

JavaScript com Alana



Alana Neo

JavaScript com Alana

Uma abordagem objetiva

Alana Neo

Índice

Bem Vindo	7
1 Introdução	8
1.1 Como estudar este livro	8
1.2 O que você vai construir como estudante	8
1.3 Pré-requisitos	8
1.4 Preparando o ambiente	9
1.4.1 1) Criar uma pasta de estudos	9
1.4.2 2) Teste no navegador	9
1.4.3 3) Teste no Node.js	9
1.5 Dica de rotina semanal	10
1.6 Erros comuns de quem está começando	10
1.7 Objetivo final	10
2 O que é JavaScript?	11
2.1 Por que aprender JavaScript?	11
2.2 Onde o JavaScript roda?	11
2.3 Seu primeiro programa	11
2.4 Rodando no navegador	11
2.5 Rodando com Node.js	12
2.6 JavaScript no dia a dia de um estudante	12
2.7 Comentários no código	13
2.8 Boas práticas iniciais	13
2.9 Exercícios	13
3 Variáveis e Tipos	14
3.1 <code>let</code> , <code>const</code> e <code>var</code>	14
3.2 Regras de nome de variáveis	14
3.3 Tipos primitivos	14
3.4 Conversão de tipos	15
3.5 <code>NaN</code> e validação de número	15
3.6 Template strings	16
3.7 Exemplo prático: orçamento de passeio	16
3.8 Exemplo prático: situação do estudante	16
3.9 Exercícios	16

4	Operadores e Estruturas de Controle	17
4.1	Operadores aritméticos	17
4.2	Operadores relacionais	17
4.3	Operadores lógicos	17
4.4	if, else if, else	18
4.5	Exemplo escolar: status do aluno	18
4.6	switch	18
4.7	Operador ternário	19
4.8	Laço for	19
4.9	Laço while	19
4.10	break e continue	20
4.11	Exemplo prático: orçamento até limite	20
4.12	Exercícios	20
5	Funções	21
5.1	Por que funções são importantes?	21
5.2	Declaração de função	21
5.3	Parâmetros e retorno	21
5.3.1	Exemplo do cotidiano estudantil	22
5.4	Parâmetros com valor padrão	22
5.5	Função anônima em variável	22
5.6	Arrow functions	22
5.7	Escopo	23
5.8	Funções puras e efeitos colaterais	23
5.9	Função que calcula x função que exhibe	24
5.10	Callback (função passada para outra função)	24
5.11	Exemplo integrado: roteiro de viagem	24
5.12	Boas práticas para funções	25
5.13	Exercícios	25
6	Arrays e Objetos	26
6.1	Arrays	26
6.2	Métodos mais usados em arrays	26
6.3	Percorrendo arrays	27
6.3.1	for tradicional	27
6.3.2	for...of	27
6.4	map, filter, reduce	27
6.5	Objetos	28
6.6	Acessando e alterando propriedades	28
6.7	Array de objetos	28
6.8	Desestruturação	29
6.9	Spread em arrays e objetos	29
6.10	Exercícios	29

7	Vetores, Matrizes, Fila e Pilha	30
7.1	Vetores (arrays unidimensionais)	30
7.1.1	Exemplo 1: notas de uma turma	30
7.1.2	Exemplo 2: soma e média	30
7.1.3	Exemplo 3: filtro de valores	30
7.2	Matrizes (arrays bidimensionais)	31
7.2.1	Exemplo 1: acesso por linha e coluna	31
7.2.2	Exemplo 2: percorrer todos os elementos	31
7.2.3	Exemplo 3: diagonal principal	32
7.3	Fila (FIFO: First In, First Out)	32
7.3.1	Exemplo 1: atendimento	32
7.3.2	Exemplo 2: próximo da fila	32
7.3.3	Exemplo 3: fila com validação	33
7.4	Pilha (LIFO: Last In, First Out)	33
7.4.1	Exemplo 1: empilhar e desempilhar	33
7.4.2	Exemplo 2: topo sem remover	33
7.4.3	Exemplo 3: desfazer ação	34
7.5	Aplicação prática combinando as estruturas	34
7.6	Exercícios	35
8	Funções, Blocos, Procedimentos, Módulos e Parâmetros	36
8.1	Blocos	36
8.2	Procedimentos	36
8.3	Funções com retorno	37
8.4	Passagem de parâmetros	37
8.4.1	Primitivo: cópia por valor	37
8.4.2	Objeto: referência compartilhada	37
8.4.3	Evitando mutação com spread	38
8.5	Modularização	38
8.5.1	Export nomeado	38
8.5.2	Múltiplos exports	39
8.5.3	Export default	39
8.6	Estrutura sugerida para projetos pequenos	39
8.7	Exemplo integrador	40
8.8	Exercícios	40
9	POO e Módulos	41
9.1	Conceitos básicos de POO	41
9.2	Criando uma classe	41
9.3	Encapsulando regras com métodos	41
9.4	Herança	42
9.5	Polimorfismo (ideia prática)	43

9.6	Relembrando módulos ES	43
9.6.1	Export nomeado	43
9.6.2	Export default	44
9.7	Exemplo integrado: classe + módulo	44
9.8	Exercícios	45
10	DOM e Eventos	46
10.1	Selecionando elementos	46
10.2	Alterando estilos e classes	46
10.3	Eventos	47
10.4	Formulários	47
10.5	Criando elementos dinamicamente	48
10.6	Delegação de eventos	48
10.7	Exemplo integrado: contador	48
10.8	Exercícios	49
11	Assincronismo e APIs	50
11.1	Entendendo a ordem de execução	50
11.2	Promises	50
11.3	async/await	51
11.4	Tratamento de erros com try/catch	51
11.5	Consumindo API com fetch	52
11.6	Verificando status HTTP	52
11.7	Exemplo prático: previsão do tempo	52
11.8	Exercícios	53
12	Testes e Boas Práticas	54
12.1	Boas práticas essenciais	54
12.2	Exemplo: função clara e validada	54
12.3	Tratamento de erros	54
12.4	Validações úteis	55
12.5	Introdução a testes unitários com Vitest	55
12.6	Testando casos de erro	56
12.7	Cobertura mínima de testes	56
12.8	Checklist de revisão antes de entregar	56
12.9	Exercícios	57
13	Projeto Final Integrador	58
13.1	Objetivo do projeto	58
13.2	Funcionalidades obrigatórias	58
13.3	Funcionalidades recomendadas	58
13.4	Estrutura sugerida	58
13.5	Modelo da tarefa	59

13.6	Etapa 1: interface base	59
13.7	Etapa 2: estado da aplicação	59
13.8	Etapa 3: criar tarefas	60
13.9	Etapa 4: renderizar lista	60
13.10	Etapa 5: concluir e remover	60
13.11	Etapa 6: persistência	60
13.12	Etapa 7: validação	61
13.13	Roteiro de implementação	61
13.14	Critérios de avaliação	61
13.15	Desafios extras	62
13.16	Entrega sugerida	62

Bem Vindo

1 Introdução

Este livro foi preparado para estudantes do ensino médio técnico que estão começando em JavaScript e querem aprender de forma prática.

Ao longo dos capítulos, você vai sair do básico (variáveis e decisões) até temas mais avançados, como módulos, orientação a objetos, eventos de interface, APIs e testes.

1.1 Como estudar este livro

1. Estude um capítulo por vez, na ordem.
2. Digite os exemplos no seu computador, em vez de só ler.
3. Faça pequenas mudanças no código para observar o que acontece.
4. Resolva os exercícios antes de olhar soluções prontas.
5. Use o projeto final para consolidar os conceitos.

1.2 O que você vai construir como estudante

- programas simples de cálculo (média, desconto, orçamento);
- pequenos sistemas com listas (tarefas, alunos, produtos);
- páginas com interação (DOM e eventos);
- consumo de APIs para trazer dados reais;
- testes para verificar se funções estão corretas.

1.3 Pré-requisitos

- lógica de programação básica;
- editor de código (VS Code recomendado);
- navegador atualizado (Chrome, Edge ou Firefox);
- Node.js (recomendado a partir dos capítulos de módulos e testes).

1.4 Preparando o ambiente

1.4.1 1) Criar uma pasta de estudos

Crie uma pasta chamada `javascript-estudos` e, dentro dela, uma pasta por capítulo.

1.4.2 2) Teste no navegador

Crie um arquivo `index.html`:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8" />
  <title>Teste JavaScript</title>
</head>
<body>
  <h1>Meu primeiro teste</h1>
  <script>
    console.log("JavaScript rodando no navegador");
  </script>
</body>
</html>
```

Abra esse arquivo no navegador e depois abra o console com F12.

1.4.3 3) Teste no Node.js

Crie um arquivo `app.js`:

```
console.log("JavaScript rodando com Node.js");
```

No terminal:

```
node app.js
```

1.5 Dica de rotina semanal

1. Segunda: leitura de teoria (30 min).
2. Terça: prática de exemplos (40 min).
3. Quarta: exercícios (40 min).
4. Quinta: revisão dos erros (20 min).
5. Sexta: mini desafio com código próprio (30 min).

1.6 Erros comuns de quem está começando

1. Copiar código sem testar.
2. Pular exercícios.
3. Tentar aprender tudo em um único dia.
4. Não ler mensagens de erro no console.
5. Desistir ao primeiro bug.

1.7 Objetivo final

No último capítulo, você vai montar um projeto completo conectando os conteúdos estudados. A ideia é terminar com segurança para continuar aprendendo frameworks e desenvolvimento web moderno.

2 O que é JavaScript?

JavaScript é uma linguagem de programação criada para dar interatividade às páginas web. Com o tempo, ela também passou a ser usada no servidor, em aplicativos mobile, desktop e automações.

2.1 Por que aprender JavaScript?

1. É a linguagem principal da web.
2. Tem grande mercado de trabalho.
3. Permite criar projetos completos (front-end e back-end).
4. É ótima para começar lógica e programação prática.

2.2 Onde o JavaScript roda?

- navegador (Chrome, Edge, Firefox);
- servidor com Node.js;
- aplicações desktop e mobile usando frameworks.

2.3 Seu primeiro programa

```
console.log("Olá, mundo!");
```

Esse comando imprime uma mensagem no console.

2.4 Rodando no navegador

Crie um arquivo `index.html`:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8" />
  <title>Primeiro JS</title>
</head>
<body>
  <h1>Teste JavaScript</h1>
  <script>
    console.log("Executando no navegador");
  </script>
</body>
</html>
```

2.5 Rodando com Node.js

Crie `app.js`:

```
console.log("Executando com Node.js");
```

No terminal:

```
node app.js
```

2.6 JavaScript no dia a dia de um estudante

Exemplo: calcular tempo de deslocamento até a escola.

```
const distanciaKm = 4;
const velocidadeKmH = 5;
const tempoHoras = distanciaKm / velocidadeKmH;

console.log(`Tempo estimado: ${tempoHoras} hora(s)`);
```

Exemplo: simular gasto diário com lanche.

```
const salgado = 8;
const suco = 6;
const total = salgado + suco;

console.log(`Gasto de hoje: R$ ${total}`);
```

2.7 Comentários no código

```
// Comentário de uma linha

/*
  Comentário
  de múltiplas linhas
*/
```

Use comentários para explicar decisões importantes, não para repetir o óbvio.

2.8 Boas práticas iniciais

1. Nomeie variáveis com clareza.
2. Mantenha o código indentado.
3. Teste em pequenos passos.
4. Leia mensagens de erro com atenção.

2.9 Exercícios

1. Mostre seu nome, turma e cidade no console.
2. Crie um programa que calcule o custo de 5 passagens de ônibus.
3. Crie um HTML com botão e, no `<script>`, mostre no console "Página carregada".
4. Desafio: simule o custo semanal de transporte para a escola.

3 Variáveis e Tipos

Variável é um espaço na memória usado para armazenar dados durante a execução do programa.

3.1 let, const e var

- **let**: valor pode mudar.
- **const**: valor não pode ser reatribuído.
- **var**: forma antiga, evite em novos projetos.

```
let idade = 16;
const escola = "Escola Técnica";

idade = 17;

console.log(idade);
console.log(escola);
```

3.2 Regras de nome de variáveis

1. Comece com letra, _ ou \$.
2. Não use espaços.
3. Evite nomes genéricos como x, y, z em projetos reais.
4. Use nomes claros: `notaFinal`, `valorTotal`, `nomeAluno`.

3.3 Tipos primitivos

- **string** (texto)
- **number** (número)
- **boolean** (true ou false)
- **null**

- undefined
- bigint
- symbol

```
const nome = "Ana";
const nota = 8.5;
const aprovado = true;

console.log(typeof nome);    // string
console.log(typeof nota);    // number
console.log(typeof aprovado); // boolean
```

3.4 Conversão de tipos

```
const textoNumero = "42";
const numero = Number(textoNumero);

console.log(numero + 8); // 50
```

Outras conversões úteis:

```
console.log(String(150));    // "150"
console.log(Boolean(1));     // true
console.log(Boolean(0));     // false
```

3.5 NaN e validação de número

NaN significa “Not a Number”, quando uma operação numérica falha.

```
const entrada = "abc";
const valor = Number(entrada);

console.log(valor); // NaN
console.log(Number.isNaN(valor)); // true
```


3.6 Template strings

Template string usa crase para montar frases com variáveis.

```
const aluno = "Carlos";
const media = 7.8;
console.log(`Aluno: ${aluno} | Média: ${media}`);
```

3.7 Exemplo prático: orçamento de passeio

```
const transporte = 50;
const alimentacao = 45;
const ingresso = 30;

const total = transporte + alimentacao + ingresso;
console.log(`Custo total do passeio: R$ ${total}`);
```

3.8 Exemplo prático: situação do estudante

```
const faltas = 12;
const limite = 15;
const podeFaltarMais = faltas < limite;

console.log(`Pode faltar mais? ${podeFaltarMais}`);
```

3.9 Exercícios

1. Declare variáveis para nome, idade e cidade e exiba no console.
2. Converta a string "150" para número e some com 25.
3. Crie uma mensagem com template string informando produto e preço.
4. Crie um programa que receba preço e quantidade e exiba o total.
5. Desafio: leia uma string numérica e valide se a conversão resultou em número válido.

4 Operadores e Estruturas de Controle

Neste capítulo, você vai aprender a tomar decisões no código e repetir tarefas com laços.

4.1 Operadores aritméticos

```
const a = 10;
const b = 3;

console.log(a + b); // soma
console.log(a - b); // subtração
console.log(a * b); // multiplicação
console.log(a / b); // divisão
console.log(a % b); // resto
```

4.2 Operadores relacionais

```
console.log(8 > 5); // true
console.log(8 >= 8); // true
console.log(8 < 5); // false
console.log(8 === 8); // true
console.log(8 !== 7); // true
```

4.3 Operadores lógicos

- && (E): tudo precisa ser verdadeiro.
- || (OU): ao menos uma condição verdadeira.
- ! (NÃO): inverte.

```
const nota = 7;
const frequencia = 80;

const aprovado = nota >= 6 && frequencia >= 75;
console.log(aprovado);
```

4.4 if, else if, else

```
const temperatura = 18;

if (temperatura < 15) {
  console.log("Frio");
} else if (temperatura <= 25) {
  console.log("Agradável");
} else {
  console.log("Quente");
}
```

4.5 Exemplo escolar: status do aluno

```
const media = 6.2;

if (media >= 7) {
  console.log("Aprovado");
} else if (media >= 5) {
  console.log("Recuperação");
} else {
  console.log("Reprovado");
}
```

4.6 switch

```
const dia = 3;
```

```
switch (dia) {  
  case 1:  
    console.log("Domingo");  
    break;  
  case 2:  
    console.log("Segunda");  
    break;  
  case 3:  
    console.log("Terça");  
    break;  
  default:  
    console.log("Dia inválido");  
}
```

4.7 Operador ternário

```
const idade = 17;  
const status = idade >= 18 ? "Maior de idade" : "Menor de idade";  
console.log(status);
```

4.8 Laço for

```
for (let i = 1; i <= 5; i++) {  
  console.log(`Valor: ${i}`);  
}
```

4.9 Laço while

```
let contador = 1;  
  
while (contador <= 3) {  
  console.log(contador);  
  contador++;  
}
```

4.10 break e continue

```
for (let i = 1; i <= 10; i++) {  
  if (i === 4) continue;  
  if (i === 8) break;  
  console.log(i);  
}
```

4.11 Exemplo prático: orçamento até limite

```
const gastos = [20, 35, 18, 40, 12];  
let total = 0;  
  
for (let i = 0; i < gastos.length; i++) {  
  if (total + gastos[i] > 90) {  
    console.log("Limite de orçamento atingido");  
    break;  
  }  
  
  total += gastos[i];  
}  
  
console.log(`Total acumulado: R$ ${total}`);
```

4.12 Exercícios

1. Classifique uma nota em A, B, C ou D.
2. Mostre os números pares de 2 a 20.
3. Calcule a soma de 1 até 100 com laço.
4. Use **switch** para exibir o nome do mês com base no número.
5. Desafio: simule uma compra que para quando atingir um limite de gasto.

5 Funções

Funções são blocos de código reutilizáveis.

Elas recebem dados de entrada, executam uma tarefa e podem devolver um resultado.

5.1 Por que funções são importantes?

1. Evitam repetição de código.
2. Organizam melhor o programa.
3. Facilitam testes e manutenção.
4. Permitem dividir problemas grandes em partes menores.

5.2 Declaração de função

```
function saudacao(nome) {  
  return `Olá, ${nome}!`;  
}  
  
console.log(saudacao("Marina"));
```

5.3 Parâmetros e retorno

```
function media(n1, n2, n3) {  
  return (n1 + n2 + n3) / 3;  
}  
  
const resultado = media(7, 8, 9);  
console.log(resultado);
```

5.3.1 Exemplo do cotidiano estudantil

```
function mediaPresenca(aula1, aula2, aula3, aula4) {  
  return (aula1 + aula2 + aula3 + aula4) / 4;  
}  
  
const presenca = mediaPresenca(100, 100, 75, 100);  
console.log(`Média de presença: ${presenca}%`);
```

5.4 Parâmetros com valor padrão

```
function calcularIngresso(valor, taxaServico = 5) {  
  return valor + taxaServico;  
}  
  
console.log(calcularIngresso(30));    // 35  
console.log(calcularIngresso(30, 8)); // 38
```

5.5 Função anônima em variável

```
const dobro = function (n) {  
  return n * 2;  
};  
  
console.log(dobro(9));
```

5.6 Arrow functions

```
const quadrado = (n) => n * n;  
console.log(quadrado(6));
```

Exemplo com turismo:

```
const calcularGastoPasseio = (transporte, alimentacao, ingresso) =>
  transporte + alimentacao + ingresso;

const gastoTotal = calcularGastoPasseio(80, 55, 30);
console.log(`Gasto total no passeio: R$ ${gastoTotal}`);
```

5.7 Escopo

- variáveis globais: acessíveis fora de funções;
- variáveis locais: só existem dentro da função.

```
let curso = "Informática";

function mostrarCurso() {
  let modulo = 2;
  console.log(curso);
  console.log(modulo);
}

mostrarCurso();
```

5.8 Funções puras e efeitos colaterais

Função pura: para a mesma entrada, sempre devolve a mesma saída e não altera estado externo.

```
function somar(a, b) {
  return a + b;
}
```

Função com efeito colateral:

```
let contador = 0;

function incrementarContador() {
  contador++;
}
```


5.9 Função que calcula x função que exhibe

```
function calcularDesconto(valorCompra, porcentagem) {
  return valorCompra - (valorCompra * porcentagem) / 100;
}

function mostrarPrecoFinal(valorOriginal, desconto) {
  const precoFinal = calcularDesconto(valorOriginal, desconto);
  console.log(`De R$ ${valorOriginal} por R$ ${precoFinal}`);
}

mostrarPrecoFinal(120, 10);
```

5.10 Callback (função passada para outra função)

```
function processarAluno(nome, callback) {
  const mensagem = `Aluno: ${nome}`;
  callback(mensagem);
}

processarAluno("Bruna", (texto) => {
  console.log(texto.toUpperCase());
});
```

5.11 Exemplo integrado: roteiro de viagem

```
function somarDespesas(listaDeValores) {
  let total = 0;

  for (const valor of listaDeValores) {
    total += valor;
  }

  return total;
}
```

```
function calcularCustoPorPessoa(total, quantidadePessoas) {  
  return total / quantidadePessoas;  
}  
  
const despesas = [90, 60, 40, 30];  
const totalViagem = somarDespesas(despesas);  
const custoIndividual = calcularCustoPorPessoa(totalViagem, 4);  
  
console.log(`Total da viagem: R$ ${totalViagem}`);  
console.log(`Custo por pessoa: R$ ${custoIndividual}`);
```

5.12 Boas práticas para funções

1. Use nomes claros (calcularMedia, buscarAluno).
2. Faça funções curtas e com um objetivo.
3. Prefira `return` para reaproveitamento.
4. Evite misturar cálculo com interface.
5. Teste cenários comuns e cenários de erro.

5.13 Exercícios

1. Crie `tempoCaminhada(distanciaKm, velocidadeKmH)`.
2. Crie `converterRealParaGuarani(real, cotacao)`.
3. Crie `custoViagemBonito(transporte, hospedagem, alimentacao, passeios)`.
4. Crie `mediaNotas(notas)` para um array de notas.
5. Desafio: `situacaoAluno(media, frequencia)` com regras definidas por você.

6 Arrays e Objetos

Arrays e objetos são estruturas fundamentais para organizar dados.

- array: lista ordenada de valores.
- objeto: coleção de pares **chave: valor**.

6.1 Arrays

```
const notas = [7.5, 8.0, 6.5, 9.0];

console.log(notas[0]);    // primeiro item
console.log(notas.length); // quantidade de itens
```

6.2 Métodos mais usados em arrays

```
const alunos = ["Ana", "Bruno"];

alunos.push("Carla"); // adiciona no fim
const removido = alunos.pop(); // remove do fim

console.log(removido);
console.log(alunos);
```

Outros úteis:

```
const fila = ["A", "B", "C"];
fila.shift(); // remove do início
fila.unshift("X"); // adiciona no início
console.log(fila);
```

6.3 Percorrendo arrays

6.3.1 for tradicional

```
const precos = [12, 30, 18];
let total = 0;

for (let i = 0; i < precos.length; i++) {
  total += precos[i];
}

console.log(`Total: R$ ${total}`);
```

6.3.2 for...of

```
const cidades = ["Ponta Porã", "Dourados", "Bonito"];

for (const cidade of cidades) {
  console.log(cidade);
}
```

6.4 map, filter, reduce

```
const numeros = [1, 2, 3, 4, 5];

const dobrados = numeros.map((n) => n * 2);
const pares = numeros.filter((n) => n % 2 === 0);
const soma = numeros.reduce((acc, n) => acc + n, 0);

console.log(dobrados);
console.log(pares);
console.log(soma);
```

6.5 Objetos

```
const aluno = {
  nome: "João",
  turma: "2A",
  ativo: true,
  exibir() {
    return `${this.nome} - ${this.turma}`;
  }
};

console.log(aluno.exibir());
```

6.6 Acessando e alterando propriedades

```
const produto = {
  nome: "Caderno",
  preco: 25
};

console.log(produto.nome);
produto.preco = 22;
produto.estoque = 15;
console.log(produto);
```

6.7 Array de objetos

Muito comum em sistemas:

```
const passeios = [
  { local: "Bonito", preco: 120 },
  { local: "Pantanal", preco: 200 },
  { local: "Campo Grande", preco: 80 }
];

const baratos = passeios.filter((p) => p.preco <= 120);
console.log(baratos);
```

6.8 Desestruturação

```
const estudante = {  
  nome: "Livia",  
  cidade: "Ponta Porã",  
  curso: "Informática"  
};  
  
const { nome, cidade } = estudante;  
console.log(nome, cidade);
```

6.9 Spread em arrays e objetos

```
const base = [1, 2, 3];  
const copia = [...base, 4];  
  
const alunoBase = { nome: "Rafa", turma: "1B" };  
const alunoAtualizado = { ...alunoBase, turma: "2B" };  
  
console.log(copia);  
console.log(alunoAtualizado);
```

6.10 Exercícios

1. Crie um array com 10 números e exiba apenas os ímpares.
2. Dado um array de preços, calcule o total com **reduce**.
3. Crie um objeto **produto** com nome, preço e estoque.
4. Crie um array de objetos **alunos** com nome e nota; exiba só os aprovados.
5. Desafio: monte uma lista de passeios com preço e exiba o mais barato.

7 Vetores, Matrizes, Fila e Pilha

Este capítulo aprofunda estruturas lineares muito usadas em programação.

7.1 Vetores (arrays unidimensionais)

Vetores armazenam elementos em sequência e cada posição tem um índice.

7.1.1 Exemplo 1: notas de uma turma

```
const notas = [7.0, 8.5, 6.0, 9.0];

console.log(`Primeira nota: ${notas[0]}`);
console.log(`Quantidade de notas: ${notas.length}`);
```

7.1.2 Exemplo 2: soma e média

```
const valores = [10, 20, 30, 40];
let soma = 0;

for (let i = 0; i < valores.length; i++) {
  soma += valores[i];
}

const media = soma / valores.length;
console.log(`Soma: ${soma} | Média: ${media}`);
```

7.1.3 Exemplo 3: filtro de valores

```
const numeros = [3, 8, 11, 14, 19, 20];
const pares = numeros.filter((n) => n % 2 === 0);

console.log(pares);
```

7.2 Matrizes (arrays bidimensionais)

Uma matriz é um array de arrays, com linhas e colunas.

7.2.1 Exemplo 1: acesso por linha e coluna

```
const matriz = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log(matriz[1][2]); // 6
```

7.2.2 Exemplo 2: percorrer todos os elementos

```
const tabela = [
  [10, 20],
  [30, 40],
  [50, 60]
];

for (let linha = 0; linha < tabela.length; linha++) {
  for (let coluna = 0; coluna < tabela[linha].length; coluna++) {
    console.log(`Posição [${linha}][${coluna}] = ${tabela[linha][coluna]}`);
  }
}
```


7.2.3 Exemplo 3: diagonal principal

```
const m = [
  [2, 1, 0],
  [4, 3, 8],
  [9, 7, 6]
];

let diagonal = 0;

for (let i = 0; i < m.length; i++) {
  diagonal += m[i][i];
}

console.log(`Soma diagonal principal: ${diagonal}`);
```

7.3 Fila (FIFO: First In, First Out)

Em fila, o primeiro a entrar é o primeiro a sair.

7.3.1 Exemplo 1: atendimento

```
const fila = [];

fila.push("Aluno 1");
fila.push("Aluno 2");
fila.push("Aluno 3");

const atendido = fila.shift();
console.log(`Atendido: ${atendido}`);
console.log(fila);
```

7.3.2 Exemplo 2: próximo da fila

```
const filaSenhas = ["A01", "A02", "A03"];
console.log(`Próximo: ${filaSenhas[0]}`);
```

7.3.3 Exemplo 3: fila com validação

```
const filaLaboratorio = [];  
  
function chamarAluno() {  
  if (filaLaboratorio.length === 0) {  
    return "Fila vazia";  
  }  
  
  return filaLaboratorio.shift();  
}  
  
console.log(chamarAluno());  
filaLaboratorio.push("Carlos");  
console.log(chamarAluno());
```

7.4 Pilha (LIFO: Last In, First Out)

Em pilha, o último a entrar é o primeiro a sair.

7.4.1 Exemplo 1: empilhar e desempilhar

```
const pilha = [];  
  
pilha.push("Prato 1");  
pilha.push("Prato 2");  
pilha.push("Prato 3");  
  
const removido = pilha.pop();  
console.log(`Removido: ${removido}`);  
console.log(pilha);
```

7.4.2 Exemplo 2: topo sem remover

```
const historico = ["home", "produtos", "checkout"];
const topo = historico[historico.length - 1];

console.log(`Página atual: ${topo}`);
```

7.4.3 Exemplo 3: desfazer ação

```
const acoes = [];

function executar(acao) {
  acoes.push(acao);
}

function desfazer() {
  if (acoes.length === 0) return "Nada para desfazer";
  return `Desfez: ${acoes.pop()}`;
}

executar("Digitou texto");
executar("Apagou linha");
console.log(desfazer());
console.log(desfazer());
console.log(desfazer());
```

7.5 Aplicação prática combinando as estruturas

```
const filaAtendimento = ["Aluno A", "Aluno B"];
const historicoChamadas = [];
const temposEspera = [4, 6, 3, 5];

const chamado = filaAtendimento.shift();
historicoChamadas.push(chamado);

const mediaEspera =
  temposEspera.reduce((acc, t) => acc + t, 0) / temposEspera.length;

console.log(`Chamado agora: ${chamado}`);
console.log(`Média de espera: ${mediaEspera} min`);
```

7.6 Exercícios

1. Crie um vetor de 6 números e calcule a média.
2. Crie uma matriz 3×2 e exiba só os valores maiores que 25.
3. Simule uma fila com 4 pessoas e atenda 2.
4. Simule uma pilha de páginas e implemente “voltar”.
5. Desafio: combine fila e pilha em um sistema simples de atendimento escolar.

8 Funções, Blocos, Procedimentos, Módulos e Parâmetros

Este capítulo organiza conceitos importantes para escrever código legível, reutilizável e escalável.

8.1 Blocos

Bloco é o trecho entre chaves {}. Ele delimita escopo.

```
let turma = "2A";

{
  let turma = "3A";
  console.log(`Dentro do bloco: ${turma}`);
}

console.log(`Fora do bloco: ${turma}`);
```

8.2 Procedimentos

Procedimento executa uma ação e normalmente não depende de retorno.

```
function mostrarCabecalho() {
  console.log("=== Sistema Acadêmico ===");
}

mostrarCabecalho();
```

Com parâmetro:

```
function saudarAluno(nome) {  
  console.log(`Bem-vindo(a), ${nome}`);  
}  
  
saudarAluno("Marina");
```

8.3 Funções com retorno

```
function calcularMedia(n1, n2, n3) {  
  return (n1 + n2 + n3) / 3;  
}  
  
console.log(calcularMedia(7, 8, 9));
```

8.4 Passagem de parâmetros

8.4.1 Primitivo: cópia por valor

```
let quantidade = 5;  
  
function alterarNumero(n) {  
  n = 99;  
}  
  
alterarNumero(quantidade);  
console.log(quantidade); // 5
```

8.4.2 Objeto: referência compartilhada

```
const aluno = { nome: "Ana", nota: 8 };  
  
function atualizarNota(dados) {  
  dados.nota = 10;  
}
```

```
atualizarNota(aluno);  
console.log(aluno.nota); // 10
```

8.4.3 Evitando mutação com spread

```
const tarefa = { titulo: "Estudar", concluida: false };  
  
function concluirSemMutar(item) {  
  return { ...item, concluida: true };  
}  
  
const tarefaConcluida = concluirSemMutar(tarefa);  
console.log(tarefa);  
console.log(tarefaConcluida);
```

8.5 Modularização

Quando o projeto cresce, separar por arquivos ajuda muito.

8.5.1 Export nomeado

matematica.js

```
export function somar(a, b) {  
  return a + b;  
}
```

main.js

```
import { somar } from "./matematica.js";  
  
console.log(somar(5, 7));
```

8.5.2 Múltiplos exports

boletim.js

```
export function media(a, b) {  
  return (a + b) / 2;  
}  
  
export function aprovado(mediaFinal) {  
  return mediaFinal >= 6;  
}
```

app.js

```
import { media, aprovado } from "../boletim.js";  
  
const m = media(7, 5);  
console.log(m);  
console.log(aprovado(m));
```

8.5.3 Export default

mensagem.js

```
export default function gerarMensagem(nome) {  
  return `Olá, ${nome}!`;  
}
```

principal.js

```
import gerarMensagem from "../mensagem.js";  
  
console.log(gerarMensagem("Carlos"));
```

8.6 Estrutura sugerida para projetos pequenos


```
projeto/  
  src/  
    calculos.js  
    validacoes.js  
    app.js
```

8.7 Exemplo integrador

calculos.js

```
export function totalDespesas(lista) {  
  return lista.reduce((acc, item) => acc + item, 0);  
}
```

app.js

```
import { totalDespesas } from "./calculos.js";  
  
const gastos = [45, 30, 20, 10];  
console.log(`Total: R$ ${totalDespesas(gastos)}`);
```

8.8 Exercícios

1. Crie um módulo `conversoes.js` com metro para centímetro.
2. Crie um módulo `boletim.js` com funções de média e status.
3. Crie um `export default` para formatar preço em reais.
4. Desafio: monte um mini projeto com `app.js`, `dados.js` e `calculos.js`.

9 POO e Módulos

Programação Orientada a Objetos (POO) ajuda a modelar o mundo real em código usando classes e objetos.

9.1 Conceitos básicos de POO

- classe: molde para criar objetos;
- objeto: instância da classe;
- atributo: característica do objeto;
- método: comportamento do objeto.

9.2 Criando uma classe

```
class Pessoa {  
  constructor(nome, idade) {  
    this.nome = nome;  
    this.idade = idade;  
  }  
  
  apresentar() {  
    return `Meu nome é ${this.nome} e tenho ${this.idade} anos.`;  
  }  
}  
  
const p1 = new Pessoa("Lia", 17);  
console.log(p1.apresentar());
```

9.3 Encapsulando regras com métodos

```

class ContaEstudante {
  constructor(saldoInicial = 0) {
    this.saldo = saldoInicial;
  }

  depositar(valor) {
    if (valor <= 0) return;
    this.saldo += valor;
  }

  sacar(valor) {
    if (valor > this.saldo) return "Saldo insuficiente";
    this.saldo -= valor;
    return "Saque realizado";
  }
}

const conta = new ContaEstudante(100);
conta.depositar(50);
console.log(conta.sacar(30));
console.log(conta.saldo);

```

9.4 Herança

Herança permite reutilizar código de uma classe base.

```

class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }
}

class Aluno extends Pessoa {
  constructor(nome, idade, curso) {
    super(nome, idade);
    this.curso = curso;
  }

  dados() {

```

```

    return `${this.nome} - ${this.curso}`;
  }
}

const aluno = new Aluno("Rafa", 16, "Informática");
console.log(aluno.dados());

```

9.5 Polimorfismo (ideia prática)

Diferentes classes podem ter métodos com o mesmo nome e comportamentos específicos.

```

class Transporte {
  calcularTempo() {
    return "Tempo padrão";
  }
}

class Onibus extends Transporte {
  calcularTempo() {
    return "Tempo estimado: 25 min";
  }
}

class Bicicleta extends Transporte {
  calcularTempo() {
    return "Tempo estimado: 18 min";
  }
}

const opcoes = [new Onibus(), new Bicicleta()];
opcoes.forEach((item) => console.log(item.calcularTempo()));

```

9.6 Relembrando módulos ES

9.6.1 Export nomeado

matematica.js

```
export function somar(a, b) {  
  return a + b;  
}
```

main.js

```
import { somar } from "./matematica.js";  
console.log(somar(5, 7));
```

9.6.2 Export default

formatador.js

```
export default function formatarMoeda(valor) {  
  return `R$ ${valor.toFixed(2)}`;  
}
```

app.js

```
import formatarMoeda from "./formatador.js";  
console.log(formatarMoeda(35.5));
```

9.7 Exemplo integrado: classe + módulo

produto.js

```
export class Produto {  
  constructor(nome, preco) {  
    this.nome = nome;  
    this.preco = preco;  
  }  
  
  aplicarDesconto(percentual) {  
    this.preco -= this.preco * (percentual / 100);  
  }  
}
```

index.js

```
import { Produto } from "../produto.js";

const caderno = new Produto("Caderno", 30);
caderno.aplicarDesconto(10);
console.log(caderno);
```

9.8 Exercícios

1. Crie uma classe `Produto` com nome, preço e método de desconto.
2. Crie `Funcionario` e `Professor` usando herança.
3. Crie uma classe `RoteiroTuristico` com método para calcular custo total.
4. Separe funções em módulos e importe no arquivo principal.
5. Desafio: combine uma classe com persistência em array de objetos.

10 DOM e Eventos

DOM (Document Object Model) é a estrutura da página HTML que o JavaScript consegue manipular.

Com DOM e eventos, você cria interfaces interativas.

10.1 Selecionando elementos

```
<h2 id="titulo">Loja Técnica</h2>  
<button id="botao">Clique</button>
```

```
const titulo = document.getElementById("titulo");  
const botao = document.getElementById("botao");  
  
titulo.textContent = "Sistema Escolar";
```

Outros seletores:

```
const primeiroCard = document.querySelector(".card");  
const todosOsCards = document.querySelectorAll(".card");
```

10.2 Alterando estilos e classes

```
const card = document.querySelector(".card");  
  
card.classList.add("ativo");  
card.classList.toggle("destaque");  
card.style.border = "2px solid #1f6feb";
```

10.3 Eventos

```
botao.addEventListener("click", () => {  
  alert("Botão clicado!");  
});
```

Eventos comuns:

- click
- input
- change
- submit
- keydown

10.4 Formulários

```
<form id="formAluno">  
  <input id="nome" placeholder="Nome" />  
  <button type="submit">Salvar</button>  
</form>
```

```
const form = document.getElementById("formAluno");  
  
form.addEventListener("submit", (event) => {  
  event.preventDefault();  
  
  const nome = document.getElementById("nome").value.trim();  
  
  if (!nome) {  
    alert("Informe o nome");  
    return;  
  }  
  
  console.log(`Aluno cadastrado: ${nome}`);  
});
```


10.5 Criando elementos dinamicamente

```
<ul id="lista"></ul>
```

```
const lista = document.getElementById("lista");
const itens = ["Caderno", "Caneta", "Lápis"];

for (const item of itens) {
  const li = document.createElement("li");
  li.textContent = item;
  lista.appendChild(li);
}
```

10.6 Delegação de eventos

Útil quando itens são criados dinamicamente.

```
lista.addEventListener("click", (event) => {
  if (event.target.tagName === "LI") {
    event.target.classList.toggle("feito");
  }
});
```

10.7 Exemplo integrado: contador

```
<p id="valor">0</p>
<button id="menos">-</button>
<button id="mais">+</button>
```

```
const campo = document.getElementById("valor");
const botaoMenos = document.getElementById("menos");
const botaoMais = document.getElementById("mais");

let contador = 0;

function renderizar() {
  campo.textContent = contador;
}
```

```
}  
  
botaoMenos.addEventListener("click", () => {  
  contador--;  
  renderizar();  
});  
  
botaoMais.addEventListener("click", () => {  
  contador++;  
  renderizar();  
});
```

10.8 Exercícios

1. Crie uma página com botão que altera a cor de fundo.
2. Faça um contador com botões de aumentar e diminuir.
3. Valide um formulário para impedir envio com campo vazio.
4. Crie uma lista de tarefas com botão de remover item.
5. Desafio: implemente filtro de tarefas (todas, pendentes, concluídas).

11 Assincronismo e APIs

Nem toda tarefa no JavaScript termina imediatamente. Chamamos isso de comportamento assíncrono.

Exemplos:

- requisições de rede;
- leitura de arquivos;
- timers (`setTimeout`, `setInterval`).

11.1 Entendendo a ordem de execução

```
console.log("Início");

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

console.log("Fim");
```

Saída esperada:

1. Início
2. Fim
3. Executou depois de 2 segundos

11.2 Promises

Promise representa um resultado futuro: pendente, resolvido ou rejeitado.

```
function buscarDados() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Dados carregados"), 1000);
  });
}

buscarDados().then((mensagem) => console.log(mensagem));
```

Com erro:

```
function validarIdade(idade) {
  return new Promise((resolve, reject) => {
    if (idade >= 18) resolve("Acesso liberado");
    else reject(new Error("Acesso negado"));
  });
}
```

11.3 async/await

async e await deixam o código assíncrono mais legível.

```
async function executar() {
  const resposta = await buscarDados();
  console.log(resposta);
}

executar();
```

11.4 Tratamento de erros com try/catch

```
async function testarIdade() {
  try {
    const msg = await validarIdade(16);
    console.log(msg);
  } catch (erro) {
    console.error("Erro:", erro.message);
  }
}
```

```
}  
  
testarIdade();
```

11.5 Consumindo API com fetch

```
async function carregarUsuarios() {  
  const response = await fetch("https://jsonplaceholder.typicode.com/users");  
  const usuarios = await response.json();  
  console.log(usuarios);  
}  
  
carregarUsuarios();
```

11.6 Verificando status HTTP

```
async function buscarPost(id) {  
  const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`);  
  
  if (!response.ok) {  
    throw new Error(`Falha HTTP: ${response.status}`);  
  }  
  
  return response.json();  
}
```

11.7 Exemplo prático: previsão do tempo

Fluxo comum:

1. usuário escolhe cidade;
2. aplicação chama API;
3. mostra temperatura e condição do clima;
4. em erro, exibe mensagem amigável.

```
async function carregarClima(url) {  
  try {  
    const response = await fetch(url);  
  
    if (!response.ok) {  
      throw new Error("Não foi possível buscar o clima");  
    }  
  
    const dados = await response.json();  
    console.log(dados);  
  } catch (erro) {  
    console.error("Erro ao carregar clima:", erro.message);  
  }  
}
```

11.8 Exercícios

1. Faça uma função assíncrona que aguarde 3 segundos e retorne mensagem.
2. Consuma uma API pública e mostre apenas os nomes retornados.
3. Trate erros com `try/catch` no `fetch`.
4. Crie função que busque um post por ID e valide status da resposta.
5. Desafio: montar tela com busca e carregamento (`loading`) para uma API.

12 Testes e Boas Práticas

Escrever código que funciona é importante.

Escrever código confiável e fácil de manter é essencial.

12.1 Boas práticas essenciais

1. Use nomes claros para variáveis e funções.
2. Crie funções pequenas com uma responsabilidade.
3. Evite duplicação de código.
4. Trate erros de forma explícita.
5. Comente apenas quando necessário.

12.2 Exemplo: função clara e validada

```
function calcularMedia(notas) {  
  if (!Array.isArray(notas) || notas.length === 0) {  
    throw new Error("Informe um array de notas");  
  }  
  
  const soma = notas.reduce((acc, n) => acc + n, 0);  
  return soma / notas.length;  
}
```

12.3 Tratamento de erros

```
function dividir(a, b) {  
  if (b === 0) {  
    throw new Error("Divisão por zero não permitida");  
  }  
}
```

```

    return a / b;
}

try {
  console.log(dividir(10, 0));
} catch (erro) {
  console.error("Erro:", erro.message);
}

```

12.4 Validações úteis

```

function criarAluno(nome, idade) {
  if (!nome || typeof nome !== "string") {
    throw new Error("Nome inválido");
  }

  if (typeof idade !== "number" || idade < 0) {
    throw new Error("Idade inválida");
  }

  return { nome, idade };
}

```

12.5 Introdução a testes unitários com Vitest

sum.js

```

export function sum(a, b) {
  return a + b;
}

```

sum.test.js

```

import { describe, it, expect } from "vitest";
import { sum } from "../sum.js";

describe("sum", () => {

```



```
it("deve somar dois números", () => {  
  expect(sum(2, 3)).toBe(5);  
});  
});
```

12.6 Testando casos de erro

dividir.test.js

```
import { describe, it, expect } from "vitest";  
import { dividir } from "../dividir.js";  
  
describe("dividir", () => {  
  it("deve lançar erro quando divisor é zero", () => {  
    expect(() => dividir(10, 0)).toThrow("Divisão por zero");  
  });  
});
```

12.7 Cobertura mínima de testes

Para funções críticas, tente cobrir:

1. caso comum;
2. caso de borda;
3. caso inválido.

12.8 Checklist de revisão antes de entregar

1. O código está legível?
2. Há validações mínimas?
3. Erros são tratados?
4. Existem testes para as funções principais?
5. Os nomes estão claros?

12.9 Exercícios

1. Refatore um código longo em funções menores.
2. Implemente validações com `throw` e `try/catch`.
3. Escreva 3 testes para funções matemáticas simples.
4. Escreva teste para um cenário de erro.
5. Desafio: testar uma função que recebe array e retorna média.

13 Projeto Final Integrador

Neste capítulo, você vai aplicar os principais conceitos do livro em um projeto completo.

13.1 Objetivo do projeto

Desenvolver um sistema de cadastro e organização de tarefas escolares.

13.2 Funcionalidades obrigatórias

1. Cadastrar tarefa com título e prazo.
2. Marcar tarefa como concluída.
3. Excluir tarefa.
4. Salvar dados em `localStorage`.
5. Organizar código em funções e, se possível, módulos.

13.3 Funcionalidades recomendadas

1. Filtro por status (todas, pendentes, concluídas).
2. Busca por texto.
3. Contador de tarefas concluídas e pendentes.
4. Validação de formulário.

13.4 Estrutura sugerida

```
projeto/  
  index.html  
  style.css  
  app.js
```

Versão modular:

```
projeto/  
  index.html  
  style.css  
  src/  
    app.js  
    storage.js  
    tarefas.js  
    ui.js
```

13.5 Modelo da tarefa

```
const tarefa = {  
  id: Date.now(),  
  titulo: "Estudar JavaScript",  
  prazo: "2026-03-01",  
  concluida: false  
};
```

13.6 Etapa 1: interface base

Campos mínimos:

- input de título;
- input de prazo;
- botão de salvar;
- lista de tarefas.

13.7 Etapa 2: estado da aplicação

```
let tarefas = [];
```

13.8 Etapa 3: criar tarefas

```
function criarTarefa(titulo, prazo) {  
  return {  
    id: Date.now(),  
    titulo,  
    prazo,  
    concluida: false  
  };  
}
```

13.9 Etapa 4: renderizar lista

```
function renderizarTarefas(lista) {  
  console.log("Renderizar", lista);  
}
```

Depois substitua o `console.log` pela renderização real no DOM.

13.10 Etapa 5: concluir e remover

```
function alternarConclusao(id) {  
  tarefas = tarefas.map((t) =>  
    t.id === id ? { ...t, concluida: !t.concluida } : t  
  );  
}  
  
function removerTarefa(id) {  
  tarefas = tarefas.filter((t) => t.id !== id);  
}
```

13.11 Etapa 6: persistência

```
function salvarNoStorage() {
  localStorage.setItem("tarefas", JSON.stringify(tarefas));
}

function carregarDoStorage() {
  const dados = localStorage.getItem("tarefas");
  tarefas = dados ? JSON.parse(dados) : [];
}
```

13.12 Etapa 7: validação

```
function validarFormulario(titulo, prazo) {
  if (!titulo.trim()) return "Informe o título da tarefa";
  if (!prazo) return "Informe o prazo";
  return null;
}
```

13.13 Roteiro de implementação

1. Montar HTML e CSS.
2. Implementar criação de tarefa.
3. Renderizar no DOM.
4. Implementar concluir/remover.
5. Integrar `localStorage`.
6. Adicionar filtros e busca.
7. Revisar e testar.

13.14 Critérios de avaliação

- funcionamento correto (40%);
- organização e legibilidade (25%);
- uso de boas práticas (15%);
- interface e experiência do usuário (10%);
- documentação do projeto (10%).

13.15 Desafios extras

1. Ordenar por prazo.
2. Destacar tarefas atrasadas automaticamente.
3. Exportar tarefas em JSON.
4. Criar modo escuro com botão de alternância.

13.16 Entrega sugerida

1. Repositório com código fonte.
2. README.md com instruções de uso.
3. Capturas de tela da aplicação.
4. Lista de melhorias futuras.