



SI700 – Programação para Dispositivos Móveis

Aula 7 – Processos Concorrentes

Prof. Ulisses Martins Dias

2022

Faculdade de Tecnologia – Unicamp

Table

DataTable

Concorrência

Isolates

- Se você tem que mostrar vários dados aos usuários em um aplicativo, então terá que organizá-los de alguma forma para que a visualização seja agradável na tela pequena de um celular.
- A forma mais simples de visualizar os dados é preenchendo uma planilha. Nesse contexto, os widgets **Table** e **DataTable** são úteis e funcionam de modo parecido com o que você faria para web.
- Para que o tempo de aula seja melhor utilizado, não falaremos sobre este widget em sala de aula, mas um material introdutório existe aqui neste grupo de slides para você ver.

Table

- Você irá simplesmente definir as bordas da tabela (ou manter o padrão sem bordas) para depois adicionar as linhas usando **TableRow**. Dentro das linhas, você pode colocar qualquer widget que você quiser.

Table

```
Table(  
  border: TableBorder(  
    top: BorderSide(width: 2),  
    bottom: BorderSide(width: 2),  
    left: BorderSide(width: 2),  
    right: BorderSide(width: 2)),  
  children: [  
    TableRow(  
      decoration: BoxDecoration(  
        borderRadius: BorderRadius.circular(15),  
        border: Border.all(  
          color: Colors.black,  
          width: 1.0,  
        )),  
      children: [  

```

DataTable

- Um outro padrão que você já deve ter visto várias e várias vezes. Um **DataTable** requer que você informe quais são as colunas com um **DataColumn** e que depois comece a informar as linhas, uma após a outra, com **DataRow**.
- Você pode adicionar o parâmetro **sortColumnIndex** para indicar por qual coluna a tabela está ordenada. Note que isso é só um indicador, não passando de perfumaria, você é quem deve garantir que os dados estão realmente ordenados.

Table

```
DataTable(sortColumnIndex: 1, columns: [
    DataColumn(label: Text("Nome")),
    DataColumn(label: Text("Sobrenome"))
],
rows: [
    DataRow(cells: [
        DataCell(Text("Ulisses")),
        DataCell(Text("Dias"))
    ])
    ,
    DataRow(
        cells: [
            DataCell(Text("Guilheme")),
            DataCell(Text("Coelho"))
        ]
    )
])
```

Concorrência

- Em programas Android nativos, existem pelo menos três tipos de threads: **RenderThread**, **MainThread** e **OutrasThreads**.
- A **MainThread** é aquela que gerencia os cliques dos botões, a interação com a tela, as chamadas do ciclo de vida, e assim por diante. Em geral, não colocamos nada muito pesado na **MainThread** sob o risco de congelar a interface gráfica.
- Nesse caso, em programação para android nativo, usamos threads em paralelo para operações longas, para operações que precisam executar em paralelo (como músicas, por exemplo), para operações que usam rede (internet, download, ...), operações em arquivos e operações em bancos de dados.

- Em android nativo, fazemos o uso das classe **Thread**, **AsyncTask**, **Service**, ou da interface **Runnable**.
- Em flutter, todos os códigos executarão na **MainThread**. Usando a nomenclatura da linguagem, tudo executa em um **Isolate** e os comandos dentro desse **isolate** são executados de maneira sequencial.
- Nesse caso, um código muito lento pode ter o efeito de bloquear a interface gráfica, piorando a experiência do usuário. Assim como podemos fazer para que isso não aconteça?

- Uma solução em alguns casos gira em torno das seguintes palavrinhas: **Future**, **Await** e **Async**. A resposta do flutter consiste em adicionar à sintaxe essas palavras reservadas que indicam que algumas operações não deverão bloquear a **MainThread**.
- Não daremos uma explicação extensa e completa sobre o tema, apenas o suficiente para prosseguirmos com a disciplina.
- **Future**: um tipo que permite “**prometer**” a entrega de um objeto no futuro. Quando invocamos uma função que retorna Future, não podemos associar o valor prometido imediatamente, porque o valor é só uma promessa. Se quisermos imprimir o valor, e não a promessa, usamos **await**.

```
operacaoLonga() {  
    Future<String> result = Future.delayed(  
        Duration(seconds: 5), () {  
            return "Mensagem aguardada";  
        })  
    );  
    return result;  
}
```

- **Await**: indica que queremos aguardar que uma promessa se concretize antes de prosseguir com o código. Nesse caso, indica que queremos pagar o preço de uma operação longa para ter o objeto, e não uma promessa apenas.
- A função onde a palavra reservada **await** está ficará bloqueada aguardando a concretização da promessa. Entretanto, não é sensato que ela bloqueie toda a **MainThread**. Nesse caso, deverá ser colocada em paralelo.

```
var aux = await operacaoLonga();
```

- **Async:** a palavra reservada “async” deverá ser usada para informar que aquela função (notadamente a que usa o await) não irá bloquear a MainThread, o código que invocou a função com async irá prosseguir sem aguardar o término dela.

```
esperaOperacaoLonga() async {  
    var aux = await operacaoLonga();  
    print("O valor aguardado é: $aux");  
}
```


- Outra forma de esperar a concretização de um promessa consiste em utilizar a API “then” do objeto Future. Esse método permite que passemos uma função por parâmetro que será executada quando Future se concretizar no objeto que está destinado a ser.

```
var aux = operacaoLonga();  
aux.then((conc) {  
    print("Valor: $conc");  
});
```

Isolates

- Por padrão, todo código em Dart usa um **Isolate**, uma seção de código que tem o seu próprio bloco de memória e a sua própria sequência de processamento.
- Todo código dentro de um isolate, síncrono ou assíncrono, irão executar um de cada vez.
- Mesmo que coloquemos um código um código de maneira assíncrona usando **Future**, ele irá bloquear todo o **isolate** se fizer um uso intenso de CPU.
- Uma solução para o problema seria fazer o lançamento de outro **isolate**, o que gera outra linha de execução, e fazer o processamento pesado lá.

- Algumas noções sobre lançamentos de novos **isolates** precisam ser entendidas.
 - O pai de um Isolate não pode usar a memória dos filhos, dado que um Isolate gerencia a sua própria memória.
 - Difere então de Java e C++ que permite compartilhamento de memória.
 - Esse mecanismo é uma solução que parece pouco poderosa (isso é só aparência, dado que trocas de mensagens podem suprir a necessidade de compartilhamento), mas diminui a complexidade dos códigos.

- É possível ter mais de um isolate, mas eles não irão compartilhar memória entre si. Na verdade, os isolates só podem se comunicar uns com os outros por meio de troca de mensagem.
- Por mais que você possa ter vários isolates, não significa que você precisa ter mais de um.
- Como sugestão, use apenas um Isolate, a não ser que precisa realmente de um novo.

- A primeira forma de criar um Isolate é usando `Isolate.spawn()`. A segunda forma é usando `compute()`.
- Você comunica entre isolates usando `sendPort` e `ReceivePort`. Fora essa forma, eles são isolados, daí o nome `isolate`.

```
Isolate.spawn(highLevelFunction, parameters);
```

```
compute(highLevelFunction, parameters);
```