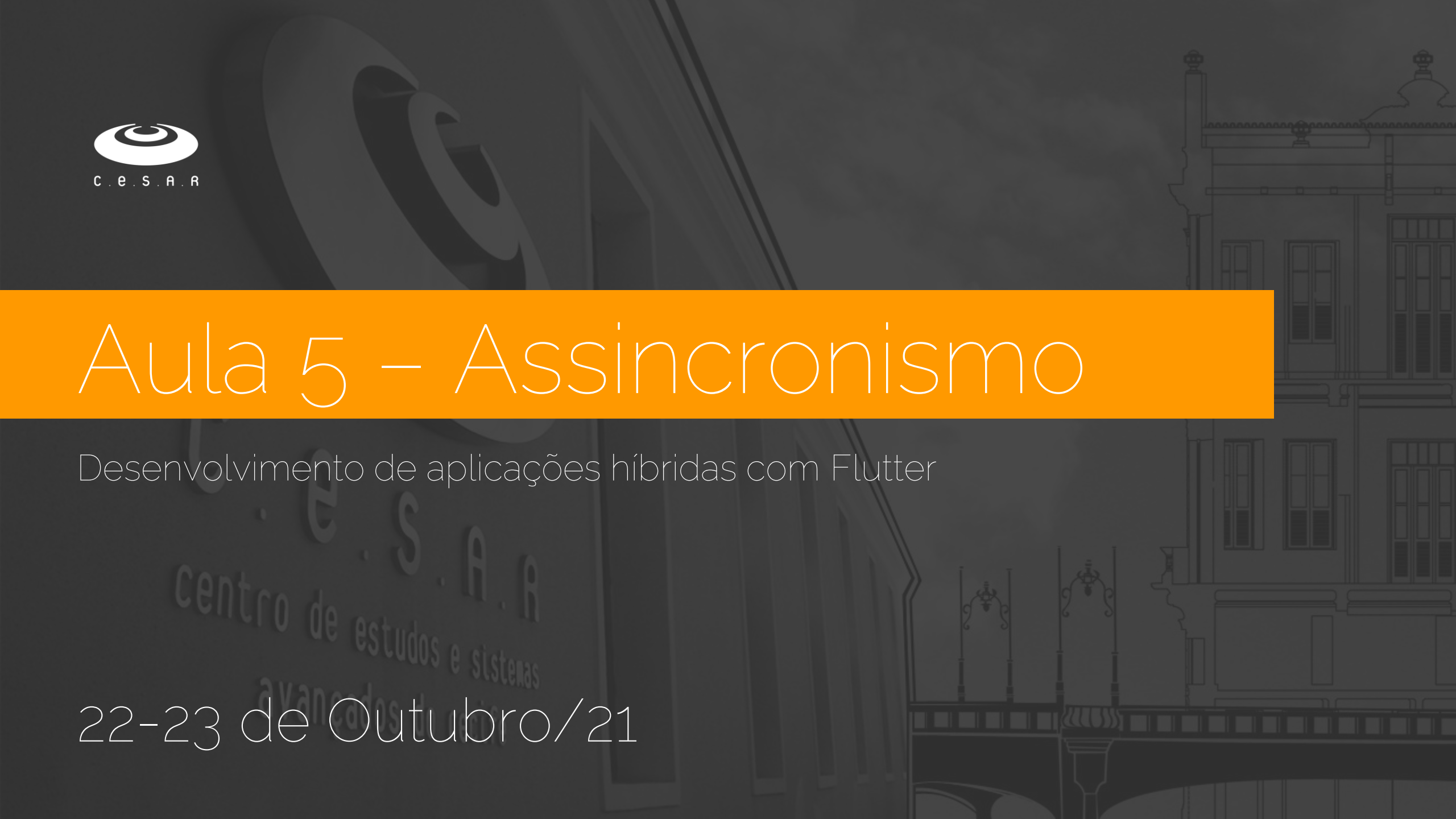


Aula 5 – Assincronismo

Desenvolvimento de aplicações híbridas com Flutter

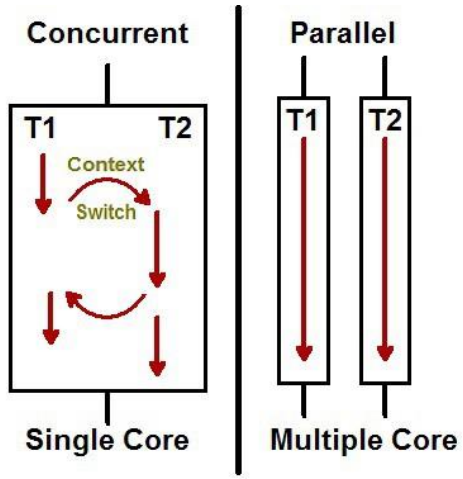
22-23 de Outubro/21

Concorrência x Paralelismo

Assincronismo

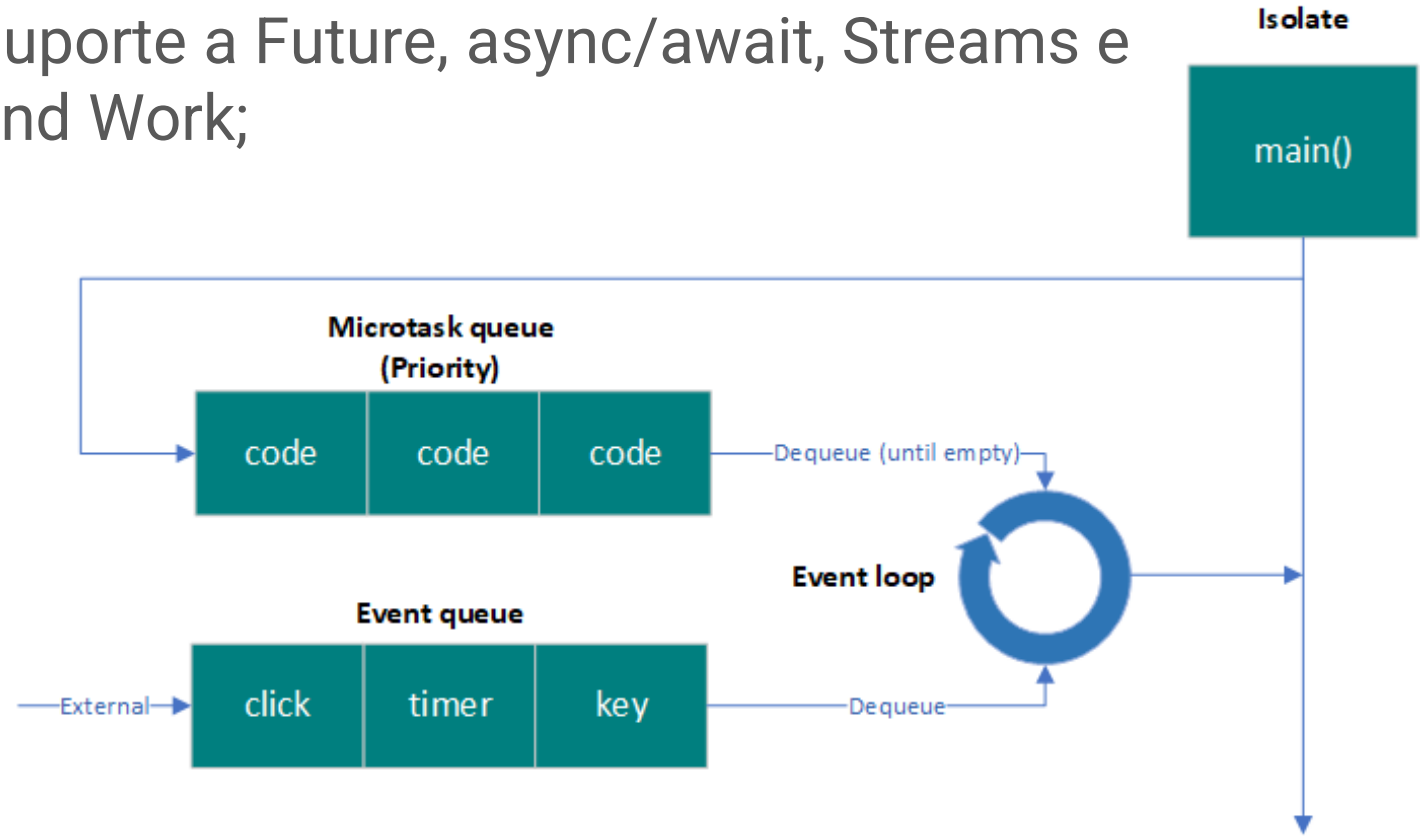
Concorrência x Paralelismo

- Na concorrência, os processos disputam por CPU, e o Sistema operacional alterna rapidamente entre eles para dar a ilusão de paralelismo.
- No paralelismo, processos executam em Cores separados em paralelo



Assincronismo

- Dart é uma linguagem Single Thread que executa em um Event Loop - Isolate Main;
- Oferece suporte a Future, async/await, Streams e Background Work;



Assincronismo

```
void main() {
     print("Main ${Isolate.current.debugName}, $pid");
     runApp(const MyApp());
}
```

```
@override
Widget build(BuildContext context) {
  print("App.build() ${Isolate.current.debugName}, $pid");
```

13:50:02.991 24733 24806 I flutter : Main main, 24733
13:50:03.307 24733 24806 I flutter : My.App.build() main, 24733

Process Id: ***24733***

Main Thread: ***24733***

Flutter Main: ***24806***

Isolate

- Isolate é uma thread e ele não compartilha memória como as threads convencionais;
- Ele possui o seu event loop;
- A interações entre diferentes isolates é feita via Messages;
- Cada Isolate tem torno de ~2MB;
- São bem leves mas use com cuidado;
- Cada mensagem passada requer "message size * 2" de memória;

Isolate

```
entryFunction(input) {
  print("log 2($input): ${Isolate.current.debugName}, $pid");
}


print("log 1: ${Isolate.current.debugName}, $pid");
Isolate.spawn(entryFunction, 'hello').then(
  (...) { print("Then: ${Isolate.current.debugName}, $pid"); }
); // or compute(entryFunction, 'hello')
```

```
14:14:19.994 24733 24806 I flutter : log 1: main, 24733
14:14:20.342 24733 30414 I flutter : log 2(hello): entryPointWith2Args, 24733
14:14:24.624 24733 24806 I flutter : Then: main, 24733
```

Future

Future é uma classe que representa o resultado de uma operação assíncrona (concorrência e não paralelismo), podendo ser:

- Uncompleted;
- Completed:
  - With value;
  - With error;

```
Future<String> getName() {
    return Future.delayed(const Duration(milliseconds: 4000), () {
        return "name";
    });
}
```

Future

Obtendo resultado com Callbacks

Podemos encadear Callbacks para quando a Future for completa.

```
getName()
    .then((value) => print(value))
    .catchError(() => print("Error"))
    .whenComplete(() => print("Completed fetching name"));
```

Future

Obtendo resultado com Async-Await

Com as Keywords Async e Await podemos tornar o código mais legível, e fácil de entender.

```
try {
    final name = await getName();
    print(name);
} catch (error) {
    print("Error");
} finally {
    print("Completed fetching name")
}
```

Stream

Declarando e emitindo uma Stream

Stream é uma sequência de eventos assíncronos, que notificam quando existe um evento novo.

```
Stream<String> readFile() async* {
  for(int i = 0; i < 7; i++) {
    await Future.delayed(const Duration(seconds: 1));
    yield "Line " + i.toString();
  }
}
```

Stream

Escutando com listen()

```
stream.listen(
    (data) {
      print('Data: $data');
    },
    onError: (err) {
      print('Error: $err');
    },
    cancelOnError: false,
    onDone: () {
      print('Done');
    }
  );
```

Stream

Escutando com await-for

Com a sintaxe await-for, podemos escutar por eventos ocorrendo múltiplas vezes.

```
await for (final line in readFile()) {
    print(line);
}
```

Stream

Mais informações

https://dart.dev/codelabs/async-await
https://dart.dev/tutorials/language/streams
https://www.youtube.com/watch?v=5AxWC49ZMzs&t=186s