5954024 - Introdução ao Desenvolvimento Web

JavaScript - Teoria e Prática - Parte 3

Prof. Dr. Denis M. L. Martins
DCM | FFCLRP | USP

Objetivos de Aprendizagem

- Entender o conceito de execução **síncrona** vs **assíncrona**
- Entender o funcionamento do Event Loop
- Identificar problemas de bloqueio
- Trabalhar com Callbacks, Promises e async/await

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Dica

Use o JSFiddle para os exemplos da aula.

Introdução ao JavaScript Assíncrono

- JavaScript é **sincrono por padrão**: executa comando após comando.
  - Operações lentas (ex: carregar imagens, API) podem travar a página.
  - Exemplo: Carregar uma imagem por 2s $\longrightarrow$ usuário vê "página em branco" por 2s.
- **Assíncrono**: "não bloquem a execução" enquanto esperamos algo acontecer.
  - Foco na **melhoria** de experiência do usuário.
  - Delega tarefas ao ambiente (browser / Node).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Execução síncrona

Como se comporta a call stack neste exemplo?

```
function task(message) {
    // simular tarefa demorada
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
task('Call an API');
console.log('Done!');
```

> Exemplo adaptado de javascripttutorial.net

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Execução Assíncrona

```
console.log("1");
console.log("2");
setTimeout(() => console.log("3"), 2000);
console.log("4");
```

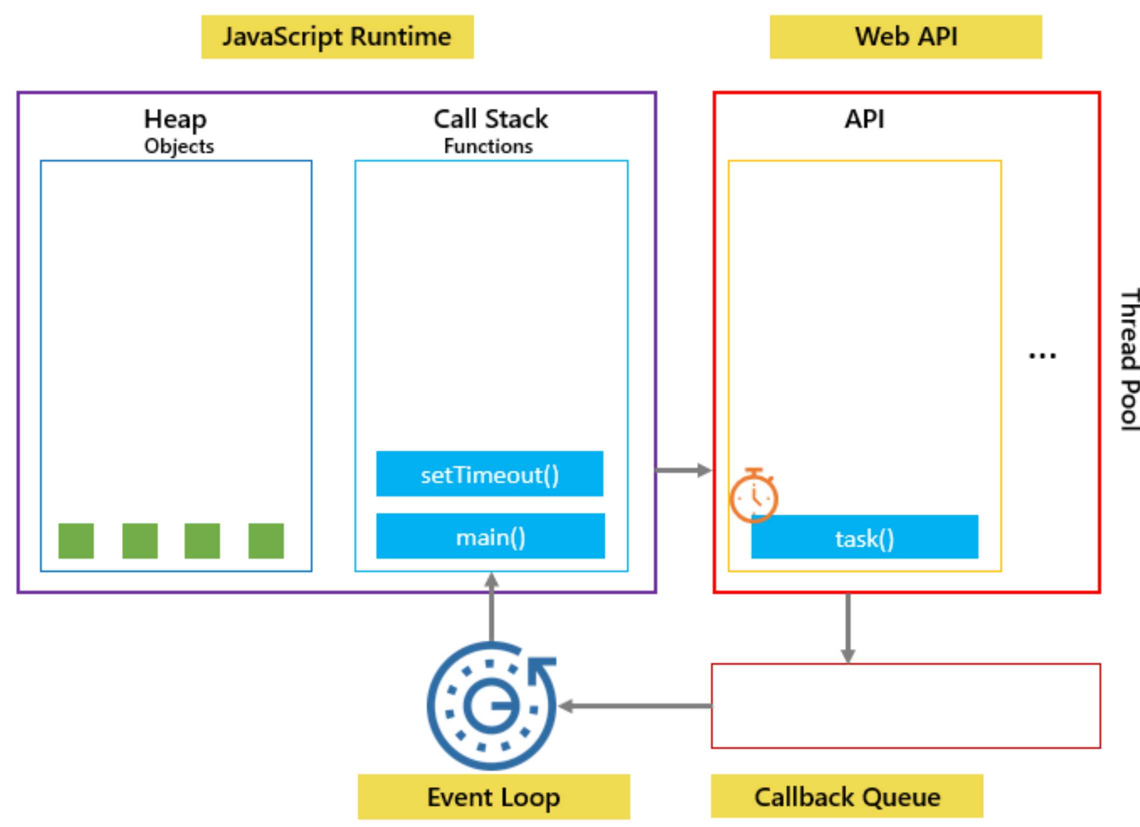
- **Saída**: “1”, “2”, “4”, “3” — o navegador continua funcionando enquanto espera.
- **Por que isso acontece?**

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Execução Assíncrona

- JavaScript usa Event Loop.
- Tarefas assíncronas vão para uma fila.
- **setTimeout(fn, tempo)** é usado para agendar execução futura.

> Fonte da imagem ao lado: JavaScript Tutorial



Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Outro Exemplo Assíncrono

```
function getData(callback) {
    fetch("https://api.restful-api.dev/objects/1")
        .then(response => response.json())
        .then(data => callback(data))
        .catch(error => console.error("Error:", error));
}

function handle(data) {
    console.log("Fetched Data:", data);
}

getData(handle);
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Callbacks

Callback é uma função passada como parâmetro.

É executada quando uma operação termina.

```
function sayHello(name, callback) {
    console.log("Hello, " + name);
    callback();
}

function sayBye() {
    console.log("Goodbye!");
}

sayHello("World", sayBye);
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Callback Assíncrona

É possível "agendar" a execução da callback para um momento posterior.

```
function fazerCafe(tempo, callback) {
    console.log("Fazendo o café")
    setTimeout(() => {
        callback("Café tá pronto!");
    }, tempo);
    console.log("Ainda fazendo o café...")
}

fazerCafe(2000, function (msg) {
    console.log(msg);
});
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Event Loop

JavaScript é single-threaded (uma tarefa por vez). Mas consegue lidar com tarefas assíncronas usando o Event Loop.

1. Código entra na Call Stack
2. Operações assíncronas vão para Web APIs
3. Quando terminam → vão para a fila
4. O Event Loop verifica:
   i. Se a stack está vazia
   ii. Se sim → envia próxima tarefa da fila para execução

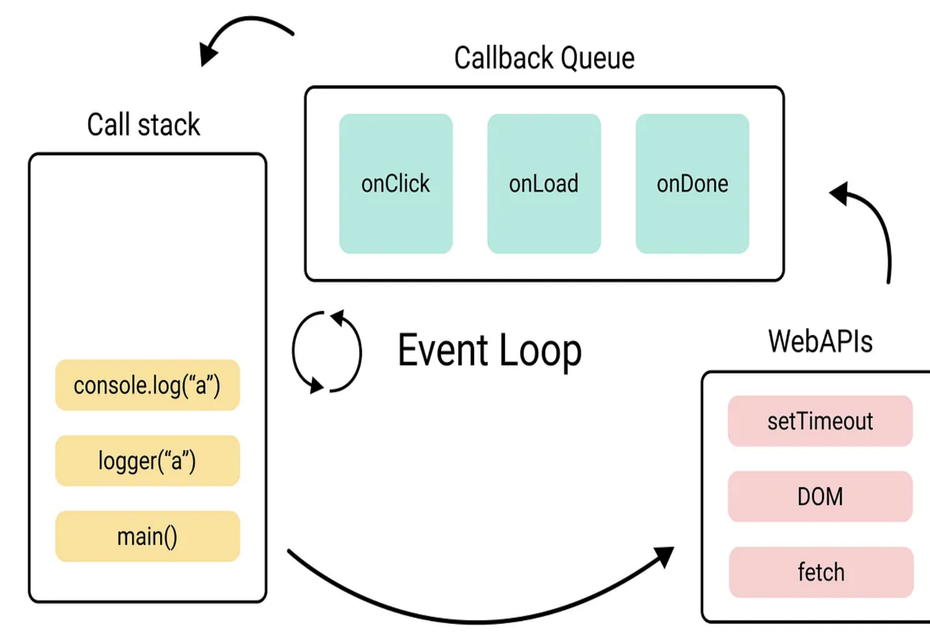


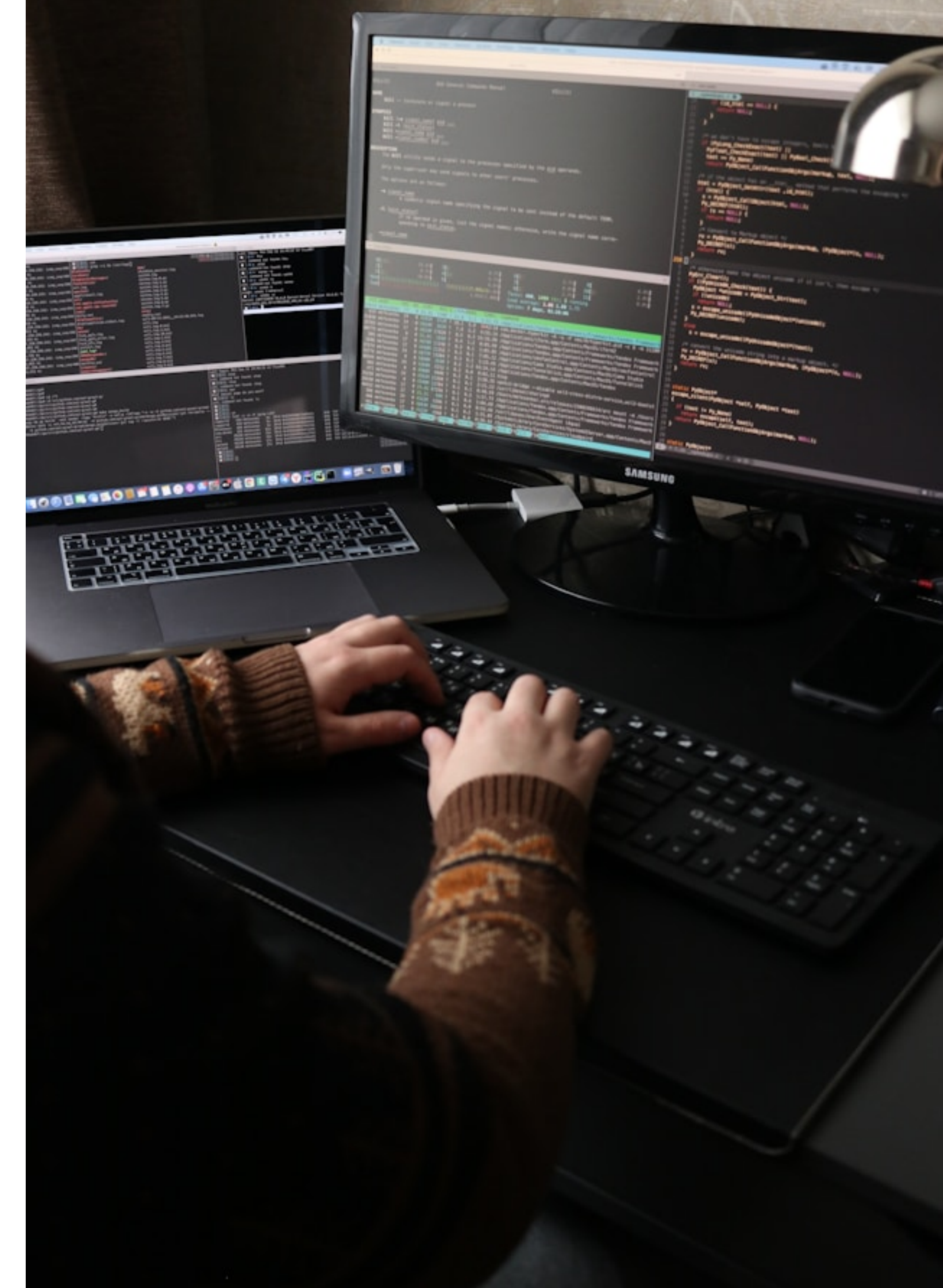
Fig. .1: Event loop. Fonte: Vigen Hovhannisyan @Medium.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício `setTimeout`

Crie um código que:

1. Exiba "Início"
2. Após 2 segundos, exiba "Fim"



Callback hell

Callbacks aninhadas ficam difícil de ler. Isso é chamado de callback hell.

```
setTimeout(() => {
    console.log("1");
    setTimeout(() => {
        console.log("2");
        setTimeout(() => console.log("3"), 1000);
    }, 1000);
}, 1000);
```

Solução: Promises + Async/Await.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Promises

- Uma **Promise** representa um valor que será resolvido ou rejeitado no futuro.
- Estados: **pending** (não resolvido), **fulfilled** (resolvido), **rejected** (rejeitado).
- Usamos **.then()** para tratar sucesso e **.catch()** para erro.

```
const minhaPromise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Concluído!"), 2000);
});

minhaPromise
    .then((msg) => console.log(msg))
    .catch((err) => console.log(err));
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exemplo

```
function buscarDados() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const sucesso = true; // simula sucesso ou erro
            if (sucesso) {
                resolve("Dados carregados com sucesso!");
            } else {
                reject("Erro ao buscar dados.");
            }
        }, 2000);

    });
}
buscarDados()
    .then(resultado => {
        console.log(resultado);
    })
    .catch(erro => {
        console.log(erro);
    });
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Função `fetch` (Requisições HTTP no JavaScript)

- `fetch` é uma função **assíncrona** usada para buscar dados de uma API
- Faz requisições HTTP (GET, POST, etc.)
- Retorna uma `Promise`

```
fetch("https://api.restful-api.dev/objects/1")
    .then(response => response.json())
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.error("Erro:", error);
    });
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Async/Await: A sintaxe mais simples

Adiciona **async** antes da função → torna ela uma promise.

Adiciona **await** antes de uma promise → aguarda o resultado.

Fácil de ler, sem usar **.then()**.

```
async function exibirMensagem() {
    const msg = await new Promise(resolve => {
        setTimeout(() => resolve("Olá, Mundo!"), 2000);
    });
    console.log(msg);
}

exibirMensagem();
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Async/Await com fetch()

Usamos **await** para obter o resultado.

A resposta é um **Response**, que pode ser convertido em **JSON**.

```
async function carregarDados() {
    try {
        const response = await fetch("https://api.restful-api.dev/objects/1");
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Erro:", error);
    }
}

carregarDados();
```

> Use **fetch()** + **async/await** para APIs modernas.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Tratando Erros com `.catch()`

Promises podem falhar → usamos `.catch()` para tratar.

`await` + `try/catch` é o jeito mais simples.

```
async function buscarErro() {
    try {
        const res = await fetch("https://api.com/erro");
        const data = await res.json();
        console.log(data);
    } catch (error) {
        console.log("Erro:", error.message);
    }
}
buscarErro();
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Async/Await sem bloqueio para múltiplas chamadas

O navegador não fica bloqueado: ele faz as 3 chamadas em paralelo.

```
async function chamarTudo() {
    const res1 = await fetch("https://api.restful-api.dev/objects/1");
    const res2 = await fetch("https://api.restful-api.dev/objects/2");
    const res3 = await fetch("https://api.restful-api.dev/objects/3");

    console.log(await res1.json());
    console.log(await res2.json());
    console.log(await res3.json());
}

chamarTudo();
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício

Desenvolva uma página em HTML e JavaScript que:

- faça uma requisição assíncrona para a API https://api.restful-api.dev/objects
- receba os dados com **fetch**
- utilize **async/await**
- manipule o DOM para exibir os dados na página

Você deve criar uma página com: título, botão chamado "Carregar Produtos" e uma área de resultados.

Ao clicar no botão, o programa deverá fazer uma requisição para a API, aguardar o retorno da API com async/await, converter a resposta para JSON, exibir os resultados em uma lista.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Conclusão e Próximos Passos

Conceitos-Chave:

- JavaScript é *single-threaded*, mas suporta **assincronicidade**.
- **Event Loop** gerencia a execução de tarefas.
- Callbacks $\longrightarrow$ Promises $\longrightarrow$ async/await (evolução do padrão).
- `fetch` permite comunicação com APIs externas.
- Prefira async/await para melhor legibilidade
  Próxima aula: Framework Python Django.

Material adicional:

- Leitura do capítulo 11 do livro Elegant JavaScript, por Marijn Haverbeke.
- Vídeo no YouTube: Asynchronous JavaScript in ~10 Minutes.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br