Por: Alice Dantas

Funções em JS

Função Tradicional

Ela é declarada usando a palavra-chave *function*. Pode ser chamada **antes ou depois** de sua declaração (hoisting). Nessa função, a palavra "return" é opcional quando a função só possui uma única linha de comando e não precisa retornar nenhum valor.

```
function soma(a, b) {
    return a + b;
};
console.log(soma(2, 3)); // 5
```

Função Anônima

É uma função que **não tem nome** e geralmente é atribuída a uma variável. Não sofre hoisting, então só pode ser chamada **após** a linha em que foi definida.

```
const multiplicar = function(a, b) {
    return a * b;
};
console.log(multiplicar(4, 5)); // 20
```

Arrow Function

Usa `=>` ao invés de *function*.

```
const soma = (a, b) => {
    return a + b;
};
console.log(soma(2, 3)); // 5

// Se tiver apenas um parâmetro, os parênteses podem ser omitidos:
const quadrado = x => x * x;
console.log(quadrado(4)); // 16

// Se a função só tem uma instrução, o return é implícito, logo não usa {}, nem return:
const dobro = n => n * 2;
console.log(dobro(7)); // 14
```

Por: Alice Dantas

Função Imediatamente Invocada

É uma função que é **executada imediatamente** após ser definida.

```
(function(a, b) {
    console.log(a + b);
})(3, 4); // 7
```

Funções com Parâmetros Padrão

Você pode definir valores padrões para parâmetros caso não sejam passados.

```
function saudacao(nome = "visitante") {
    return `Olá, ${nome}!`;
};

console.log(saudacao("Alice")); // Olá, Alice!
console.log(saudacao());        // Olá, visitante!
```

Funções Rest (...args)

Se você não sabe quantos argumentos serão passados, pode usar **rest parameters**. ...args transforma todos os argumentos em um array.

```
function somaTudo(...numeros) {
    return numeros.reduce((total, num) => total + num, 0);
};

console.log(somaTudo(1, 2, 3, 4)); // 10
```

Funções de *temporização*

Executam algo depois de um atraso de milissegundos.

- setTimeout

Executa uma função após atraso, sem travar a execução das próximas linhas.
Sintaxe: `setTimeout(função, tempo);` Exemplo:

```
console.log("Início");

setTimeout(() => {
    console.log("Depois de 2 segundos");
}, 2000);

console.log("Fim");

// Início
// Fim
// Depois de 2 segundos
```

- setInterval

Executa uma função **repetidamente** a cada intervalo de tempo.
Sintaxe: `setInterval(função, tempo);` Exemplo:

```
setInterval(() => {
    console.log("Isso aparece a cada 2 segundos");
}, 2000);
```

- clearTimeout e clearInterval

Cancelam a execução agendada pelo `setTimeout` ou `setInterval`.

Exemplo setTimeout:

```
const id = setTimeout(() => {

    console.log("Essa mensagem NÃO vai aparecer.");
}, 3000);

clearTimeout(id);
console.log("Timeout cancelado!"); // única saída: Timeout cancelado!
```

Exemplo clearInterval:

```
let contador = 0;
```

```
const idDoIntervalo = setInterval(() => {
    contador++;
    console.log("Contagem:", contador);

    if (contador === 5) {
        clearInterval(idDoIntervalo);
        console.log("Intervalo parado!");
    }
}, 1000);
```

A saída será:

```
Contagem: 1
Contagem: 2
Contagem: 3
Contagem: 4
Contagem: 5
Intervalo parado!
```

Funções com Callback

Uma função pode receber outra função como parâmetro.

```
// declarando uma função callback
function cumprimentar(callback) {
    const nome = "Alice";
    callback(nome);
};

// chamando a função, passando uma função anônima como argumento
cumprimentar(function(nome) {
    console.log(`Olá, ${nome}!`);
});

// chamando a função de novo, mas passando uma arrow function agora
cumprimentar(nome => console.log(`Oi, ${nome}!`));
```

Funções de array que recebem callbacks

Por: Alice Dantas

Métodos de array permitem que você passe **uma função como argumento**.

- **forEach**: executa uma função para cada elemento da lista.

```
[1, 2, 3].forEach(item => console.log(item * 2));
```

- **map**: cria um novo array aplicando uma função a cada elemento.

```
const quadrados = [1, 2, 3].map(x => x * x);
console.log(quadrados); // [1, 4, 9]
```

- **filter**: cria um novo array com os elementos que passam no teste da função.

```
const pares = [1, 2, 3, 4].filter(x => x % 2 === 0);
console.log(pares); // [2, 4]
```

- **reduce**: reduz o array a um único valor, acumulando os resultados da função (soma, média, objeto, etc.).

```
const soma = [1, 2, 3].reduce((acumulador, valoratual) => acumulador + valoratual, 0);
console.log(soma); // 6
```