quais ferramentas vamos trabalhar.

Primeiramente, os operadores aritméticos:

- $X + Y$  : operador de adição de X mais Y
- $X - Y$  : operador de subtração de X menos Y
- $X * Y$  : operador de multiplicação de X vezes Y
- $X / Y$  : operador de divisão de X dividido por Y
- $X \sim / Y$  : operador de divisão, mas retornando apenas resultados inteiros da operação de X dividido por Y
- $X \% Y$  : o restante de uma operação de X dividido por Y, também conhecido como *módulo*
- $X++$  : incremento, funciona como uma operação de "X recebe o valor de  $X + 1$ "
- $X--$  : decremento, funciona como uma operação de "X recebe o valor de  $X - 1$ "

Os operadores relacionais testam e definem relação entre dois operandos. Operadores relacionais retornam valores booleanos (verdadeiro ou falso).

- $X > Y$ : testa ou define se X é maior que Y
- $X < Y$ : testa ou define se X é menor que Y
- $X \geq Y$ : testa ou define se X é maior ou igual a Y
- $X \leq Y$ : testa ou define se X é menor ou igual a Y
- $X == Y$ : testa ou define se X é igual a Y
- $X != Y$ : testa ou define se X é diferente de Y

Quadro 1 – Exemplos

	X = 10, Y = 20	X = 15, Y = 15	X = 20, Y = 10
$X > Y$	<b>Falso</b>	<b>Falso</b>	<b>Verdadeiro</b>
$X < Y$	<b>Verdadeiro</b>	<b>Falso</b>	<b>Falso</b>
$X \geq Y$	<b>Falso</b>	<b>Verdadeiro</b>	<b>Verdadeiro</b>
$X \leq Y$	<b>Verdadeiro</b>	<b>Verdadeiro</b>	<b>Falso</b>
$X == Y$	<b>Falso</b>	<b>Verdadeiro</b>	<b>Falso</b>
$X != Y$	<b>Verdadeiro</b>	<b>Falso</b>	<b>Verdadeiro</b>

Perceba que os 6 operadores relacionais apresentam todas as 6 combinações de verdadeiro e falso para os 3 casos citados. As outras duas situações possíveis seriam tudo verdadeiro, ou tudo



falso, mas nesses casos não precisamos de um operador para analisar se o caso é verdadeiro ou falso, já que o resultado sempre será verdadeiro, ou sempre falso.

Existem também operadores que testam o tipo de uma variável. Vamos estudar os tipos de variáveis no próximo tópico.

- `X is T`: testa se a variável `X` é do tipo `T`
- `X is! T`: testa se a variável `X` não é do tipo `T`

Assim como vimos em Lógica, o Dart pode se utilizar de operadores lógicos, assim como demonstramos a seguir:

- `&&`: operador AND, também conhecido como “e” lógico
- `||`: operador OR, também conhecido como “ou” lógico
- `NOT` : operador NOT, também conhecido como “não” lógico

Os operadores de designação são operadores que alteram o valor de uma variável logo após a realização da operação. Essas operações ajudam a simplificar a programação, reduzindo a necessidade de variáveis de auxílio.

- `X = Y`: designação simples, `X` recebe o valor de `Y`
- `X += Y`: designação de adição, `X` recebe a soma do valor de `X` e `Y`
- `X -= Y`: designação de subtração, `X` recebe a subtração do valor de `X` por `Y`
- `X *= Y`: designação de multiplicação, `X` recebe a multiplicação dos valores de `X` e `Y`
- `X /= Y`: designação de divisão, `X` recebe a divisão do valor de `X` pelo valor de `Y`
- `X %= Y`: designação de restante de divisão, `X` recebe o restante da divisão de `X` por `Y`

Quadro 2 – Exemplos

X antes	Y antes	Operação	X depois	Y depois
10	15	<code>X = Y</code>	15	15
10	15	<code>X += Y</code>	25	15
10	15	<code>X -= Y</code>	-5	15
10	15	<code>X *= Y</code>	150	15
10	15	<code>X /= Y</code>	0.66666666	15

10	15	X %= Y	10	15
----	----	--------	----	----

## 2.2 VARIÁVEIS

Assim como acontece com a maioria das linguagens de programação, uma variável precisa ser declarada antes de ser utilizada. Nessa declaração, algumas informações importantes são definidas, como o nome pelo qual a variável será chamada, o tipo de variável, e possivelmente o valor da variável, dados que podem ser configurados na declaração.

Vejamos alguns exemplos de tipos de variáveis básicas:

- var: abreviação para "VARiável". Tecnicamente, "var" não é um tipo de variável. Quando se cria uma variável "var", o desenvolvedor deixa que o programa defina o tipo de variável, com base no que o desenvolvedor fizer com ela.
- int: abreviação para "INTeiro", uma variável básica para armazenamento de números inteiros.
- Bool: abreviação para "BOOLEano", uma variável que armazena apenas os valores Verdadeiro ou Falso.
- String: do inglês fio/corda, com o significado de sequência, essa variável armazena uma sequência de caracteres como nomes, frases etc.
- Double: do inglês, duplo/dobro, uma variável com o dobro da precisão numérica, pois é capaz de comportar mais bits

Além disso, existem as coleções. Uma coleção pode ser um entre os seguintes três subtipos de variáveis:

- List: do inglês lista, uma coleção ordenada de objetos, em que é possível usar índices para acessar um objeto da lista.
- Set: do inglês conjunto, uma coleção não ordenada de objetos, ou seja, não é possível usar índices para acessar um objeto de um conjunto. Porém, um conjunto verifica e impede a entrada de objetos duplicados.
- Map: do inglês mapa, também é uma coleção não ordenada, porém possibilita acessar os objetos do mapa através de chaves em vez de índices numéricos.

**Saiba mais**

Caso o leitor esteja interessado em uma coleção mais completa de variáveis possíveis, a documentação do site apresenta mais informações: <https://api.dart.dev/stable/2.16.1/dart-core/dart-core-library.html>. Acesso em: 17 maio 2022.

## TEMA 3 – DART: CLASSES

As classes no Dart são declaradas utilizando a palavra-chave “class”. Durante a criação de uma classe, é possível definir um conjunto de informações e características, além de personalizar a classe para conseguir as funcionalidades de que o desenvolvedor necessita.

Uma classe comumente apresenta os seguintes membros:

- Campos: nome mais técnico para as variáveis da sua classe.
- Acessos: funções de “get” e “set” da sua classe. Essas funções têm como objetivo alterar e acessar as variáveis da classe.
- Construtor: método especial para criar um objeto da classe.
- Métodos: funções de uma classe, que definem as suas funcionalidades. Acessos e construtores são considerados métodos.

Figura 8 – Exemplo de código

```
1 //classe food
2 class Comida{
3
4     String nome;
5     int quantidade;
6
7     //construtor de comida
8     Comida(this.nome, this.quantidade);
9
10    //acessor set de quantidade
11    set setQuantidade(int novaQuantidade){
12        quantidade = novaQuantidade;
13    }
14
15    //acessor get de nome
16    String get nomeDaComida{
17        return nome;
18    }
19
20    //método que passa uma String com a quantidade e nome
21    String estoque(){
22        return `Quantidade x $nome`;
23    }
24
25 }
26
27 //main
28 void main() {
29     //declara e constrói um objeto da classe comida
30     Comida comida1 = Comida("Hamburger", 4);
31
32     //imprime o retorno do método estoque do objeto comida1
33     print(comida1.estoque());
34
35     //usa o acessor set para mudar a quantidade de comida1
36     comida1.setQuantidade = 2;
37
38     //imprime na tela a situação do estoque de comida1
39     print("Estoque de " + comida1.nomeDaComida + " foram alterados.");
40
41     //imprime o retorno do método estoque do objeto comida1
42     print(comida1.estoque());
43 }
```

Fonte: Dartpad, 2022.

A Figura 8 apresenta um exemplo de código com os membros de classe mencionados. Na imagem, temos duas classes, a classe Comida na linha 1, e a classe Main na linha 28.

- Campos: a classe comida tem os campos nome e quantidade nas linhas 4 e 5; a classe main tem um campo Comida na linha 30.
- Acessos: a classe Comida possui os acessos de set para a quantidade na linha 11, e o acesso de get do nome na linha 16; a classe main não apresenta nenhum acesso de set ou get.
- Construtores: a classe Comida tem um construtor na linha 8 que define os valores de nome e quantidade de objeto; a classe main não tem um construtor.
- Métodos: além dos acessos e do construtor, a classe Comida tem um método chamado estoque, que retorna uma String com a quantidade e o nome do objeto da classe Comida.

Figura 9 – Resultado do exemplo de código

A screenshot of a dark-themed console window. The word "Console" is at the top left. Below it, the output of a Dart program is displayed in three lines: "4 x Hamburger", "Estoque de Hamburger foram alterados.", and "2 x Hamburger".

```
Console  
  
4 x Hamburger  
Estoque de Hamburger foram alterados.  
2 x Hamburger
```

Fonte: Dartpad, 2022.

A Figura 9 mostra o resultado quando se executa o código apresentado na Figura 8. O código apresentado cria um objeto da classe Comida, com o nome Hamburger, e a quantidade 4. Assim, ele imprime na tela o estoque desse objeto, muda a quantidade, imprime na tela que o estoque foi alterado, e imprime o novo estoque.

### 3.1 TOSTRING

Em Dart, as classes apresentam um método chamado `toString()`. O objetivo desse método é que um objeto dê informações sobre ele mesmo quando o método for chamado. A Figura 10 apresenta o mesmo código da Figura 8, com a diferença de que a função de estoque foi trocada pela `toString()`; na classe main, estamos agora chamando o método de `toString()`, em vez de `estoque`.

Figura 10 – Código exemplo com `toString()`

```
1 //classe food
2 class Comida{
3
4     String nome;
5     int quantidade;
6
7     //construtor de comida
8     Comida(this.nome, this.quantidade);
9
10    //acessor set de quantidade
11    set setQuantidade(int novaQuantidade){
12        quantidade = novaQuantidade;
13    }
14
15    //acessor get de nome
16    String get nomeDaComida{
17        return nome;
18    }
19
20    //método que passa uma String com a quantidade e nome
21    String toString() => "$quantidade x $nome";
22
23 }
24
25 //main
26 void main() {
27     //declara e constrói um objeto da classe comida
28     Comida comida1 = Comida("Hamburger", 4);
29
30     //imprime o retorno do método estoque do objeto comida1
31     print(comida1.toString());
32
33     //usa o acessor set para mudar a quantidade de comida1
34     comida1.setQuantidade = 2;
35
36     //imprime na tela a situação do estoque de comida1
37     print("Estoque de " + comida1.nomeDaComida + " foram alterados.");
38
39     //imprime o retorno do método estoque do objeto comida1
40     print(comida1.toString());
41 }
42
```

Fonte: Dartpad, 2022.

## TEMA 4 – CLASSES ABSTRATAS E INTERFACEAMENTO

Em Dart, assim como em outras linguagens de programação, é possível utilizar classes abstratas e interfaces para criar classes que herdam métodos de outras classes.

### 4.1 CLASSES ABSTRATAS E IMPLEMENTAÇÃO

Em linguagens de programação orientadas a objeto, classes abstratas servem como base para outras classes. As classes abstratas não podem ser usadas para criar objetos diretamente; porém, quando usadas como base para outras classes, permitem que as outras classes usem os seus métodos.

Figura 11 – Exemplo de código com abstrato

```
1 void main(){
2
3     Pessoa p1 = Boxeador();
4     Pessoa p2 = Pessoa();
5     p1.setForca = 10;
6     print('Força do Soco do Boxeador: ${p1.soco()}');
7
8 }
9
10 abstract class Pessoa{
11
12     int soco();
13     set setForca(int forca);
14
15 }
16
17 class Boxeador implements Pessoa{
18
19     int forca = 0;
20
21     Boxeador();
22
23     @override
24     set setForca(int forca){
25         this.forca = forca;
26     }
27
28     @override
29     int soco(){
30         return forca;
31     }
32 }
```

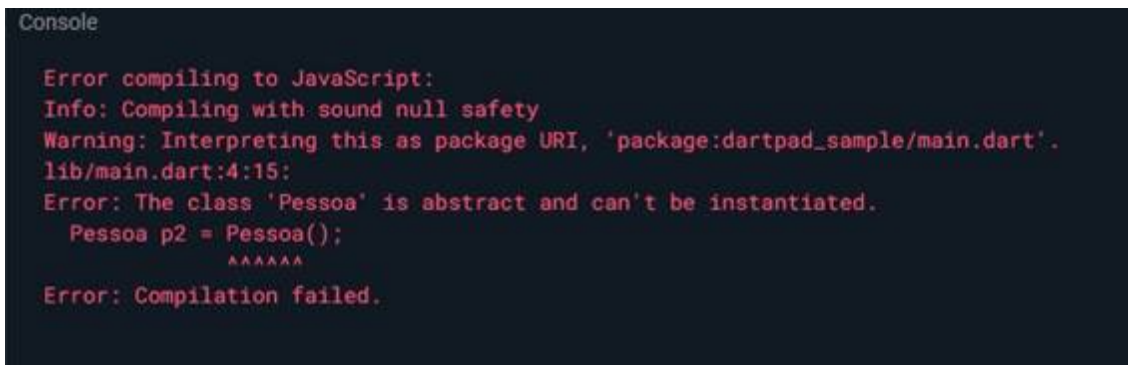
Fonte: Dartpad, 2022.

A Figura 11 traz um exemplo de código com classe abstrata. Na linha 10, temos a classe abstrata Pessoa, que cria métodos vazios de soco e um acesso do tipo set da força. Na linha 17, temos a classe Boxeador que implementa a classe Pessoa.

Para implementar uma classe abstrata, temos que tomar alguns cuidados. Quando implementamos uma classe abstrata, devemos implementar todos os seus métodos, no caso do método "soco", usamos o método de acesso do tipo set para a força. A anotação "@Override" é opcional, mas é importante sempre deixar um código bem documentado e anotado quando você sobrescreveu (do inglês, *override*) um método.

Nas figuras 12 e 13, vemos duas saídas diferentes para o código da Figura 11: uma delas da forma como está, e a outra removendo a linha 4.

Figura 12 – Saída do Código da Figura 11

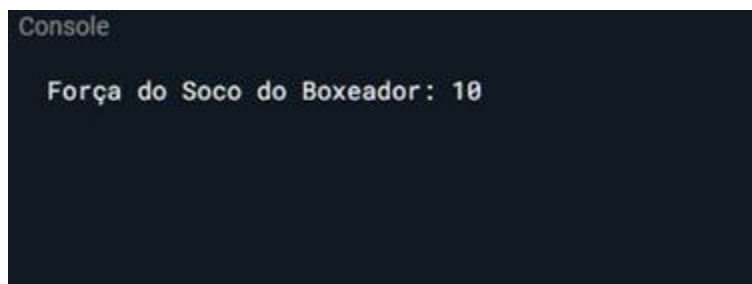


```
Console
Error compiling to JavaScript:
Info: Compiling with sound null safety
Warning: Interpreting this as package URI, 'package:dartpad_sample/main.dart'.
lib/main.dart:4:15:
Error: The class 'Pessoa' is abstract and can't be instantiated.
  Pessoa p2 = Pessoa();
                ^^^^^^
Error: Compilation failed.
```

Fonte: Dartpad, 2022.

Como mencionamos anteriormente, isso é o que acontece quando executamos o código da forma como aparece na Figura 11. A figura apresenta um erro, pois tentamos instanciar um objeto da classe Pessoa, porém a classe Pessoa é uma classe abstrata.

Figura 13 – Código



```
Console
Força do Soco do Boxeador: 10
```

Fonte: Dartpad, 2022.

A Figura 13 apresenta o resultado quando removemos a linha 4 do código da Figura 11, que tenta instanciar um objeto de uma classe abstrata. A saída é o valor de força do boxeador p1, obtido através do método soco.

## 4.2 EXTENSÃO

Extensões são uma outra forma a partir da qual uma classe pode obter métodos de outra classe. Uma implementação é usada a princípio para implementar uma classe abstrata, enquanto uma extensão é utilizada para criar uma classe filha que herda as suas propriedades.

A Figura 14 traz um exemplo de extensão, aproveitando o código da Figura 11.



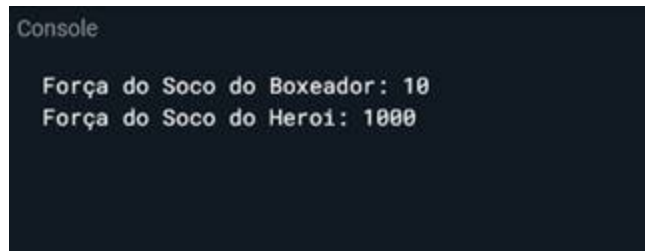
Figura 14 – Exemplo de código com extensão

```
1 void main(){
2
3     Pessoa p1 = Boxeador();
4     Pessoa p2 = Heroi();
5     p1.setForca = 10;
6     p2.setForca = 10;
7     print('Força do Soco do Boxeador: ${p1.soco()}');
8     print('Força do Soco do Heroi: ${p2.soco()}');
9
10 }
11
12 abstract class Pessoa{
13
14     int soco();
15     set setForca(int forca);
16
17 }
18
19 class Boxeador implements Pessoa{
20
21     String nome = "Sem Nome";
22     int forca = 0;
23
24     Boxeador();
25
26     @override
27     set setForca(int forca){
28         this.forca = forca;
29     }
30
31     @override
32     int soco(){
33         return forca;
34     }
35 }
36
37 class Heroi extends Boxeador{
38
39     @override
40     int soco(){
41         return 100*forca;
42     }
43 }
```

Fonte: Dartpad, 2022.

Como podemos observar na Figura 14, na linha 37 temos a classe herói, que estende a classe boxeador. Lembramos que uma classe pode estender qualquer classe, não necessariamente uma classe que implementa outra classe. O exemplo utilizado ajuda a mostrar que é possível encadear extensões. Na Figura 15, vemos o resultado do código da Figura 14.

Figura 15 – Saída do código da figura 14



```
Console  
  
Força do Soco do Boxeador: 10  
Força do Soco do Hero1: 1000
```

Fonte: Dartpad, 2022.

## TEMA 5 – DART: ASSÍNCRONO E FUTURO

Dart é uma linguagem de programação que opera em uma única “thread”, ou fio, o que significa que todo o programa opera em sequência. Simplificando, isso quer dizer que o aplicativo começa a executar um código que demora para terminar. O aplicativo ficará parado esperando o código terminar de executar. Para o usuário, parece que o aplicativo “congelou” e parou de funcionar, portanto é importante usar essas duas funções para trabalhar ao redor dessa limitação.

Para um melhor entendimento, vamos observar o caso da figura a seguir.

Figura 16 – Exemplo de código sem assincronia

```
1  import 'dart:io';
2
3  void funcaoLenta() {
4
5      sleep(Duration(seconds: 5));
6
7      print("Operação lenta de 5 segundos.");
8
9  }
10
11  Run | Debug
12  main() {
13
14      final stopwatch = Stopwatch();
15      stopwatch.start();
16
17      print("Início da Aplicação. Chamando Função Lenta.");
18      print(stopwatch.elapsedMilliseconds);
19
20      funcaoLenta();
21      print(stopwatch.elapsedMilliseconds);
22
23      print("Continuando a Main.");
24      print(stopwatch.elapsedMilliseconds);
25
26      sleep(Duration(seconds: 1));
27      print("Executado operação rápida de 1 segundo.");
28      print(stopwatch.elapsedMilliseconds);
29
30      sleep(Duration(seconds: 1));
31      print("Executado outra operação rápida de 1 segundo.");
32      print(stopwatch.elapsedMilliseconds);
33
34      sleep(Duration(seconds: 7));
35      print("Executado operação lenta de 7 segundos.");
36      print(stopwatch.elapsedMilliseconds);
37
38      print("Final da Main.");
39      print(stopwatch.elapsedMilliseconds);
40  }
```

Fonte: Dartpad, 2022.

Primeiramente, vamos iniciar pela linha 11, onde a nossa main começa.

- Um "stopwatch", ou um contador de tempo, é criado na linha 13 e iniciado na linha 14.
- Na linha 19, a main inicia a nossa função lenta.
- A função lenta é uma função que leva 5 segundos para ser executada.
- A main executa uma função que leva 1 segundo para ser executada.
- A main então executa novamente a função que leva 1 segundo para ser executada.
- Finalmente, antes de terminar o programa, o código executa uma função que leva 7 segundos para ser executada.

Fazendo as contas, vemos que, se um programa executa todas essas funções uma após a outra, levará um total de 14 segundos esperando que as funções sejam executadas.

Figura 17 – Saída da execução do exemplo sem assincronia

```
Connecting to VM Service at http://127.0.0.1:55795/JewaYcMqrx0=/  
Início da Aplicação. Chamando Função Lenta.  
0  
Operação lenta de 5 segundos.  
5006  
Continuando a Main.  
5006  
Executado operação rápida de 1 segundo.  
6012  
Executado outra operação rápida de 1 segundo.  
7016  
Executado operação lenta de 7 segundos.  
14025  
Final da Main.  
14025  
Exited
```

Fonte: Dartpad, 2022.

Podemos observar a saída do programa pela Figura 17. Temos os textos que são impressos na tela pelas funções print. Logo abaixo de cada uma delas, temos o tempo em milissegundos, desde que o contador de tempo foi iniciado. Note que o programa levou um total de 14 segundos, assim como previsto. Podemos melhorar esse tempo com o uso de assincronia.

Figura 18 – Exemplo de código com assincronia

```
1  import 'dart:io';
2  import 'dart:async';
3
4  Future funcaoLenta() async{
5
6      await Future.delayed(Duration(seconds: 5));
7
8      print("Operação lenta de 5 segundos.");
9
10 }
11
12 Run | Debug
13 main() {
14
15     final stopwatch = Stopwatch();
16     stopwatch.start();
17
18     print("Início da Aplicação. Chamando Função Lenta.");
19     print(stopwatch.elapsedMilliseconds);
20
21     funcaoLenta();
22     print(stopwatch.elapsedMilliseconds);
23
24     print("Continuando a Main.");
25     print(stopwatch.elapsedMilliseconds);
26
27     sleep(Duration(seconds: 1));
28     print("Executado operação rápida de 1 segundo.");
29     print(stopwatch.elapsedMilliseconds);
30
31     sleep(Duration(seconds: 1));
32     print("Executado outra operação rápida de 1 segundo.");
33     print(stopwatch.elapsedMilliseconds);
34
35     sleep(Duration(seconds: 7));
36     print("Executado operação lenta de 7 segundos.");
37     print(stopwatch.elapsedMilliseconds);
38
39     print("Final da Main.");
40     print(stopwatch.elapsedMilliseconds);
41 }
```

Fonte: Dartpad, 2022.

A Figura 18 apresenta uma versão alterada do código da Figura 17. As alterações foram feitas na função lenta, integrando futuro e assincronia. Veremos os resultados dessas alterações na próxima figura.

Figura 19 – Saída da execução do exemplo com assincronia

```
Connecting to VM Service at http://127.0.0.1:64372/ETCxmUi0bsw=/
Início da Aplicação. Chamando Função Lenta.
0
5
Continuando a Main.
5
Executado operação rápida de 1 segundo.
1015
Executado outra operação rápida de 1 segundo.
2029
Executado operação lenta de 7 segundos.
9040
Final da Main.
9041
Operação lenta de 5 segundos.
Exited
```

Fonte: Dartpad, 2022.

A figura 19 mostra o resultado com o uso de futuro e assincronia. A função lenta foi chamada no começo. Em vez de parar, a main, como no exemplo anterior, ficou rodando por trás, enquanto a main realizava as suas operações. Podemos observar pelo contador de tempo que, dessa forma, economizamos 5 segundos do tempo de execução do programa, um ganho de 35%. Se dividimos os passos que foram executados, teremos a seguinte sequência:

- Um “stopwatch”, ou um contador de tempo, é criado na linha 14 e iniciado na linha 15.
- Na linha 20, a main inicia a função lenta.
- A função lenta é uma função que leva 5 segundos para ser executada. Porém, ela não trava a main, que continua com as suas operações.
- A main executa uma função que leva 1 segundo para ser executada.
- A main então executa novamente a função que leva 1 segundo para ser executada.
- Finalmente, o código executa uma função que leva 7 segundos para ser executada.
- Durante essa última execução, a função lenta termina a operação de 5 segundos, mas ela espera 7 segundos para poder finalizar a execução.
- Quando a função de 7 segundos termina, a função lenta ganha prioridade novamente, para imprimir a mensagem na tela.

## FINALIZANDO

Nesta etapa, desenvolvemos uma base de Dart, em suas diferenças com outras linguagens de programação. Estudamos também algumas técnicas de programação que são bastante comuns em ambientes de desenvolvimento de código. Esta etapa servirá como base para a programação com Flutter, como veremos posteriormente.

## REFERÊNCIAS

DART. Disponível em: <<https://dart.dev/>>. Acesso em: 14 maio 2022.

DARTPAD. Disponível em: <<https://dartpad.dev/?>>. Acesso em: 14 maio 2022.

FLUTTER. **Using the Performance view.** Disponível em: <<https://docs.flutter.dev/development/tools/devtools/performance>>. Acesso em: 14 maio 2022.