

# Javascript

Programa de atração de formação de talentos  
Bradesco / Visionaire

**Aula 10 – Programação assíncrona**  
**Parte 1 – Callbacks e Promises**

Mark Joselli  
[mark.Joselli@pucpr.br](mailto:mark.Joselli@pucpr.br)



# Código assíncrono

O JavaScript não suporta execução em paralelo (multithread), mas suporta a programação assíncrona. Para isso utiliza três conceitos:

- **Callbacks**: Funções que são chamadas quando uma computação terminou. Por exemplo, a função `onClick` do botão, ou o callback da classe `Timer`;
- **Promises**: Representam a promessa de que uma computação irá ocorrer.
- **async / await**: Comandos que simplificam o uso dos Promises



# Callbacks

- Uma função A pode ser passada para uma função B, contendo um código a ser executado quando B terminar.
- Chamamos essa função A de *callback*
- Embora não seja multi-tarefa, o javascript pode disparar serviços que executam em paralelo, por exemplo, ao requisitar um dado em outro servidor ou ao consultar o banco de dados
- O código continua sendo executado e, quando o serviço termina, ele aciona um callback para que se lide com o resultado



# Callbacks

- Vamos exemplificar criando um Timer. Existem 2 comandos para criar timers:
  - **setTimeout**(callback, tempo): Para um evento que ocorre uma única vez, após x milissegundos
  - **setInterval**(callback, tempo): Para um evento que ocorre a cada x milissegundos


Ambas retornam um identificador que pode ser usado nas funções **clearTimeout** e **clearInterval** para cancelar o timer.

# Executando um callback

- Execute o código a seguir:

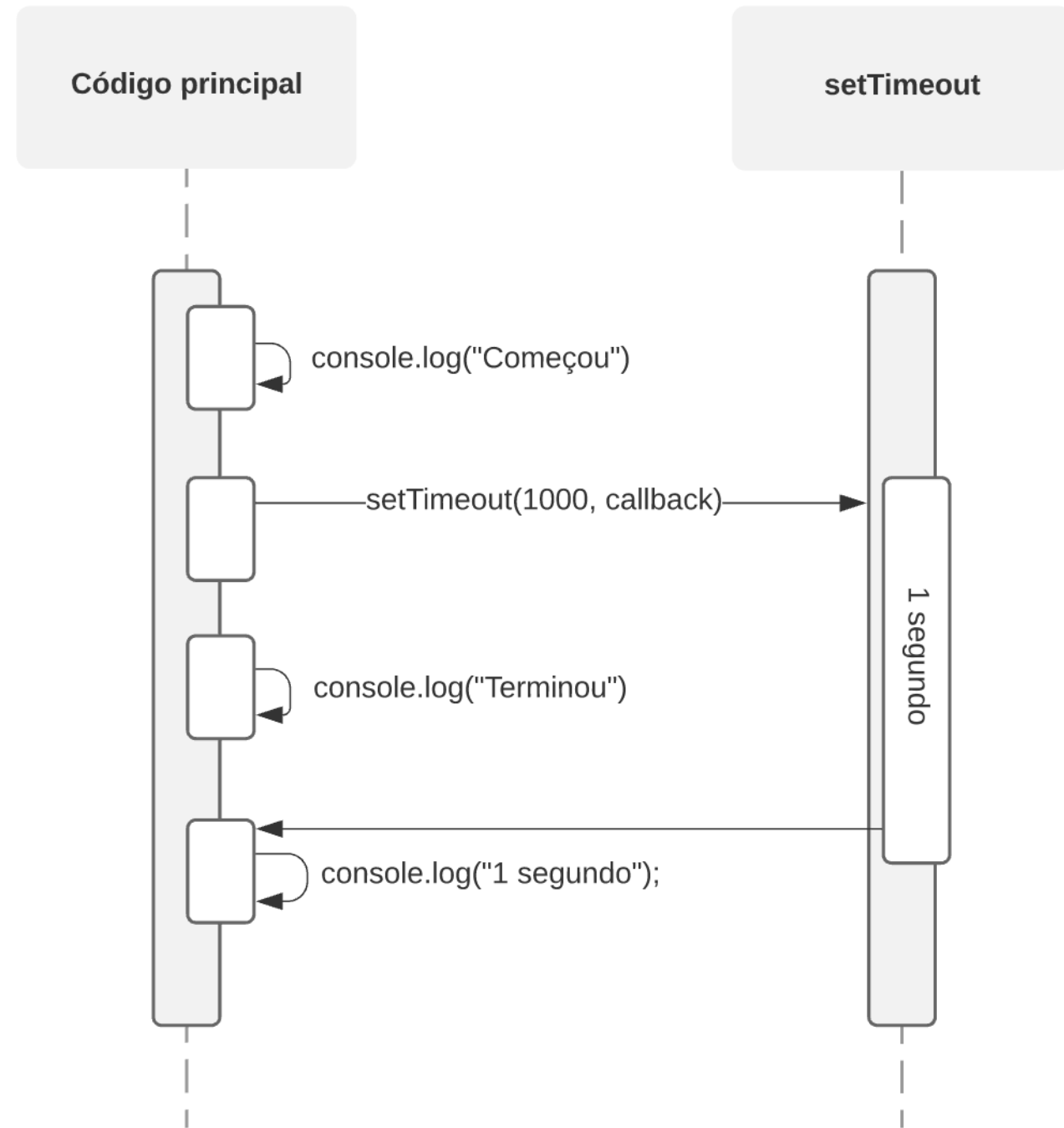
```
console.log("Contando...");  
setTimeout(function() {  
    console.log("1 segundo passou...");  
}, 1000);  
console.log("Terminou!");
```

Esta função é  
o callback



# Callbacks

Note que o código prossegue sua execução enquanto aguarda o timer



## Exemplo 2: XMLHttpRequest

- O objeto `XMLHttpRequest` pode ser utilizado para realizar requests em outras páginas. Para isso:
  - Chama-se a função `open` com o verbo e url desejada
  - Adiciona-se ao objeto os callbacks `onload` e `error`, que serão acionadas ao final da requisição
  - Chama-se a função `send()` para realizar a requisição
- Ao final da requisição o objeto `XMLHttpRequest` terá definido o código de status http, a resposta (response) e a mensagem.
- Observe uma requisição de sucesso pode retornar um código de status de falha.

## Exemplo 2: XMLHttpRequest

```
const req = new XMLHttpRequest();
req.open('GET', 'https://jsonplaceholder.typicode.com/posts');
req.onload = () => {
    if (req.status >= 200 && req.status <= 299) { //OK
        console.log(req.response);
    } else {
        console.log(`${req.status}: ${req.message}`);
    }
};

req.onerror = () => { console.log("Network error"); }
req.send();
```



# Convertendo dados

- Podemos converter objetos JavaScript em JSON com o comando `JSON.stringify(obj)`
- O interessante é que também podemos fazer o contrário. Uma String contendo JSON pode ser convertida em um objeto Javascript do comando `JSON.parse(texto)`

# Promises

- Esses problemas foram resolvidos com a criação do conceito de **Promises**;
- Trata-se de um objeto, representando a computação que está ocorrendo em paralelo
- Toda função interessada em executar código assíncrono deve retorna-lo
- É no objeto Promise retornado que os callbacks poderão ser registrados



# Criando um Promise

Como exemplo, vamos criar uma função que envie um valor a outra função após x milissegundos. A abordagem somente com callbacks seria assim:

```
function emitirValor(valor, timeout, callback) {  
    setTimeout(() => callback(valor), timeout);  
}
```

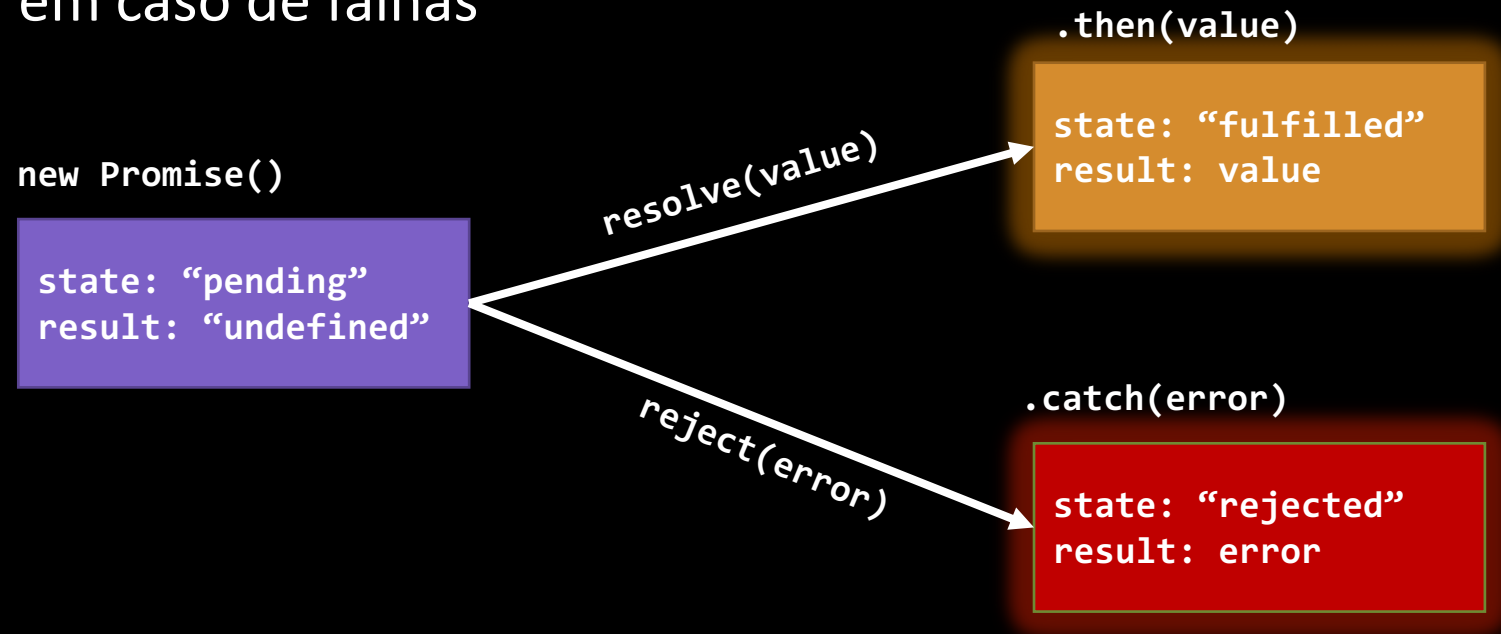
Agora, iremos altera-la para retornar o objeto Promise

# Promises

O construtor do Promise recebe uma função com dois argumentos chamados `resolve` e `reject`. Trata-se de duas funções que devem ser chamadas quando a computação assíncrona terminar:

**resolve**: Chamada em caso de sucesso

**reject**: Chamada em caso de falhas



# Criando um Promise

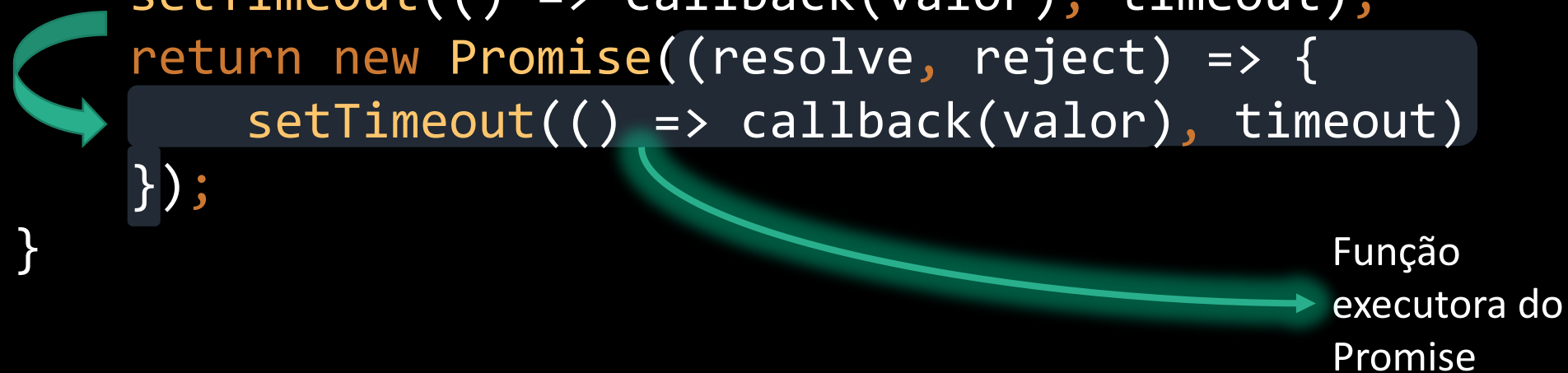
Nossa função agora retornará o Promise.

```
function emitirValor(valor, timeout, callback) {  
  setTimeout(() => callback(valor), timeout);  
  return new Promise();  
}
```

# Criando um Promise

Vamos agora incluir a função executora. Ela deve encapsular o processamento assíncrono, isto é, o `setTimeout`

```
function emitirValor(valor, timeout, callback) {  
  setTimeout(() => callback(valor), timeout);  
  return new Promise((resolve, reject) => {  
    setTimeout(() => callback(valor), timeout)  
  });  
}
```



Função executora do Promise



# Criando um Promise

Note que o callback será registrado no Promise. Isso significa que ele não precisa mais ser passado para a função. O callback do Promise será chamado sempre que as funções **resolve** ou o **reject** forem utilizadas.

```
function emitirValor(valor, timeout, callback) {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => resolve(valor), timeout)  
    });  
}
```

# Criando um Promise

Como toque final, vamos adicionar o timeout padrão de 1 segundo (1000). Assim, a função `emitirValor` ficará desse jeito:

```
function emitirValor(valor, timeout = 1000) {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => resolve(valor), timeout)  
    });  
}
```

# Utilizando a função assíncrona

- É assim que podemos utiliza-la:


```
const promise = emitirValor(50);  
promise.then(valor => {  
    console.log(`O valor ${valor} foi emitido.`);  
});
```

Também poderíamos utilizar o objeto promise diretamente:

```
emitirValor(50).then(valor => {  
    console.log(`O valor ${valor} foi emitido.`);  
});
```

# Promises

Os métodos *then*, *catch* e *finally* **sempre** retornam outro Promise. Esse promise representa a promessa de toda a computação até ali.




```
emitirValor(50)
  .then(valor => {
    const num = parseFloat(valor);
    if (isNaN(num)) {
      throw `Não numérico: ${valor}`;
    }
    return valor * 50;
  })
  .then(produto => emitirValor(produto, 2000))
  .then(console.log)
  .catch(error => console.log(`ERRO: ${error}`));
```

# Promises

Caso o then retorne um valor este valor será colocado em um Promise, para que possa ser usado no próximo then


```
emitirValor(50)
  .then(valor => {
    const num = parseFloat(valor);
    if (isNaN(num)) {
      throw `Não numérico: ${valor}`;
    }
    return valor * 50;
  })
  .then(produto => emitirValor(produto, 2000))
  .then(console.log)
  .catch(error => console.log(`ERRO: ${error}`));
```



# Promises

Se ele retornar um Promise, este será usado diretamente como retorno do then. Assim, podemos chama-lo por fora da função

```
emitirValor(50)
  .then(valor => {
    const num = parseFloat(valor);
    if (isNaN(num)) {
      throw `Não numérico: ${valor}`;
    }
    return valor * 50;
  })
  .then(produto => emitirValor(produto, 2000))
  .then(console.log)
  .catch(error => console.log(`ERRO: ${error}`));
```






# Promises

Caso qualquer then lance erro, ou um promise chame a função reject, o código é desviado para a função catch.

```
emitirValor(50)
  .then(valor => {
    const num = parseFloat(valor);
    if (isNaN(num)) {
      throw `Não numérico: ${valor}`;
    }
    return valor * 50;
  })
  .then(produto => emitirValor(produto, 2000))
  .then(console.log)
  .catch(error => console.log(`ERRO: ${error}`));
```

A green curved arrow originates from the 'throw' statement in the first .then block and points to the .catch block, illustrating the error handling flow in a Promise chain.

# Promises

- Uma Promise *guarda* o resultado da computação na memória.
  - Assim, chamar a função *then* num promise que já tenha concluído **não disparará** uma nova requisição.

```
const promise = emitirValor(50);  
//Roda após 1s  
promise.then(v => console.log(`Valor emitido: ${v}`));  
...  
//Executa imediatamente  
promise.then(v => console.log(`Valor emitido: ${v}`));
```

# Atividades

- Vamos estudar os problemas dessa abordagem na prática?
  1. Crie um timer que imprima o texto “PUCPR” a cada meio segundo. Pare o timer após 5 execuções.
  2. Crie um timer que mostre no console a palavra “Pontifícia Universidade”. Ele deve disparar um segundo timer, que mostre no console o texto “Católica do”. Repita para um terceiro timer que mostre o texto “Paraná”.
  3. Crie 3 timers A (0.5s), B (0.2s) e C (0.8s). Os timers devem produzir os valores  $a=5$ ,  $b=10$  e  $c=2$ , respectivamente. Ao final dos 3 timers, deve ser calculada a expressão:  $a + b * c$ . O programa deve continuar funcionando mesmo se alterarmos os tempos dos timers entre uma execução e outra.

# Atividades

4. Escreva uma função **testNum** que receba um número como um argumento e retorne um **Promise** que resolve em caso o número seja maior que dez, ou rejeite caso contrário. Use-o exibindo uma mensagem em cada caso.
5. Escreva duas funções puras que retornem **Promises**:
  - A primeira, **makeAllCaps()**, receberá um array de palavras e tornará todas maiúsculas. Ela rejeitará caso o array contenha um dado que não seja string.
  - A segunda, **sortWords()**, ordenará as palavras em ordem alfabética.
  - Em seguida, teste-as
6. Crie a função **request**(url, verbo= 'GET' ) que utilize o XMLHttpRequest na forma de um Promise.
  - Use-a para fazer o mesmo get que fizemos no exemplo 2. Não esqueça de tratar possíveis erros com o .catch.

# Dúvidas?

- Materiais:
  - Livro Eloquent Javascript:
    - 3ª edição (inglês):  
<https://eloquentjavascript.net/>
    - 2ª edição (português):  
<https://github.com/braziljs/eloquente-javascript>
  - The Modern Javascript Tutorial:  
<https://javascript.info/>