



UNITAU
Universidade de Taubaté





Programação Assíncrona



Programação Assíncrona



“A assincronia, na programação de computadores, refere-se à ocorrência de eventos independentes do fluxo principal do programa e às formas de lidar com esses eventos. Estes podem ser eventos "externos", como a chegada de sinais ou ações instigadas por um programa que ocorrem simultaneamente à execução do programa, sem que o programa seja bloqueado para aguardar resultados.”

Programação Assíncrona



A princípio, o fluxo de execução de um programa é síncrono, o fluxo do programa só passa para a próxima operação após a operação atual ser completada. A operação atual bloqueia o fluxo do programa até terminar.

Porém as linguagens atuais implementam mecanismo para a execução assíncrona do programa, permitindo que o fluxo siga para a próxima operação enquanto uma operação demorada é executada. Posteriormente, quando a operação demorada termina, o programa trata o resultado dessa operação. Dessa forma, a operação demorada não bloqueia o fluxo de execução do programa.

Programação Assíncrona



Uma das formas de se implementar programação assíncrona e com o uso de multithread, iniciando uma nova thread para executar a operação demorada. Outra forma é colocar a operação demorada em uma fila de eventos e um gerenciador de eventos executa a operação demorada quando for possível.

Event Loop



Dart é uma linguagem orientada a eventos, o gerenciamento desses eventos é feito pelo **Event Loop**.

Quando uma aplicação Dart ou Flutter inicia, é criada uma nova thread, chamada **UI thread**. Ao criar essa thread, o Dart automaticamente:

1. Inicializa duas filas chamadas **MicroTask** e **Event**
2. Executa o método **main()**
3. Terminada a execução de **main()**, lança o **Event Loop**

MicroTask é uma fila usada para ações internas bem pequenas que precisam ser executadas assincronamente, logo após o término do código que estava sendo executado antes da execução de qualquer evento externo.

Event é uma fila usada para eventos externos (I/O, gesture, timers, streams e outros) e futures.

Event Loop



Durante a vida da aplicação, o processo do Event Loop irá determinar a sequência de execução do código dependendo do conteúdo das filas MicroTask e Event.

O Event Loop é um loop infinito que, quando nenhum outro código Dart está sendo executado, processa as tarefas da fila MicroTask e se não houver nada pendente nessa fila, processa as tarefas da fila Event.

Isolate



Segundo a documentação, Dart é uma linguagem single-thread. Mas na verdade Dart permite a criação de mais threads.

Cada thread da aplicação Dart é chamada de **Isolate** porque cada thread tem suas próprias memória, filas MicroTask e Event e Event Loop.

A comunicação entre os **Isolate** é feita através de mensagens enviadas por uma porta de comunicação.

Isolate



Isolate.dart:

```
import 'dart:io';
import 'dart:async';
import 'dart:isolate';

void main() async {
  stdout.writeln('spawning isolate...');
  ReceivePort receivePort = ReceivePort(); //port for this
  main isolate to receive messages.
  Isolate isolate = await Isolate.spawn(runTimer,
  receivePort.sendPort);
  stdout.writeln('listening...');
  await start(receivePort);
  stdout.writeln('press enter key to quit...');
  await stdin.first;
  stop(isolate);
  stdout.writeln('goodbye!');
  exit(0);
}
```

Isolate



```
Future start(ReceivePort receivePort) async {
  receivePort.listen((data) {
    stdout.writeln('RECEIVE: ' + data);
  });
}

void runTimer(SendPort sendPort) {
  int counter = 0;
  Timer.periodic(new Duration(seconds: 1), (Timer t) {
    counter++;
    String msg = 'notification ' + counter.toString();
    stdout.writeln('SEND: ' + msg);
    sendPort.send(msg);
  });
}

void stop(Isolate isolate) {
  stdout.writeln('killing isolate');
  isolate.kill(priority: Isolate.immediate);
}
```


Isolate



```
> dart isolate.dart  
spawning isolate...  
listening...  
press enter key to quit...  
SEND: notification 1  
RECEIVE: notification 1  
SEND: notification 2  
RECEIVE: notification 2  
SEND: notification 3  
RECEIVE: notification 3
```

```
killing isolate  
goodbye!
```


Future



Um objeto **Future** corresponde a uma tarefa que será executada assincronamente em algum ponto no futuro e retornará um valor.

Quando um novo **Future** é instanciado:

1. Uma instância do **Future** é criada e armazenada em um array gerenciado pelo Dart
2. O código que será executado pelo **Future** é colocado na fila Event
3. A instância do **Future** é retornada com status incompleto
4. O próximo código síncrono (não o código do **Future**) é executado, se houver algum

Future



future.dart:

```
void main() {  
    print('Início do programa');  
    wait10secs();  
    wait5secs();  
    wait3secs();  
    print('Final do programa');  
}
```

```
Future<void> wait10secs() {  
    return Future.delayed(Duration(seconds: 10), () =>  
    print('Depois de 10 segundos'));  
}
```

```
Future<void> wait5secs() {  
    return Future.delayed(Duration(seconds: 5), () =>  
    print('Depois de 5 segundos'));  
}
```


Future



```
Future<void> wait3secs() {  
    return Future.delayed(Duration(seconds: 3), () =>  
    print('Depois de 3 segundos'));  
}
```

```
> dart future.dart  
Inicio do programa  
Final do programa  
Depois de 3 segundos  
Depois de 5 segundos  
Depois de 10 segundos
```

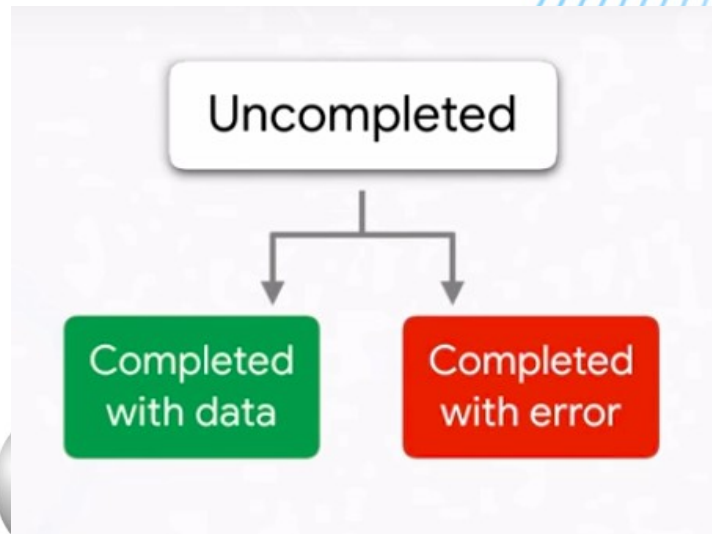

Future



O código referenciado pelo **Future** será executado como qualquer outro evento, assim que o Event Loop retirá-lo da fila.

Quando a execução do **Future** terminar, ele será encerrado com sucesso, executando a cláusula **then()** ou encerrado com erro, executando a cláusula **catchError()**.

Um objeto **Future** está sempre em um de três estados:



Future



then_catch.dart:

```
import 'dart:math';
```

```
void main() {  
  print("Inicio do programa");  
  String result = "Pending";  
  var myFuture = Future.delayed(Duration(seconds: 3), () {  
    Random rand = Random();  
    int res = rand.nextInt(2);  
    print("Randon $res");  
    if (res == 0)  
      throw Exception();  
    else result = "Future completed";  
    return "Result: $result";  
  });  
  myFuture.then((result) {  
    print(result);  
  }).catchError((error){  
    print('Caught $error');  
  });  
  print("Final do programa $result");  
}
```


Future



> dart then_catch.dart

Início do programa

Final do programa Pending

Randon 0

Caught Exception

> dart then_catch.dart

Início do programa

Final do programa Pending

Randon 1

Result: Future completed

Async/Await



Quando um objeto **Future** é criado, o código que criou o **Future** continua a ser executado. O código do **Future** somente será executado posteriormente e não é possível utilizar o resultado do **Future** imediatamente pois o mesmo está no estado incompleto.

Async/Await



await_sem.dart:

```
import 'dart:math';
```

```
void main() {  
  print('Inicio do programa');  
  var x = getRandom();  
  print("Valor ${x}");  
  print('Final do programa');  
}
```

```
Future<int> getRandom() {  
  Random rand = Random();  
  return Future.delayed(Duration(seconds: 2), () {  
    int x = rand.nextInt(10);  
    print("Random ${x}");  
    return x;  
  });  
}
```

Async/Await



```
> dart await_sem.dart
```

```
Início do programa
```

```
Valor Instance of 'Future<int>'
```

```
Final do programa
```

```
Random 1
```


Async/Await



Para que uma função ou método possa criar uma **Future** e aguardar a finalização do **Future** e utilizar o resultado do **Future** na sequência das instruções, é necessário declarar que a função ou método é **async** e a chamada do **Future** deve ser feita com **await**.

Async/Await



await_com.dart:

```
import 'dart:math';
```

```
void main() async {  
  print('Início do programa');  
  var x = await getRandom();  
  print("Valor ${x}");  
  print('Final do programa');  
}
```

```
Future<int> getRandom() {  
  Random rand = Random();  
  return Future.delayed(Duration(seconds: 2), () {  
    int x = rand.nextInt(10);  
    print("Random ${x}");  
    return x;  
  });  
}
```

Async/Await



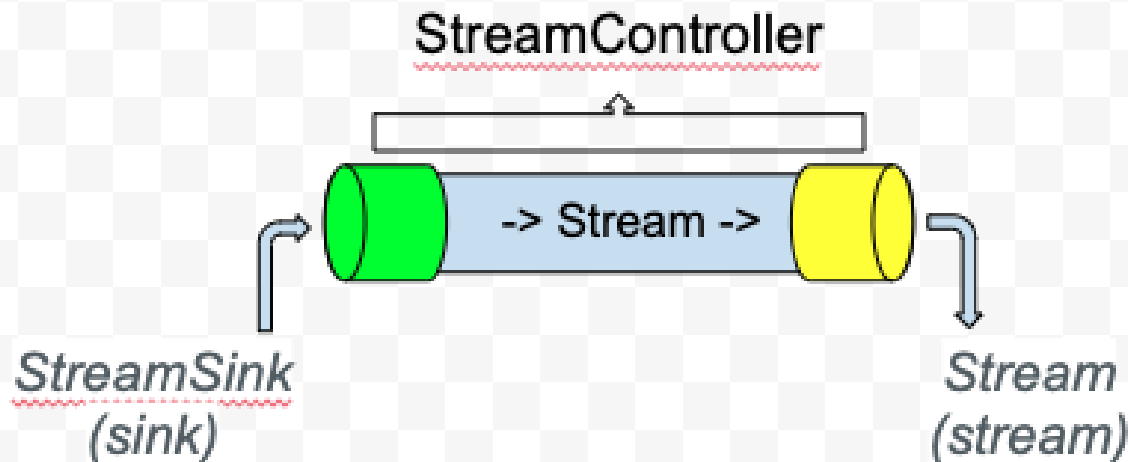
```
> dart await_com.dart  
Início do programa  
Random 7  
Valor 7  
Final do programa
```


Stream



Enquanto **Future** representa uma tarefa assíncrona que retornará um valor no futuro, **Stream** é um fluxo de eventos assíncronos. Cada evento pode retornar um valor (elemento), ou um erro.

Um **Stream** pode ser visto como um pipe onde elementos são inseridos em uma ponta e são transferidos para a outra ponta.



Stream



stream.dart:

```
main() async {  
  Stream<int> stream = createStream(5);  
  await for(int i in stream){  
    print("Recebendo o valor ${i}");  
    print(i);  
  }  
}
```

```
Stream<int> createStream(int max) async* {  
  for (int i = 1; i <= max; i++) {  
    await Future.delayed(const Duration(seconds: 2));  
    print("Gerando o valor ${i}");  
    yield i;  
  }  
}
```

Stream



```
> dart stream.dart  
Gerando o valor 1  
Recebendo o valor 1  
1  
Gerando o valor 2  
Recebendo o valor 2  
2  
Gerando o valor 3  
Recebendo o valor 3  
3  
Gerando o valor 4  
Recebendo o valor 4  
4  
Gerando o valor 5  
Recebendo o valor 5  
5
```


Stream



É possível aplicar várias operações sobre **Stream** como filtros, map, reduce, transformações e outras.

Stream



stream_transformation.dart:

```
import 'dart:async';
```

```
main() async {  
  Stream<int> head = createStream(5).take(2);  
  print("Head");  
  await for(int i in head){ print(i); }  
  Stream<int> tail = createStream(5).skip(3);  
  print("Tail");  
  await for(int i in tail){ print(i); }  
  Stream<int> even = createStream(5)  
    .where((value) => value%2==0);  
  print("Even");  
  await for(int i in even){ print(i); }  
  StreamTransformer<int,int> doubleTransformer =  
    StreamTransformer.fromHandlers(handleData: handleData);  
  Stream<int> dobro = createStream(5)  
    .transform(doubleTransformer);  
  print("Double");  
  await for(int i in dobro){ print(i); }  
}
```

Stream



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

```
void handleData(data, EventSink sink) {  
    sink.add(data*2);  
}
```


Stream



> dart stream_transformation.dart

Head

1

2

Tail

4

5

Even

2

4

Double

2

4

6

8

10

Stream



Para utilizar um **Stream** é necessário ouvir (**listen**) o **Stream**, também chamado de inscrever (**subscribe**) o **Stream**. Quando se faz o registro para ouvir um **Stream** com **listen()**, é criado um objeto **StreamSubscription**. Não é obrigatório se atribuir esse objeto a uma variável se não for necessário utilizá-lo diretamente.

Quando ocorre algum evento (geração de um dado, erro ou fechamento do **Stream**), o **Stream** notifica os objetos **StreamSubscription** que estão ouvindo o **Stream**.

Stream



stream_listen.dart:

```
import 'dart:io';  
import 'dart:async';
```

```
main() {  
  print("Inicio do programa");  
  Stream<String> stream = new Stream.fromFuture(getData());  
  StreamSubscription<String> subscription =  
    stream.listen((data) {  
      print("DataReceived: "+data); },  
      onDone: () { print("Task Done"); },  
      onError: (error) { print("Some Error"); }  
    );  
  print("Final do programa");  
}
```

```
Future<String> getData() async {  
  var file = File('/etc/os-release');  
  var contents = await file.readAsString();  
  print("Fetched Data");  
  return contents;  
}
```


Stream



> dart stream_listen.dart

Início do programa

Final do programa

Fetches Data

DataReceived: NAME="openSUSE Tumbleweed"

VERSION="20200224"

ID="opensuse-tumbleweed"

ID_LIKE="opensuse suse"

VERSION_ID="20200224"

PRETTY_NAME="openSUSE Tumbleweed"

ANSI_COLOR="0;32"

CPE_NAME="cpe:/o:opensuse:tumbleweed:20200224"

BUG_REPORT_URL="https://bugs.opensuse.org"

HOME_URL="https://www.opensuse.org/"

LOGO="distributor-logo"

Task Done

Stream



Existem dois tipos de Stream:

- **Single Subscription** – um Stream que aceita apenas um listner, uma única subscrição.
- **Broadcast** – um Stream que aceita um número infinito de listners, varias subscrições.

Se não for declarada que o Stream é um broadcast, o Stream só aceitará uma subscrição.

Stream



```
stream_broadcast.dart:  
import 'dart:async';  
  
main() async {  
  Duration interval = Duration(seconds: 2);  
  Stream<int> stream = Stream<int>.periodic(interval,  
callback)  
    .asBroadcastStream();  
  stream.listen((data) {  
    print("Data: ${data}");  
  });  
  stream.listen((data) {  
    if (data%2==0)  
      print("Even");  
    else  
      print("Odd");  
  });  
  stream.listen((data) {  
    print("Double: ${data*2}");  
    print('');  
  });  
}
```


Stream



```
int callback(int value) => value;
```

```
> dart stream_broadcast.dart
```

```
Data: 0
```

```
Even
```

```
Double: 0
```

```
Data: 1
```

```
Odd
```

```
Double: 2
```

```
Data: 2
```

```
Even
```

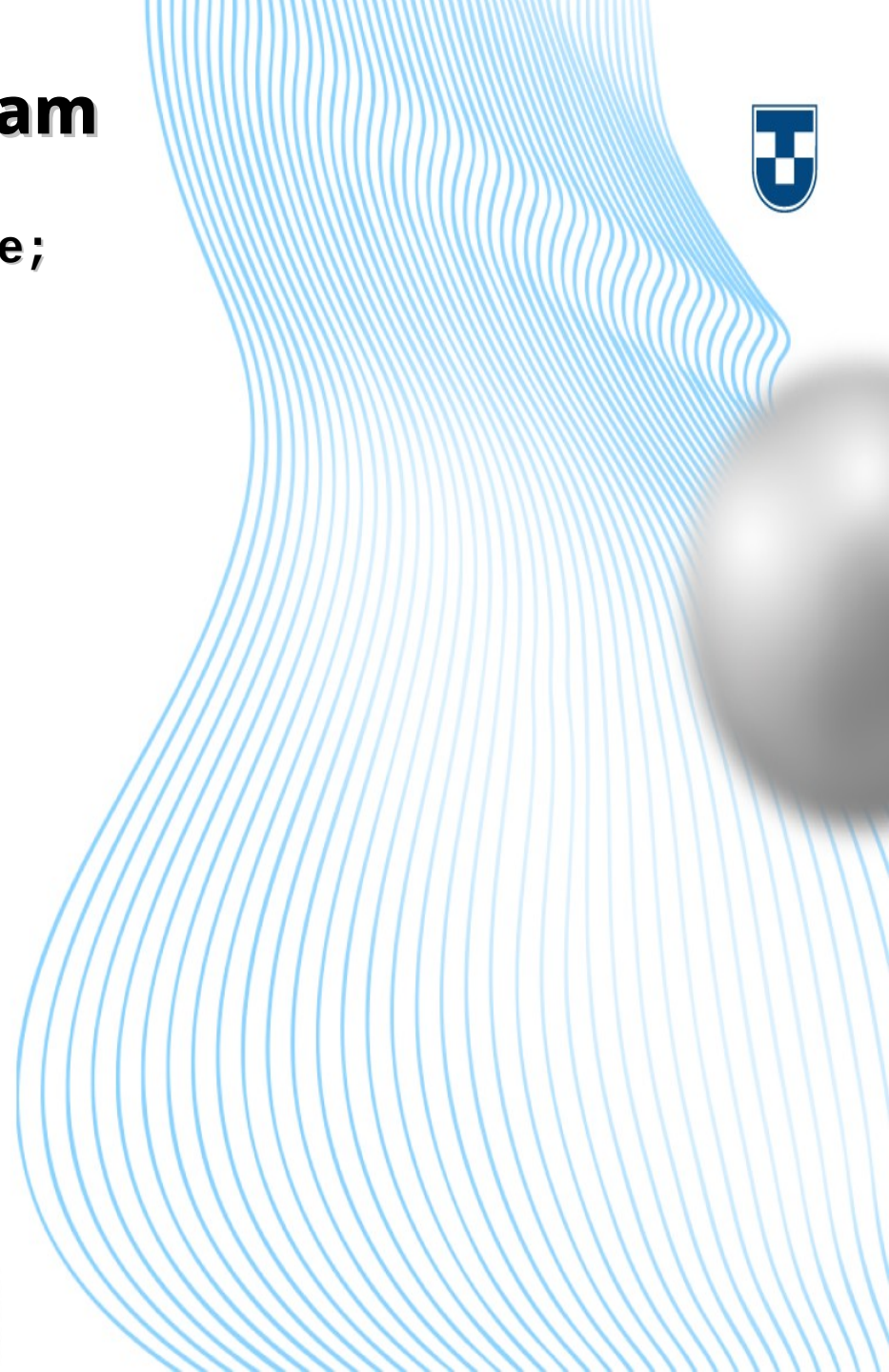
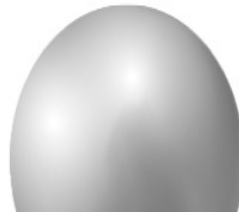
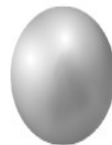
```
Double: 4
```

```
Data: 3
```

```
Odd
```

```
Double: 6
```

```
^C
```



Stream



O objeto **StreamController** permite gerenciar o **Stream**. Com o **StreamController** é possível incluir elementos ou erros e fechar o **Stream**.

Stream



stream_controller.dart:

```
import 'dart:async';
```

```
main() {  
  print("Inicio do programa");  
  StreamController streamController = StreamController();  
  streamController.stream.listen((data) {  
    print("DataReceived: " + data); },  
    onDone: () {  
      print("Task Done"); },  
    onError: (error) {  
      print("Some Error"); }  
  );  
  
  streamController.add("This a test data");  
  streamController.addError(new Exception('An exception'));  
  streamController.add("This a test data 2");  
  streamController.add("This a test data 3");  
  streamController.close();  
  print("Final do programa");  
}
```


Stream



```
> dart stream_controller.dart:  
Inicio do programa  
Final do programa  
DataReceived: This a test data  
Some Error  
DataReceived: This a test data 2  
DataReceived: This a test data 3  
Task Done
```

Programação Reativa



“Na computação, a programação reativa é um paradigma de programação declarativa relacionado aos fluxos de dados e à propagação da mudança. Com esse paradigma, é possível expressar fluxos de dados estáticos (por exemplo, matrizes) ou dinâmicos (por exemplo, emissores de eventos) com facilidade e também comunicar que existe uma dependência inferida no modelo de execução associado, o que facilita a propagação automática dos fluxo de dados alterados.”

Atualmente a programação reativa é muito incentivado para programação com Flutter, principalmente como uma alternativa para lidar com um grande fluxo de dados.

Programação Reativa



“A programação reativa é a programação com fluxos de dados assíncronos.

Em outras palavras, tudo, desde um evento (por exemplo, toque), alterações em uma variável, mensagens, ... para criar solicitações, tudo o que pode mudar ou acontecer será transmitido, acionado por um fluxo de dados.

Programação Reativa



Isso significa que, com a programação reativa, o aplicativo:

- **torna-se assíncrono,**
- **é arquitetado em torno da noção de Streams e listeners,**
- **quando algo acontece em algum lugar (um evento, uma alteração de uma variável ...) uma notificação é enviada para um Stream,**
- **se "alguém" ouvir esse Stream, ele será notificado e tomará as ações apropriadas, qualquer que seja sua localização no aplicativo."**