

Sintaxe da Linguagem

Curso Flutter de Verão



- Explorar os conceitos de programação assíncrona em Dart.
- Apresentar as estruturas Future e Stream em Dart.
- Discutir as palavras-chave async e await em Dart.
- Realizar exercícios para aplicar os conceitos aprendidos na programação assíncrona.

Objetivo da aula de hoje



Revisão da aula anterior

Programação assíncrona



A programação assíncrona executa tarefas demoradas sem bloquear o programa, melhorando a responsividade do aplicativo.

Ela permite que o programa continue sua execução enquanto aguarda o término das tarefas, resultando em uma experiência mais fluída para o usuário. Isso é especialmente útil em operações que envolvem acesso a recursos externos, como chamadas de API ou acesso a banco de dados.

Contextos assíncronos

1. **Chamadas de API:** Quando precisamos fazer requisições a serviços externos, é importante realizar essa operação de forma assíncrona para evitar bloqueios na execução do programa enquanto aguardamos a resposta.
2. **Operações de E/S:** Qualquer operação que envolva entrada e saída (E/S) de dados pode ser demorada. A programação assíncrona permite executar essas operações de forma não bloqueante, garantindo a responsividade do programa durante o processo.
3. **Eventos de Interface do Usuário:** Em jogos ou aplicativos com animações, é comum lidar com eventos em tempo real. A programação assíncrona permite tratar esses eventos de forma responsiva, mantendo a fluidez da interação com o usuário.
4. **Processamento de dados em lote:** Quando temos grandes volumes de dados para processar, a programação assíncrona pode ser aplicada para dividir o processamento em tarefas menores e executá-las paralelamente, otimizando o desempenho geral.

Funções Future e Stream

O **Future** representa um valor que estará disponível no futuro. É utilizado quando esperamos a conclusão de uma operação assíncrona, como uma chamada de API ou acesso a um banco de dados.

O **Stream** é utilizado para trabalhar com fluxos contínuos de dados assíncronos. Ele permite a transmissão e recebimento de dados no futuro, à medida que eles se tornam disponíveis.

Com **Future** e **Stream**, podemos implementar operações assíncronas de forma eficiente, garantindo uma melhor responsividade e experiência para o usuário em aplicações Dart.

Future

Uma função `Future` permite que você execute uma tarefa em segundo plano enquanto o restante do programa continua a ser executado. Quando a tarefa em segundo plano é concluída, o resultado é retornado por meio do objeto `Future`. Isso permite que você continue trabalhando com outros aspectos do programa sem bloquear a execução.

Exemplo

```
Future<int> totalCookiesCount() async {  
  int cookiesCount = await lookupTotalCookiesCountDatabase();  
  return cookiesCount;  
}  
  
totalCookiesCount().then((count) {  
  print('cookiesCount: ${count}');  
});
```


Métodos e funções

Future.delayed: Uma função estática que retorna um futuro que será concluído após um determinado período de tempo especificado.

Future.value: Uma função estática que retorna um futuro já completado com um determinado valor.

then: Um método que permite encadear uma ação ou uma função de retorno de chamada quando o futuro é concluído com sucesso.

catchError: Um método que permite definir um tratamento de erro para o futuro. É executado quando ocorre uma exceção no futuro.

whenComplete: Um método que permite definir uma ação a ser executada quando o futuro é concluído, independentemente de sucesso ou erro.

Exemplo

```
Future.delayed(Duration(seconds: 2)).then((_) {  
    print("Passaram-se 2 segundos!");  
});  
  
Future<String> futureValue = Future.value("Olá, mundo!");  
  
Future<int> future = obterNumero();  
  
future.then((numero) {  
    print("Número: $numero");  
});  
  
future.catchError((erro) {  
    print("Ocorreu um erro: $erro");  
});  
  
future.whenComplete(() {  
    print("Operação concluída");  
});
```

Exemplo

```
Future<String> fetchData() async {  
  final response = await http.get(Uri.parse('https://api.example.com/data'));  
  if (response.statusCode == 200) {  
    return response.body;  
  } else {  
    throw Exception('Falha ao carregar os dados');  
  }  
}  
  
FutureBuilder<String>(  
  future: fetchData(),  
  builder: (context, snapshot) {  
    if (snapshot.connectionState == ConnectionState.waiting) {  
      return CircularProgressIndicator();  
    } else if (snapshot.hasError) {  
      return Text('Ocorreu um erro: ${snapshot.error}');  
    } else {  
      return Text('Dados carregados: ${snapshot.data}');  
    }  
  },  
)
```

async/await

`async/await` é uma construção de linguagem no Dart que facilita a programação assíncrona, permitindo que você escreva código de maneira síncrona, mesmo quando está lidando com operações assíncronas.

Quando você declara uma função como `async`, você pode usar a palavra-chave `await` para aguardar a conclusão de uma operação assíncrona antes de continuar a execução do código. Isso permite que você escreva código de forma sequencial, como se estivesse trabalhando com código síncrono.

Exemplo

```
Future<void> fetchData() async {  
  try {  
    final data = await fetchSomeData(); // Aguarda a conclusão de uma operação assíncrona  
  
    print('Dados recebidos: $data');  
  } catch (error) {  
    print('Ocorreu um erro: $error');  
  }  
}  
  
Future<String> fetchSomeData() {  
  return Future.delayed(Duration(seconds: 2), () {  
    return 'Dados importantes';  
  });  
}  
  
void main() {  
  fetchData();  
  print('Continuando a execução...');  
}
```

Stream

Um `stream` é uma sequência de eventos ordenados que podem ser transmitidos e processados de forma assíncrona à medida que são produzidos. Os streams são úteis para lidar com operações assíncronas, como leitura de arquivos, chamadas de API e interações com bancos de dados.

Para trabalhar com streams em Dart, você pode usar as classes `Stream` e `StreamController`. O `Stream` representa uma sequência de eventos assíncronos e o `StreamController` é usado para controlar a produção desses eventos.

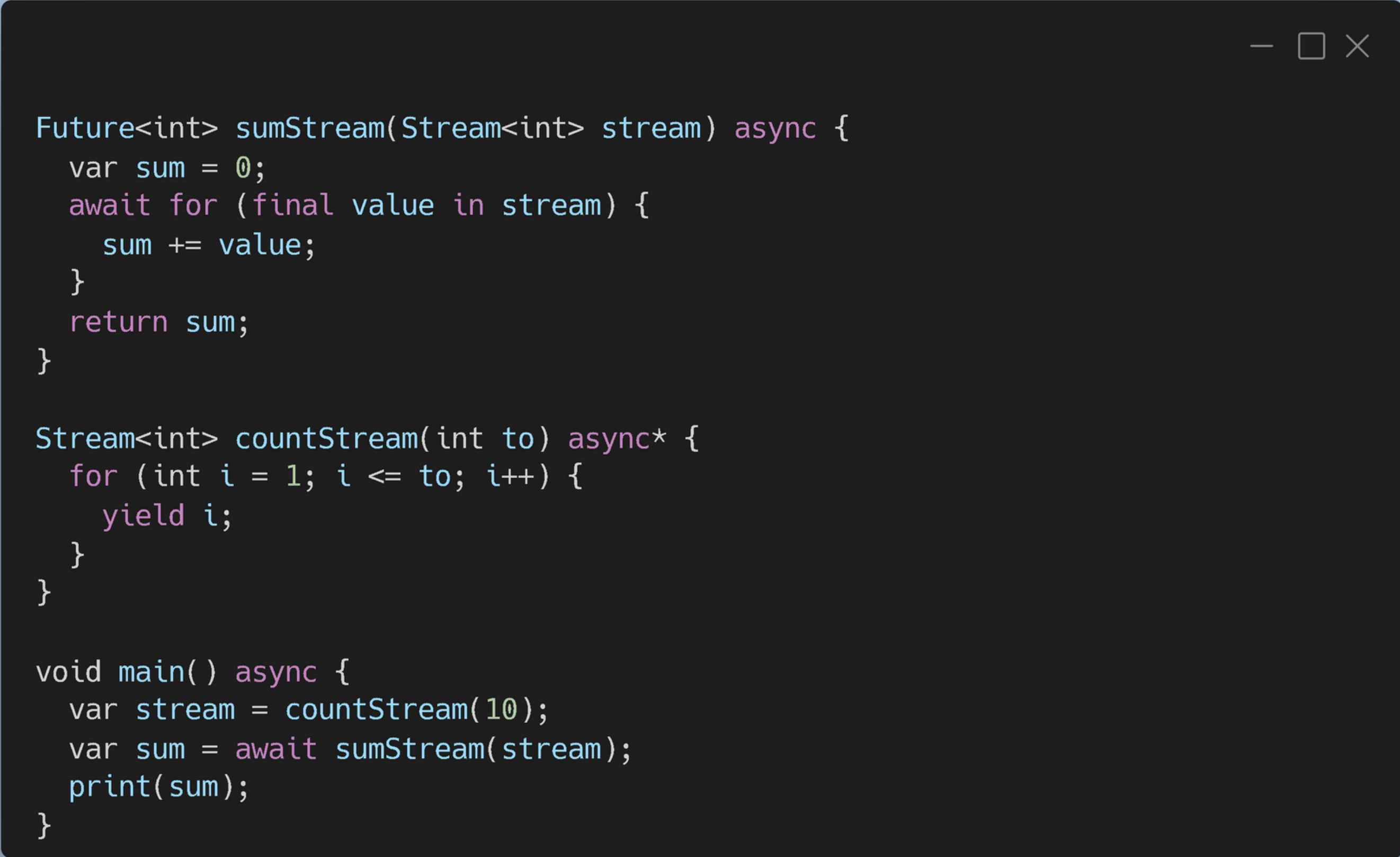
Exemplo

```
import 'dart:async';

Stream<int> countNumbers(int max) async* {
  for (int i = 1; i <= max; i++) {
    yield i;
    await Future.delayed(const Duration(milliseconds: 500));
  }
}

void main() {
  final Stream<int> numberStream = countNumbers(5);

  numberStream.listen((number) {
    print('Número: $number');
  });
}
```



```
Future<int> sumStream(Stream<int> stream) async {  
    var sum = 0;  
    await for (final value in stream) {  
        sum += value;  
    }  
    return sum;  
}
```

```
Stream<int> countStream(int to) async* {  
    for (int i = 1; i <= to; i++) {  
        yield i;  
    }  
}
```

```
void main() async {  
    var stream = countStream(10);  
    var sum = await sumStream(stream);  
    print(sum);  
}
```


async*/yield

Você adiciona a palavra-chave `async*` para criar uma função que retorna vários valores futuros, um de cada vez. Os resultados são encapsulados em um Stream.

O termo técnico para isso é função geradora assíncrona. Você usa `yield` para retornar um valor em vez de `return` porque não está saindo da função.

```
Stream<int> countForOneMinute() async* {  
    for (int i = 1; i <= 60; i++) {  
        await Future.delayed(const Duration(seconds: 1));  
        yield i;  
    }  
}  
  
// Consumindo Stream  
await for (int i in countForOneMinute()) print(i);
```

Você pode usar `await for` para aguardar cada valor emitido pelo Stream.

Uso do StreamController

```
import 'dart:async';

void main() {
  final streamController = StreamController<int>();

  final stream = streamController.stream;
  final subscription = stream.listen((data) {
    print('Recebido: $data');
  });

  for (var i = 1; i <= 5; i++) {
    streamController.sink.add(i);
  }

  streamController.close();
  subscription.cancel();
}
```

Exemplo em Flutter

```
Stream<int> countStream() async* {  
  for (int i = 1; i <= 5; i++) {  
    await Future.delayed(Duration(seconds: 1));  
    yield i;  
  }  
}  
  
StreamBuilder<int>(  
  stream: countStream(),  
  builder: (context, snapshot) {  
    if (snapshot.hasData) {  
      return Text('Count: ${snapshot.data}');  
    } else if (snapshot.hasError) {  
      return Text('Error: ${snapshot.error}');  
    } else {  
      return CircularProgressIndicator();  
    }  
  },  
)
```

Métodos e funções

`Stream.periodic`: Cria um fluxo que emite eventos repetidamente em intervalos de período.

`Stream.value`: Cria um fluxo que emite um único evento de dados antes de fechar.

`Stream.fromFuture`: Cria uma nova Stream de subscription única do futuro.

`single`, `first`, `last`: O único elemento desta Stream.

`forEach`: Executa a ação em cada elemento da Stream.

`listen`: Adiciona um subscription a este stream.

Mais em: <https://api.dart.dev/stable/3.0.7/dart-async/Stream-class.html>

```
import 'dart:async';

void main() {
  // Criação de Streams
  final periodicStream = Stream<int>.periodic(const Duration(seconds: 1), (count)
=> count++).take(5);
  final valueStream = Stream.value('Hello, world!');
  final singleStream = Stream<int>.single(42);

  // Consumo de Streams
  periodicStream.listen((value) {
    print('Valor periódico: $value');
  });

  valueStream.listen((value) {
    print('Valor único: $value');
  });

  singleStream.listen((value) {
    print('Valor único: $value');
  });
}
```

Exercício

01 Utilizando Future e async/await

Escreva uma função assíncrona em Dart chamada `fetchData` que simula uma requisição assíncrona a um servidor. A função deve retornar um `Future` que resolve para uma string "Dados obtidos" após um atraso de 3 segundos. Em seguida, escreva um programa principal que chama a função `fetchData` utilizando a palavra-chave `await` e exibe a mensagem de dados obtidos.



Exercício

02 Utilizando Stream e async*

Escreva uma função assíncrona em Dart chamada `countDownStream` que recebe um número máximo como parâmetro. A função deve retornar um `Stream` que emite contagem regressiva de 5 até 1, com um atraso de 1 segundo entre cada emissão. Em seguida, escreva um programa principal que assina o `Stream` retornado pela função `countDownStream` e exibe os números emitidos.



Exercício

03 Utilizando Future.wait e then

Escreva um programa em Dart que usa a função Future.wait para aguardar a conclusão de três tarefas assíncronas diferentes. Cada tarefa deve ser uma função assíncrona que retorna um Future que resolve algum dado após um tempo aleatório. Após aguardar a conclusão das três tarefas, o programa deve exibir os dados obtidos.

OBS: Use a API de referência:

<https://api.dart.dev/stable/3.0.7/dart-async/Future-class.html>



Exercício

04 Utilizando `Stream.value`, `map` e `toList`

Escreva um programa em Dart que usa um `Stream` para processar uma sequência de nomes. O programa deve criar um `Stream` usando `Stream.value` com uma lista de nomes fornecida pelo usuário. Em seguida, deve usar o método `map` para transformar cada nome em seu comprimento (número de caracteres). Por fim, o programa deve usar o método `toList` para coletar os comprimentos dos nomes em uma lista e exibi-la.



Referências

Documentação

Documentação oficial do Dart: Assincronia em Dart:
[Documentação do Dart: https://dart.dev/guides](https://dart.dev/guides)

Documentação Flutter - Async widgets:
<https://docs.flutter.dev/ui/widgets/async>

Livros

"Beginning Flutter: A Hands On Guide To App
Development" por Marco L. Napoli

"Dart for Absolute Beginners" por David Kopec.

Outros

[Curso de Flutter #37 - \[API\] Programação
assíncrona](#)

[Curso de Flutter #38 - \[API\] Geradores
Assíncronos \(async await\).](#)

[Async/Await - Flutter in Focus](#)