Lógica de Programação - JavaScript

Aula 04

Rafael Neri

Na aula anterior....

Funções

Funções são blocos de código que executam uma tarefa específica e podem ser reutilizadas ao longo do programa.

A **assinatura** de uma função representa seu nome, parâmetros e (quando aplicável) valores retornados.

Exemplo

```
function saudacao() {
  console.log("Bem-vindo à Farmácia Boa Saúde!");
}
```

Função com parâmetros e retorno

```
function calcularDesconto(valor, percentual) {
  return valor - valor * (percentual / 100);
}

const resultado = calcularDesconto(100, 10);

console.log(resultado); // 90
```

Parametrização de Funções

Parametrizar funções significa torná-las **genéricas**, permitindo reutilizar o mesmo código em situações diferentes.

```
function elevar(base, expoente) {
  return base ** expoente;
}

const resultado1 = elevar(2, 3);
console.log(resultado1); // 8

const resultado2 = elevar(5, 2);
console.log(resultado2); // 25
```

Tratamento de Erros

Rafael Neri

Tratamento de Erros

Programas podem falhar por diversos motivos: entradas inválidas, falhas de rede, valores inesperados...
Para evitar que o programa quebre, utilizamos mecanismos de tratamento de erros.

`try / catch / finally`

- **try**: execute aqui o código que pode lançar um erro (apenas o necessário).
- **catch (captura erro)**: executa se um erro for lançado dentro do `try`;
- **finally**: bloco opcional que ao ser definido sempre será executado — útil para liberar recursos, fechar conexões ou fazer limpeza, independentemente de erro.

Observação:

- Erros em chamadas assíncronas ou Promises não são capturados por um `try` (use `.catch()` ou `async/await` + `try/catch`).

Exemplo com erro

- Código sem tratamento de erro: abrir arquivo inexistente causa exceção.

```
const fs = require('fs');
const conteudo = fs.readFileSync('arquivo.txt', 'utf8');
console.log('Conteúdo lido:', conteudo);
```

Exemplo com tratamento de erro

```
const fs = require('fs');

try {
  const conteudo = fs.readFileSync('arquivo.txt', 'utf8');
  console.log('Conteúdo lido:', conteudo);
} catch (err) {
  console.error('Arquivo não encontrado...');
  console.error(err.message);
}
```

Exemplo do uso do `finally`

- Exemplo que dispara um erro real: tentativa de parse de JSON inválido.

```
try {
  const dados = JSON.parse('isto não é um JSON válido');
  console.log('JSON parseado:', dados);
} catch (err) {
  console.error('JSON inválido');
} finally {
  console.log('Fechar conexão com banco de dados');
}
```

Dica: `finally` é ideal para liberar recursos (fechar conexões, limpar estados), pois sempre executa, com ou sem erro.

O objeto `err`

- Quando um erro é capturado no `catch`, o parâmetro (comumente `err` ou `e`) é um objeto que descreve o problema.
- **Propriedades úteis**:
  - `err.name` — nome do erro (ex.: `TypeError`, `ReferenceError`, `SyntaxError`).
  - `err.message` — mensagem legível explicando o erro.
  - `err.stack` — pilha de chamadas (stack trace) útil para depuração (onde o erro ocorreu).
  - Em Node.js, erros também podem ter `err.code` (ex.: `'ENOENT'`) para identificar tipos específicos.

Boas práticas ao tratar `err`:

- Logue `err.message` para mostrar a causa ao usuário e `err.stack` ao depurar.
- Não confunda `err.message` com `err` — o objeto pode ter mais informações.

```
try {
  // código que pode falhar
} catch (err) {
  console.error('Erro:', err.message); // mensagem amigável
  console.debug(err.stack); // stack para depuração
  if (err.code === 'ENOENT') {
    // tratamento específico para arquivo não encontrado (Node.js)
  } else {
    // rethrow quando não souber tratar
    throw err;
  }
}
```

Disparando erros

- Use `throw` quando detectar condições que impedem a função de continuar normalmente.
- Prefira lançar instâncias de `Error` (ex.: `new Error()`, `new TypeError()`), com mensagens claras.

```
// Exemplo: validar entradas e lançar erro
function dividir(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new TypeError('Parâmetros devem ser números');
  }
  if (b === 0) {
    throw new Error('Divisão por zero');
  }
  return a / b;
}
```

```
try {
  console.log(dividir(10, 0));
} catch (err) {
  console.error('Erro capturado:', err.name, '-', err.message);
}
```

- Boas práticas:
  - Lance `Error` com mensagens específicas e legíveis para quem depura.
  - Use tipos de erro (`TypeError`, `RangeError`) quando fizer sentido.
  - Evite lançar valores primitivos (string/number) — prefira `Error`.
  - Valide entradas cedo (fail fast) e documente quais erros sua função pode lançar.

Tipos de erro em JS

- **JavaScript possui várias classes de erro nativas. Conhecê-las ajuda a depurar e tratar corretamente.**
- **Principais tipos:**
  - `SyntaxError` — erro de sintaxe.
  - `ReferenceError` — referência a variável não declarada.
  - `TypeError` — operação inválida para o tipo.
  - `RangeError` — valor fora do intervalo aceitável.
  - `URIError` — erro em funções de URI.
  - `EvalError` — relacionado ao `eval()`.

Como identificar e tratar

- Use `e.name` ou `instanceof` para distinguir tipos:

```
try {
  // código
} catch (e) {
  if (e instanceof TypeError) {
    // trata TypeError
  } else if (e.name === 'SyntaxError') {
    // trata SyntaxError
  } else {
    throw e; // repropaga erros desconhecidos
  }
}
```

- Regra prática: trate apenas os erros que você sabe como recuperar; registre e re-lance os demais.

Map

O que é `Map`?

- `Map` é uma coleção de pares chave→valor introduzida no ES6.
- Ao contrário de `Object`, as chaves em um `Map` podem ser qualquer valor (strings, números, objetos, funções).

Criando e usando um `Map`

```
const m = new Map();

const chave = 'nome';
const valor = 'Ana';

m.set(chave, valor);

if (m.has(chave)) { // true
    console.log(m.get(chave)); // Ana
}

console.log(m.size); // 2

m.delete('nome');
```

Métodos úteis

- `set(key, value)` — adiciona ou atualiza
- `get(key)` — retorna valor ou `undefined`
- `has(key)` — verifica existência
- `delete(key)` — remove um elemento
- `clear()` — remove todos
- `size` — número de pares

Iterando sobre um `Map`

```
const m2 = new Map([['x', 10], ['y', 20]]);

for (const [k, v] of m2) console.log(k, v);
// ou
for (const k of m2.keys()) console.log(k);
for (const v of m2.values()) console.log(v);

m2.forEach((v, k) => console.log(k, v));
```

Map vs Object — quando usar

- Use `Object` para estruturas simples, especialmente quando precisa serializar para JSON (chaves são strings).
- Use `Map` quando:
  - precisar de chaves não-string (objetos, funções);
  - desejar garantias de ordem de inserção;
  - houver muitas operações de inserção/remoção (performance melhor em alguns casos).

Conversões entre Map / Array / Object

```
// Map -> Array
const arr = Array.from(m2.entries()); // [['x',10], ['y',20]]

// Array -> Map
const m3 = new Map(arr);

// Object -> Map (quando chaves forem strings)
const obj = { a: 1, b: 2 };
const mapFromObj = new Map(Object.entries(obj));

// Map -> Object (só se as chaves forem strings)
function mapToObject(map) {
  const o = {};
  for (const [k, v] of map) o[k] = v;
  return o;
}
```

Exemplos práticos

- Contador de ocorrências:

```
const words = ['a','b','a','c','b','a'];
const counts = new Map();
for (const w of words) counts.set(w, (counts.get(w) || 0) + 1);
// Map {'a' => 3, 'b' => 2, 'c' => 1}
```

- Cache onde a chave é um objeto:

```
const cache = new Map();
function compute(obj) {
  if (cache.has(obj)) return cache.get(obj);
  const res = /* trabalho pesado */ JSON.stringify(obj);
  cache.set(obj, res);
  return res;
}
```

Dicas e armadilhas

- `WeakMap` aceita apenas objetos como chave e não impede garbage collection — útil para metadados ligados a objetos.
- Ao converter `Map` → `Object`, verifique se as chaves são strings; caso contrário, perca informação ou cause coerção indesejada.
- `Map` preserva ordem de inserção — isso pode ser útil para apresentar dados previsíveis.