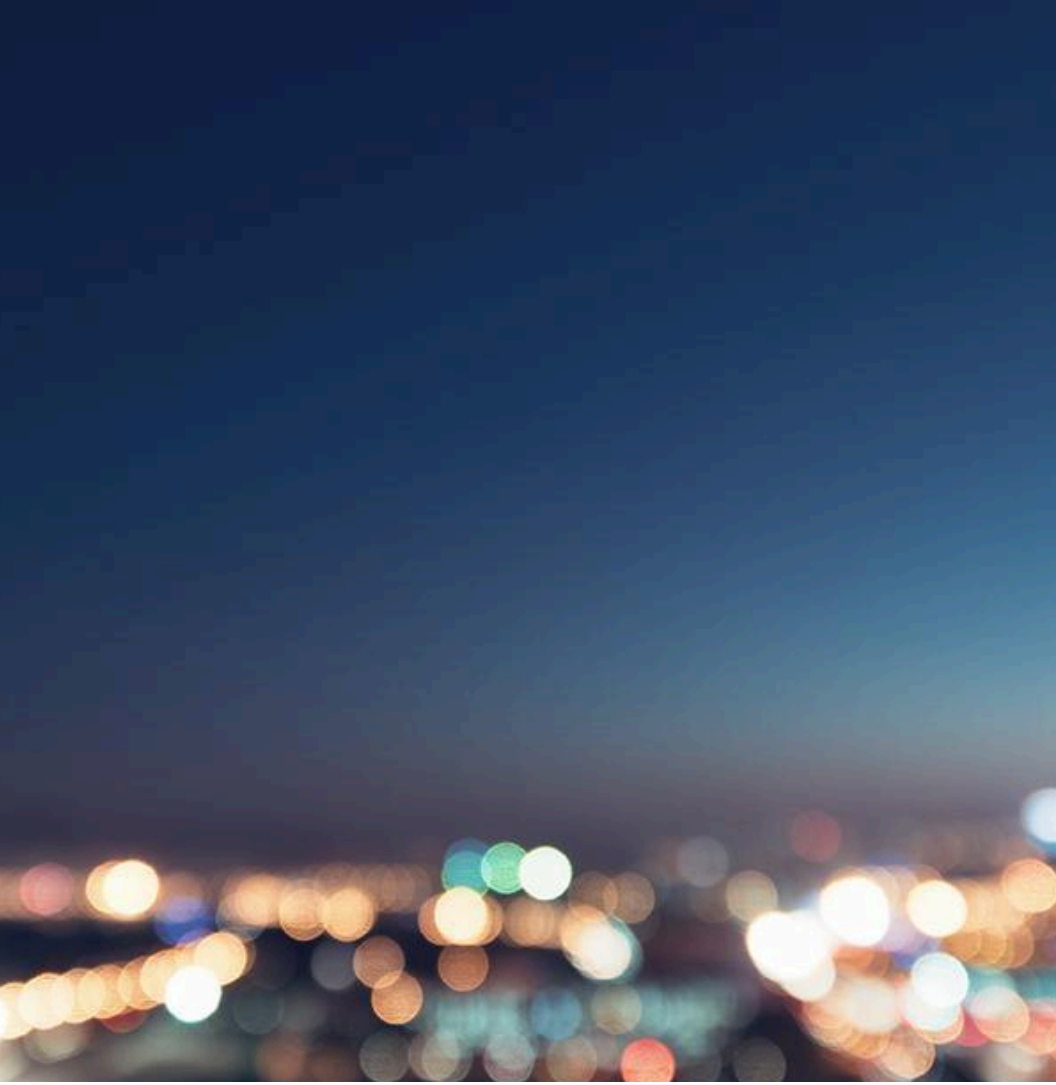

Pós-Graduação em Automação de Testes de Software

PROGRAMAÇÃO PARA AUTOMAÇÃO DE TESTES

Curso de Pós-Graduação ***Lato Sensu*** de Automação de Testes de Software em parceria entre Julio de Lima Consultoria e Treinamentos de Testes e Qualidade de Software e a Faculdade VINCIT.

Dados comerciais:
Julio de Lima Consultoria e Treinamentos de Testes e Qualidade de Software LTDA
Rua Antônio Rosa Felipe, 203
Jardim Universo
Araçatuba, São Paulo
CEP 16056-808
CNPJ 18.726.544/0001-09
IE 177.428.549.110

Contato:
comercial@juliodelima.com.br
(18) 99793-6246

Objetivo Geral

Capacitar os alunos a compreender e aplicar conceitos fundamentais de programação, com foco específico no ambiente de automação de testes de software. Espera-se que os alunos adquiram conhecimentos sólidos sobre os princípios de programação, sintaxe e estruturas de dados em Javascript, com uma compreensão clara de como esses elementos se relacionam com a automação de testes.

Conteúdo

Introdução a Programação com Javascript

A linguagem Javascript é uma das linguagens mais populares no universo de programação atualmente. Segundo a pesquisa Jetbrains Dev Ecosystem 2024, 61% dos desenvolvedores de todo o mundo usam Javascript.

De forma similar, a pesquisa StackOverflow Developer Survey 2024 mostra que o Javascript tem sido a linguagem mais popular em praticamente todos os anos da pesquisa, sendo atualmente dona de 62% das respostas de uso.

O Javascript é uma linguagem relativamente nova. Surgiu em meados de 1996 e desde então não parou de evoluir. Podemos destacar alguns marcos na evolução da linguagem no ecossistema de desenvolvimento, como:

- 1ª versão oficial pela ECMA, o ES1 (1997)
- A principal biblioteca frontend, o JQuery (2006)
- Revolução no uso em projetos, com o NodeJS (2009)
- ES6 e aumento no suporte em navegadores (2013)

Atualmente, existem inúmeras possibilidades de uso do Javascript em software: aplicações web (jquery, react, vue), aplicações mobile (react native), backend (nodejs, deno).

Com isso, naturalmente começaram a surgir ferramentas para diferentes finalidades de testes e com suporte ao uso da linguagem Javascript para a escrita de seus scripts.

Vejamos alguns exemplos e qual o papel do Javascript em cada uma destas ferramentas.

WebDriverIO

Em ferramentas como o WebDriverIO, que é um *wrapper* do Selenium escrito em Javascript, o JS é utilizado para escrever os testes e se comunicar com o WebDriver. Ele envia comandos para o WebDriver, que por sua vez controla o navegador real ou *headless*. O Javascript atua como **ponte entre o script de testes e o navegador**, através do protocolo WebDriver.

Cypress e Playwright

Ao contrário do Selenium, Cypress e Playwright executam o código JS no mesmo loop de eventos do navegador (Cypress) ou controlam diretamente os navegadores via protocolos nativos (Playwright). Nessas ferramentas, o Javascript (ou Typescript) é usado para **escrever testes de ponta a ponta (e2e)**, simulando o comportamento real do usuário no navegador.

K6

No k6, o Javascript é usado para **definir o comportamento do teste de carga**, de forma declarativa e intuitiva, atuando como uma DSL (Domain Specific Language). Internamente, o código JS é convertido e executado por uma *engine* baseada em Go. Ou seja, o Javascript serve como uma linguagem de configuração e definição dos testes.

Supertest, FrisbyJS e Postman

Ao usar ferramentas como Supertest ou FrisbyJS, o Javascript é usado para **fazer chamadas HTTP programáticas**, assim como validar as respostas. No Postman, é usado para manipular dados, gerenciar fluxo e também para as validações das respostas.

Appium

No contexto de testes mobile, o JS pode ser usado com ferramentas como o Appium, em conjunto com frameworks como WebDriverIO ou CodeceptJS. Neste caso, o Javascript é usado para **escrever os scripts que interagem com apps Android/iOS** por meio do Appium Server, que traduz os comandos para os drivers nativos dos sistemas operacionais.

Em resumo:

Contexto	**Papel do Javascript**
WebDriverIO (Selenium)	Comunicação com o WebDriver
K6	Linguagem de domínio
Supertest/Frisby/Postman	Requisições HTTP
Playwright/Cypress	Execução no navegador ou via protocolo
Appium	Comunicação com o Appium Server

Outros frameworks com focos específicos:

- Jest (2011)
- MochaJS (2011)
- Protractor (2013)
- NightwatchJS (2014)
- WebdriverIO (2015)
- TestCafe (2016)
- Cypress (2017)
- Playwright (2020)

O Jasmine, um dos primeiros frameworks populares para testes em JavaScript, ajudou a estabelecer boas práticas para testes automatizados na comunidade. Ele introduziu um estilo de escrita baseado em "*describe*" e "*it*", que facilita a organização e compreensão dos testes.

Essa abordagem influenciou outros frameworks como *Jest* e *Mocha*, que adotaram sintaxes similares para proporcionar uma experiência de desenvolvimento fluida e intuitiva.

Console API

O Console API é uma das principais aliadas no aprendizado de Javascript e é amplamente utilizada por desenvolvedores para auxiliar em tarefas de *debug*. De forma geral, ela serve para exibir valores no *console* durante a execução de um programa em Javascript.

Os principais comandos que vamos utilizar são:

- log
- warn
- error
- table

Constantes e Variáveis

As *variáveis* são um dos conceitos mais fundamentais da programação, pois nos permitem armazenar e manipular dados ao longo da execução de um programa.

No Javascript *moderno*, podemos declarar variáveis de três formas: usando *var*, *let* ou *const*. Cada uma dessas palavras-chave possui particularidades importantes que afetam o escopo, a mutabilidade e comportamento durante a execução do código.

var

A palavra-chave var foi, por muito tempo, a principal forma de declarar variáveis em JavaScript. No entanto, ela possui comportamentos peculiares relacionados ao escopo e ao *hoisting*.

Variáveis declaradas com var têm escopo de função (e não de bloco), o que pode causar confusões, especialmente em estruturas como *if* ou *for*. Apesar de ainda funcionar, seu uso é desencorajado em projetos modernos.

let

Introduzida no ES6, *let* permite declarar variáveis com escopo de bloco, o que torna o comportamento mais previsível e seguro. Uma variável declarada com *let* só existe dentro do “bloco” *{ }* onde foi criada.

Além disso, o JavaScript não permite que a mesma variável *let* seja redeclarada no mesmo escopo, evitando conflitos de nomes e erros difíceis de identificar.

const

Assim como *let*, a palavra-chave *const* também possui escopo de bloco, mas com uma diferença crucial: o valor atribuído não pode ser reatribuído.

Isso significa que a constante precisa ser inicializada no momento da declaração, e não pode ser modificada posteriormente. No entanto, quando *const* é usado para armazenar objetos ou arrays, seus conteúdos podem ser alterados, embora a referência em si permaneça constante.

hoisting

O termo *hoisting* refere-se ao comportamento do JavaScript de "mover" as declarações de variáveis para o topo do escopo antes da execução. No caso do *var*, a variável é “içada” com o valor *undefined*.

Já com *let* e *const*, as declarações também são içadas, mas não inicializadas, resultando em erro se forem acessadas antes de sua linha de declaração.

Tipos de Dados

Os tipos de dados são um conceito fundamental em qualquer linguagem de programação, e no JavaScript, eles desempenham um papel crucial na manipulação e armazenamento de informações. Em JavaScript, os dados

podem ser classificados em dois grandes grupos: **tipos primitivos** e **tipos complexos**.

Os tipos primitivos incluem:

- String
- Number
- Boolean
- null
- undefined
- Symbol

Os tipos complexos incluem:

- Object
- Array
- function

Diferente de outras linguagens de programação que possuem um sistema de tipos mais rígido, o JavaScript é uma linguagem dinamicamente tipada, o que significa que as variáveis não possuem um tipo fixo.

Uma variável pode armazenar um número em um momento e, em seguida, ser atribuída a um texto sem gerar erro.

Embora essa flexibilidade torne o desenvolvimento mais ágil, também pode levar a bugs difíceis de identificar se o código não for bem estruturado.

String

As **strings** representam sequências de caracteres e podem ser definidas usando 3 formas:

- aspas simples (')
- aspas duplas (")
- crase (`) para *template strings*

Diferente de outras linguagens que tratam caracteres individuais como um tipo específico, no JavaScript, uma string com um único caractere ainda é considerada string.

A manipulação de strings é uma das operações mais comuns em programação e conta com métodos como:

- .length
- .split()
- .toLowerCase()
- .toUpperCase()
- .includes()
- .replaceAll()
- .trim()
- .slice()
- .substring()

Number

Os números em JavaScript são do tipo Number, o que significa que não há diferenciação entre inteiros e decimais.

A linguagem também suporta números especiais como *Infinity*, *-Infinity* e *NaN* (Not a Number), que indicam valores indefinidos ou cálculos inválidos.

A partir do ES6, o JavaScript passou a suportar o tipo *BigInt*, permitindo operações com números extremamente grandes, úteis para cálculos financeiros e criptografia.

Boolean

O tipo *Booleano* representa valores lógicos, podendo ser *true* ou *false* - verdadeiro ou falso. Ele é amplamente utilizado em estruturas como if e while, permitindo que o código tome decisões baseadas em condicionais.

Além disso, valores como 0, "", null, undefined e NaN são considerados *false*, enquanto todos os outros valores são *true*. Esse comportamento pode ser explorado para simplificar expressões condicionais no código.

EXERCÍCIOS

Revisão 01:

1. Qual palavra-chave permite declarar uma variável que pode ser alterada depois?
2. Qual palavra-chave é usada para declarar constantes em Javascript?
3. O que significa dizer que uma variável *var* é içada?
4. Quando você usaria *let* em vez de *const*?

Revisão 02:

1. O que o tipo *undefined* representa em Javascript?
2. O que o tipo *null* representa?
3. O que acontece com o tipo de uma string que contém um número, como “42”?

4. Qual a diferença entre Number(“5”) e “5” + 1?

Revisão 03:

1. Quais as 3 formas de declarar uma string em Javascript?
2. O que o método .trim() faz em uma string?
3. Qual o resultado de ‘Dog’.toUpperCase()?
4. Em que situação o uso de *template strings* com crase (`) pode tornar o código mais legível do que concatenação com + ?

Exercício 01:

Hands-on:

Gerador de *tags* de identificação
Crie um script para gerar a etiqueta (tag) de identificação que será presa na coleira de um cachorro no abrigo. O dono irá informar nome, idade, peso, raça e se é adotado ou não.

A tag deve ter:
- O nome em letras maiúsculas
- A raça com a primeira letra maiúscula
- Peso

Exercício 02:

Hands-on:

Validador de nomes para as *tags*

Alguns donos estão registrando os dogs com nomes mal formatados. Crie um script para limpar e padronizar os nomes.

Aplique as formatações e exiba:

- O nome original, como foi cadastrado
- O nome formatado

Extra: aplique uma regra que confira se o nome informado possui apenas uma palavra. Exiba se o nome é válido ou não.

Exercício 03:

Hands-on:

Repita o processo feito em aula e crie um novo repositório no Github. Crie o repositório, conecte ao seu projeto local e depois suba os arquivos.

OPERADORES

Os **operadores** em JavaScript são fundamentais para manipular e comparar dados.

Eles se dividem em diferentes categorias:

- **aritméticos** (+, -, *, /, %, **)
- **de comparação** (==, ===, !=, !==, >, <, >=, <=)
- **lógicos** (&&, ||, !)
- **de atribuição** (=, +=, -=, *=, /=, %=)

Outro aspecto relevante é a **precedência de operadores**, que define a ordem de execução das operações. Por exemplo, multiplicação e divisão têm maior precedência que adição e subtração.

Para garantir que cálculos sejam feitos na ordem desejada, é comum utilizar parênteses (), de forma similar ao uso em cálculos matemáticos. Conhecer a precedência dos operadores ajuda a tornar o código mais previsível.

O JavaScript também conta com operadores especiais, como o operador ternário (? :), que simplifica expressões condicionais em uma única linha.

Em vez de escrever um bloco *if-else*, pode-se usar:

- *condição* ? *valor_se_verdadeiro* : *valor_se_falso*

Isso torna o código mais conciso e legível em muitas situações, especialmente para definir valores de variáveis com base em condições simples.

FUNÇÕES

As funções são um dos conceitos mais importantes em JavaScript, permitindo a organização e reutilização de código. Elas são blocos de código que executam uma tarefa específica e podem ser chamadas sempre que necessário.

O uso de funções torna o código mais modular, facilitando a manutenção e evitando repetições desnecessárias.

No Javascript, existem diferentes formas de declarar funções. A forma mais tradicional é utilizando a palavra-chave *function*, seguida do nome da função e dos parênteses contendo os parâmetros.

Um exemplo básico seria:

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

console.log(saudacao("Júlio"));
// Saída: Olá, Júlio!
```

A partir do ES6, foram introduzidas as *arrow functions*, uma forma mais concisa de escrever funções. Elas são declaradas utilizando a sintaxe:

- *() => {}*

E, são especialmente úteis quando usadas em callbacks e funções anônimas.

Um exemplo básico seria:

```
const saudacao = (nome) => {
        `Olá, ${nome}!`;
}

console.log(saudacao("Júlio"));
// Saída: Olá, Júlio!
```

Além das *funções nomeadas* e das *arrow functions*, o JavaScript também suporta *funções anônimas*, que são funções sem um nome específico.

Elas são frequentemente utilizadas em eventos, callbacks e como argumentos para outras funções:

```
setTimeout(function () {
        console.log("Executando após 2 segundos...");
}, 2000);
```

Outro conceito essencial é o **escopo das funções**. Em JavaScript, o escopo define a acessibilidade das variáveis.

Existem três tipos principais de escopo:

- Escopo global: variáveis declaradas fora de funções podem ser acessadas em qualquer parte do código.
- Escopo de função: variáveis declaradas dentro de uma função só podem ser acessadas dentro dela.
- Escopo de bloco: com *let* e *const*, as variáveis só existem dentro do bloco *{ }* onde foram declaradas.

As funções também podem ter *parâmetros default*, que permitem definir valores padrão caso nenhum argumento seja passado. Isso evita erros e melhora a flexibilidade do código:

```
function boasVindas(nome = "Visitante") {
    return `Bem-vindo, ${nome}!`;
}

console.log(boasVindas());
// Saída: Bem-vindo, Visitante!

console.log(boasVindas("Júlio"));
// Saída: Bem-vindo, Júlio!
```

Outro conceito importante é o *hoisting*, que permite que funções declaradas com *function* sejam chamadas antes de serem definidas no código.

Isso ocorre porque o JavaScript "move" as declarações de funções para o topo durante a execução (similar ao *var*).

```
saudacao();

function saudacao() {
    console.log("Olá, mundo!");
}
```

Além das funções normais, o JavaScript permite a criação de **funções dentro de funções**. Essas funções internas podem acessar variáveis da função externa, criando o que chamamos de *closures*.

```
function contador() {
    let count = 0;
    return function () {
        count++;
            console.log(`Contagem:  ${count}`);
    };
```

```
}

const incrementar = contador();
incrementar(); // Saída: Contagem: 1
incrementar(); // Saída: Contagem: 2
```

Por fim, as funções são amplamente utilizadas na automação de testes, pois permitem criar **métodos reutilizáveis** para validações e ações repetitivas.

No contexto de testes automatizados, as funções ajudam a estruturar os testes de forma modular, separando a lógica de preparação, execução e verificação dos resultados.

Modularização de Código

A modularização de código é uma prática essencial no desenvolvimento de software e na automação de testes, pois assim como as funções, permite organizar o código e facilitar sua reutilização e manutenção no projeto.

Em Javascript, a modularização possibilita dividir o código em arquivos e funções menores, tornando-o mais organizado e escalável.

Existem duas formas principais:

- module.exports / require()
- export / import

Vamos entender um pouco melhor cada uma.

Antes da introdução dos módulos nativos no Javascript (ESModules - ESM), se usavam soluções alternativas como **CommonJS**, que ainda é amplamente usado no NodeJS.

No CommonJS, os módulos são exportados com *module.exports* e importados com *require()*, assim:

arquivo soma.js

```
function soma (a, b) {
        return a + b
}

module.exports = soma
```

arquivo main.js

```
const soma = require('./soma.js')
console.log(soma(2, 3)); // igual a 5
```

Com a introdução do **ES Modules (ESM)** no Javascript moderno, a sintaxe de importação e exportação tornou-se mais padronizada e compatível entre navegadores e NodeJS. Agora, é possível utilizar *export* e *import* para compartilhar funcionalidades entre arquivos.

arquivo soma.js

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

arquivo main.js

```
import { soma } from './soma.js'
console.log(soma(2, 3)); // igual a 5
```

Além das exportações nomeadas, o Javascript permite **exportações padrão**, onde um único valor ou função pode ser exportado de um módulo. Isso facilita a importação quando há apenas um recurso principal no arquivo:

arquivo saudacao.js

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

arquivo main.js

```
import saudacao from './saudacao.js'
console.log(saudacao("Júlio"));
```

A modularização também melhora a separação de responsabilidade, um dos princípios do desenvolvimento de software. Em um projeto de automação de testes é possível organizar o código separando funções auxiliares, configurações ou módulos do sistema, por exemplo.

Outra vantagem da modularização é o reuso de código. Em vez de duplicar trechos de código em diferentes partes do projeto, funções e classes podem ser importadas de módulos reutilizáveis. Isso reduz a redundância e facilita a aplicação de correções e melhorias de forma centralizada.

Em projetos modernos, os gerenciadores de pacotes como **yarn** e **npm** ajudam a integrar pacotes de terceiros. É possível instalar bibliotecas específicas para testes, manipulação de dados, permitindo se concentrar na lógica dos testes sem ter que criar tudo do zero.

A modularização de código não se limita apenas a funções e classes, mas também a estruturação do projeto como um todo. Separar arquivos por categoria ou contexto, como *utils/, services/, tests/,* entre outras, melhora a organização e facilita a colaboração entre membros da equipe.

Testes de Unidade

Os **testes de unidade** são a base da pirâmide de testes e desempenham um papel crucial na garantia da qualidade do software. Eles verificam se pequenas partes isoladas do código, geralmente funções ou métodos, funcionam corretamente. A ideia é garantir que cada unidade do sistema opere conforme o esperado, independentemente de outras partes da aplicação.

Conceitualmente, a principal vantagem dos testes de unidade é que eles permitem identificar erros assim que eles surgem no código. Identificar erros rapidamente reduzem o *custo de correção* de um bug, que quanto mais tardio for identificado no processo de desenvolvimento, mais caro.

Em Javascript existem várias bibliotecas e frameworks para escrever e executar testes de unidade. Entre os mais populares estão: *Jest*, *Mocha* e *Chai*. O *Jest*, por exemplo, é amplamente utilizado por ser rápido e oferecer uma sintaxe intuitiva para escrever testes.

A estrutura básica de um teste de unidade geralmente segue um padrão de 3 etapas, o *Triple A (AAA)*:

- Arrange (Preparação)
- Act (Ação ou Execução)
- Assert (Verificação)

Vamos a um exemplo:

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

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

Neste exemplo, temos uma função *somar* que faz a soma entre dois números; e um teste que verifica se ao executar essa função o resultado será conforme esperado.

Estruturas de Repetição

As estruturas de repetição são utilizadas em programação para executar uma mesma ação diversas vezes, de acordo com uma condição ou com base nos elementos de uma lista. Em programação, é comum se referir a essas

estruturas como *loops*. No JS, as principais estruturas de repetição são: *while*, *for* e *forEach*.

while (condicional)

O *while* é uma estrutura de repetição *condicional*, ou seja, ele executa um bloco de código **enquanto** a condição for verdadeira. Esse tipo de *loop* é útil quando não sabemos exatamente quantas vezes o código precisa ser executado.

É importante garantir que a condição, em algum momento, se torne falsa e "pare" a repetição, para evitar o conhecido *loop infinito*, onde um código é repetido infinitamente.

Exemplo de *while*:

```
let contador = 0;
while (contador < 3) {
   console.log(contador);
   contador++;
}
```

for (contado)

O *for* é uma estrutura clássica e muito utilizada quando sabemos **quantas vezes queremos repetir algo**. Ele possui três partes:

- inicialização
- condição
- incremento

Exemplo de *for*:

```
for (let i = 0; i < 5; i++) {
```

```
    console.log("Valor de i:", i);
}
```

forEach (para listas)

O *forEach* é um método específico para listas (Arrays), utilizado para **percorrer cada item de uma lista** e executar uma ação para cada elemento. Ele é muito utilizado em projetos de automação, especialmente para trabalhar com dados. A principal vantagem do *forEach* é a legibilidade.

Exemplo de *forEach*:

```
const frutas = ["Maçã", "Banana", "Laranja"];
frutas.forEach((fruta) => {
    console.log("Fruta:", fruta);
});
```

EXCEÇÕES

Em qualquer aplicação, erros podem acontecer. Em Javascript, é possível tratar esses erros de forma controlada, evitando que falhas interrompam toda a execução do programa. Para isso, utilizamos estruturas como *try*, *catch*, *finally* e o comando *throw*.

try…catch

O bloco *try* é utilizado para envolver o código que **pode gerar um erro**. Caso ocorra, o Javascript interrompe a

execução do *try* e pula para o bloco *catch*, onde o erro pode ser tratado de forma amigável.

Exemplo:

```
try {
    let resultado = 10 / 0;
    console.log("Resultado:", resultado);
} catch (erro) {
    console.log("Erro capturado:", erro.message);
}
```

finally

O bloco *finally* é opcional, mas útil. Ele será executado **sempre**, tendo ocorrido um erro ou não. É geralmente usado para fechar conexões ou exibir logs, por exemplo.

Exemplo:

```
try {
    console.log("Executando...");
} catch (erro) {
    console.log("Erro!");
} finally {
    console.log("Finalizado.");
}
```

throw

Com o comando *throw*, podemos **lançar erros** de forma proposital. Isso é útil quando queremos para a execução caso alguma condição não seja atendida e fornecer uma mensagem específica sobre o problema.

Exemplo:

```
function verificarIdade(idade) {
    if (idade < 18) {
        throw new Error("Idade mínima não atingida.");
    }
    console.log("Acesso liberado.");
}
```

PROMISES E ASSINCRONISMO

Em Javascript, muitas operações demoram para acontecer, como buscar dados de uma API, ler arquivos ou executar consultas em um banco de dados. Para que o programa não **trave** enquanto espera, usamos mecanismos assíncronos como *Promises* e *async/await*.

Promises

Uma *Promise* é um objeto que representa **uma operação que ainda está em andamento**, mas que será resolvida no futuro com sucesso ou erro. Usamos *then()* para lidar com o sucesso e *catch()* para tratar erros.

Exemplo:

```
const promessa = new Promise((resolve, reject) => {
    let sucesso = true;
```

```
    setTimeout(() => {
        sucesso ? resolve("Tudo certo!") : reject("Algo
deu errado.");
    }, 2000);
});

promessa
    .then((resposta) => console.log(resposta))
    .catch((erro) => console.log("Erro:", erro));
```

async/await

A sintaxe *async/await* foi criada para deixar o código assíncrono mais fácil de ler e escrever. Usamos o *await* para esperar o resultado de uma Promise e *try…catch*, visto anteriormente, para tratar erros.

Exemplo:

```
async function carregarDados() {
    try {
        const resposta = await
fetch("https://api.exemplo.com");
        const dados = await resposta.json();
        console.log(dados);
    } catch (erro) {
        console.log("Erro ao buscar dados:",
erro.message);
    }
}
```

```
carregarDados();
```

Esses recursos são essenciais em testes automatizados, pois interações com a UI, chamadas de APIs e outras ações comuns geralmente envolvem algum tipo de espera.

Padronização e Código Limpo

Na prática da programação, especialmente em times e projetos de médio a longo prazo, escrever código que funcione não é o suficiente. Precisamos garantir que ele seja legível, organizado e padronizado.

Nisso, entram os conceitos de padronização e *código limpo.*

Código Limpo (Clean Code)

Clean Code é uma abordagem que valoriza o código fácil de entender, manter e evoluir. Um “código limpo” evita nomes genéricos, comentários desnecessários e blocos confusos. Cada função deve fazer apenas uma coisa e fazer isso bem. O código deve ser autoexplicativo enquanto lido.

```
// Difícil de entender
function x(a, b) {
   return a + b;
}

// Limpo e autoexplicativo
```

```
function somarNumeros(primeiroNumero, segundoNumero)
{
   return primeiroNumero + segundoNumero;
}
```

Existem algumas recomendações quanto a como manter um código limpo, muitas delas foram compiladas no livro de mesmo nome: Clean Code de Robert C. Martin, em 2008.

REVISÃO

Revisão 01:

1. Qual palavra-chave permite declarar uma variável que pode ser alterada depois?
2. Qual palavra-chave é usada para declarar constantes em Javascript?
3. O que significa dizer que uma variável *var* é içada?
4. Quando você usaria *let* em vez de *const*?

Revisão 02:

1. O que o tipo *undefined* representa em Javascript?
2. O que o tipo *null* representa?
3. O que acontece com o tipo de uma string que contém um número, como “42”?
4. Qual a diferença entre Number(“5”) e “5” + 1?

Revisão 03:

1. Quais as 3 formas de declarar uma string em Javascript?

2. O que o método .trim() faz em uma string?
3. Qual o resultado de ‘Dog’.toUpperCase()?
4. Em que situação o uso de *template strings* com crase (`) pode tornar o código mais legível do que concatenação com + ?

Revisão 04:

1. Qual operador compara valores e tipos em Javascript?
2. Qual a diferença entre `==` e `===` ?
3. Qual é o resultado de 5 == '5', true ou false?
4. E 5 === '5', true ou false?
5. O que retorna a expressão 10 > 5?
6. O que retorna a expressão 0 == false?
7. Por que é mais seguro usar `===` do que `==`?

Revisão 05:

1. Quais são os 3 operadores lógicos em Javascript?
2. Qual o resultado da expressão true && false ?
3. E da expressão true || false ?
4. O que o operador `!` faz com um valor booleano?

Revisão 06:

1. Quais são os principais operadores aritméticos em JS?
2. O que retorna a expressão 10 + 5 ?
3. O que retorna a expressão 7 % 2 ?
4. O que retorna a expressão 4 * 2 + 3 ?
5. O que retorna a expressão +"5" * 2 ?

Revisão 07:

1. Qual a estrutura básica do operador ternário?
2. O que retorna a expressão: true ? "sim" : "não" ?
3. Qual o valor de `x` após a expressão: let x = 1; x++ ?

EXERCÍCIOS

Exercício 01:

Hands-on:

Gerador de *tags* de identificação
Crie um script para gerar a etiqueta (tag) de identificação que será presa na coleira de um cachorro no abrigo. O dono irá informar nome, idade, peso, raça e se é adotado ou não.

A tag deve ter:
- O nome em letras maiúsculas
- A raça com a primeira letra maiúscula
- Peso

Exercício 02:

Hands-on:

Validador de nomes para as *tags*

Alguns donos estão registrando os dogs com nomes mal formatados. Crie um script para limpar e padronizar os nomes.

Aplique as formatações e exiba:

- O nome original, como foi cadastrado
- O nome formatado

Extra: aplique uma regra que confira se o nome informado possui apenas uma palavra. Exiba se o nome é válido ou não.

Exercício 03:

Hands-on:

Validador de idade mínima para adoção

Crie um script que verifique se um dog pode ser adotado com base na idade mínima definida, por exemplo, 2 anos.

Use os operadores adequados e exiba:

- Nome do dog
- Idade
- Se está apto ou não para adoção

Extra: aplique uma regra com operador lógico para permitir que se o cão for de pequeno porte, pode ser adotado independente da idade.

Exercício 04:

Calculadora de ração diária

Crie um script que receba o peso do dog em **kg** + estoque atual de ração em **gramas**. Calcule a quantidade diária de ração com base na seguinte fórmula:

Gramas por dia = peso x 30 gramas

Exiba:

- Nome do dog
- Peso
- Quantidade de ração recomendada por dia
- Quantos dias o estoque atual vai durar

Exercício 05:

Classificador de porte automático

Muitos tutores não fazem ideia do porte de seu Dog. Crie um script que classifique o porte com base no peso de forma simplificada, sendo:

- Até 10kg -> Pequeno; acima disso -> Médio

Use o operador ternário para determinar o porte. Exiba:

- Nome
- Peso
- Porte classificado

Extra: transforme a lógica em uma função.

Exercício 06:

Plano de atividades para o Pet

Crie um script que define o plano de atividades para os dogs durante a estadia. O script vai receber o porte (pequeno, médio ou grande); e o tempo disponível para a atividade representado em minutos. Exemplo:

Pantera - Médio - 45

Use uma condicional para decidir o tipo de atividade com base no porte:

- pequeno -> brincar dentro de casa
- médio -> caminhada no quarteirão
- grande -> correr no parque
- qualquer outro -> porte inválido

Depois, uma condicional para ajustar a mensagem de acordo com o tempo:

- menor que 15 -> "atividade rápida: [atividade]"
- de 15 a 30 -> "tempo ideal: [atividade]"
- acima de 30 -> "hora da diversão: [atividade]

Exercício 07:

Contador de *satisfação* de passeio

Crie um script que avise quando o Dog já passeou o suficiente. Para saber, vamos considerar que ele se sentirá satisfeito somente a partir de 5 voltas na quadra.

Use a estrutura de repetição *while*.

Exiba:

- Qual o número da volta atual
- Quando o *dog* estiver satisfeito

Extra: transforme a lógica em uma função (da forma que conseguir)

Exercício 08:

Controle de *petiscos*

Crie um script que receba uma quantidade de *petiscos* e dê 1 por vez até o *dog* estiver satisfeito.

Use a estrutura de repetição *for contado*.

Exiba:

- Cada vez que um petisco for entregue
- Quando o *dog* estiver satisfeito **(que é quando terminar os petiscos)**

Extra: transforme a lógica em uma função (da forma que conseguir)

Exercício 09:

Entregador de brinquedos para os dogs

Dado que você tenha uma lista de brinquedos com: Bola, Osso, Corda, Sino
Crie um script que passe pela lista de brinquedos e *entregue* um por vez.
Exiba o nome de cada brinquedo que for entregue.

Use a estrutura de repetição *forEach*.

Extra: transforme a lógica em uma função (da forma que conseguir)

Exercício 10:

Distribuidor de petiscos

Você tem uma lista de cães: Pantera, Luna e Thor
Crie um script que "entregue" 1 petisco para cada cão da

lista.
Exiba uma mensagem como: "Entregando petisco para Pantera"

Mantenha os **dados** e a **função** em arquivos separados; use modularização para organizar.

Exercício Extra (valendo pontos):

Crie uma função que pega uma lista de inteiros e strings e retorna uma nova lista sem as strings.

Exercício Extra (valendo pontos):

Retorne o número de vogais no texto fornecido. Consideraremos **a**, **e**, **i**, **o** e **u** como vogais para este Kata. O texto de entrada conterá apenas letras minúsculas e/ou espaços. Letras acentuadas não fazem parte desse desafio.

REFERÊNCIAS

1. **Eloquent Javascript**
   https://eloquentjavascript.net/00_intro.html

2. **Definition of JavaScript**
   https://developer.mozilla.org/pt-BR/docs/Web/JavaScript

3. **Git - Reference**
   https://git-scm.com/docs