JavaScript [1]

Um Breve Tutorial

Gabriel P. Silva[2]

18 de Outubro de 2024

[2]gabriel@ic.ufrj.br

Dedicado aos meus filhos, Vinícius, Danilo, Caio Vítor e a Madalena, minha constante inspiração.

Conteúdo

Lista de Figuras

Lista de Tabelas

Prefácio

Este é um tutorial muito simples com o objetivo de apresentar os conceitos básicos da linguagem JavaScript. É uma obra que pretendemos tenha um caráter dinâmico, evoluindo e sendo aprimorada ao longo do tempo.

O JavaScript é uma linguagem de programação que permite implementar funções complexas em páginas da web. Toda vez que uma página faz mais do que apenas exibir informações estáticas - exibindo periodicamente atualizações de conteúdo, mapas interativos ou gráficos 2D/3D animados, ou ainda rolando caixas de escolha de vídeo, e assim por diante - você pode apostar que o JavaScript provavelmente está envolvido.

Ao contrário do que alguns possam imaginar, a linguagem JavaScript tem pouca ou nenhuma relação com a linguagem Java. A linguagem JavaScript é interpretada em tempo de execução, ou seja, ao contrário do Java, não é gerado nenhum código objeto, nem mesmo para uma máquina virtual, no processo de sua execução.

Algumas características marcantes do JavaScript são: é uma linguagem orientada a objeto, onde não há nenhuma distinção entre tipos de objetos. A herança é feita através do mecanismo de protótipo, e as propriedades e métodos podem ser adicionados a qualquer objeto dinamicamente. Os tipos de dados das variáveis não são declarados (tipagem dinâmica, tipagem fraca).

É uma linguagem que tem evoluído bastante em diversos aspectos, tendo uma aceitação e utilização cada vez maior, ao longo dos anos. Apesar de ser uma linguagem interpretada, o uso de técnicas empregadas na sua interpretação, como *just in time*, tem feito com que seu desempenho se aproxime bastante daquele obtido quando se faz uso de código nativo.

Neste tutorial apresentaremos os conceitos básicos da linguagem JavaScript, além do uso do Node.js®, que é máquina de execução para o JavaScript construída com base na versão 8 máquina de execução do navegador Chrome. O Node.js usa um modelo de E/S não bloqueante, dirigido a eventos, o que o torna leve e eficiente. O ecossistema de pacotes do Node.js, npm, que é o maior ecossistema de bibliotecas de código aberto no mundo, também será apresentado neste tutorial.

Finalmente, apresentamos brevemente o JSON (JavaScript Object Notation), que é um formato leve para a troca de dados, que é fácil para leitura e escrita por humanos, além de ser fácil para ser analisado e gerado por máquinas.

Os exemplos apresentados neste tutorial esstão disponíveis em https://github.com/gpsilva2003/JavaScript.

Agradecimentos

- Ao meus alunos, pela ajuda na criação desta obra com seus comentários e sugestões.

Gabriel P. Silva

http://gabrielsilva.rio.br/

1

O que é JavaScript?

1.1 Introdução

O JavaScript é uma linguagem de programação interpretada, de alto nível, dinâmica e não-tipada. É amplamente utilizada para fornecer interatividade e dinamismo a sites, especialmente quando aplicada a documentos HTML. O JavaScript, junto com HTML e CSS, é uma das três principais tecnologias da World Wide Web. Seu papel vai além da interatividade, possibilitando também a manipulação dinâmica de elementos da página, de modo que os desenvolvedores possam criar interfaces mais ricas e responsivas. A maioria dos sites modernos utilizam JavaScript, e todos os navegadores compatíveis suportam a linguagem nativamente, sem a necessidade de extensões ou *plugins*.

A linguagem foi criada por Brendan Eich em 1995, durante seu trabalho na Netscape e, inicialmente, tinha o nome de Mocha, que depois evoluiu para LiveScript e finalmente JavaScript. Além disso, Eich é co-fundador do projeto Mozilla, da Fundação Mozilla e da Mozilla Corporation. A padronização do JavaScript ocorreu por meio da especificação ECMAScript, publicada pela Ecma International. A primeira versão foi lançada em 1997 como ECMA-262. Desde então, a especificação continua a evoluir, incorporando novos recursos à linguagem para atender às necessidades modernas de desenvolvimento *web*. Apresentamos seguir um resumo da evolução histórica do padrão:

1. **ECMAScript 1 (1997)**: primeira versão oficial da especificação JavaScript. Estabeleceu a base para a linguagem.

2. **ECMAScript 2 (1998**): pequenas correções de erros e alinhamento com o padrão ISO/IEC.

3. **ECMAScript 3 (1999)**: Uma das versões mais significativas, introduziu o RegEx para tratamento de expressões regulares, o tratamento de exceções com ‘try/catch‘ e melhorias na manipulação de *strings*, *arrays* e datas.

4. **ECMAScript 4 (Cancelada)**: a versão 4 foi planejada para ser uma grande atualização com várias mudanças radicais, mas foi cancelada devido a desacordos entre os desenvolvedores.

5. **ECMAScript 5 (2009)**: introduziu vários recursos importantes como *strict mode* (Modo estrito), métodos adicionais para objetos e *arrays*, suporte nativo ao JSON, versão de longa duração, amplamente suportada.

6. **ECMAScript 6 (2015) - ECMAScript 2015 (ES6)**: Uma das atualizações mais importantes desde a versão 3, adiconando covas funções como **let**, **const**, **arrow**, além de classes, módulos, iteradores, geradores, entre outros.

7. **ECMAScript 2016 (ES7)**: operador de exponenciação (‘**‘), e métodos como ‘Array.prototype.includes()

8. **ECMAScript 2017 (ES8)**: Introduziu‘async/await‘, métodos ‘Object.entries()‘, ‘Object.values()‘ e ‘Object.getOwnPropertyDescriptors()‘.

9. **ECMAScript 2018 (ES9)**: melhorias em *async iterators* e *generators*, introdução de *rest/spread* para objetos.

10. ECMAScript 2019 (ES10): novos métodos, como ‘Array.prototype.flat()‘ e ‘Array.prototype.flatMap()‘, ‘Object.fromEntries()‘, e melhorias no tratamento de *strings* e símbolos.

11. **ECMAScript 2020 (ES11)**: Introduziu o operador de coalescência nula (‘?¿), optional chaining (‘?.‘), ‘BigInt‘ para suporte a números grandes.

12. **ECMAScript 2021 (ES12)**: inclui os métodos ‘String.prototype.replaceAll()‘, operadores lógicos ‘&&=‘ e ‘||=‘, e o método ‘Promise.any()‘.

13. **ECMAScript 2022 (ES13)**: adição de métodos ‘Array.prototype.at()‘, ‘Top-level await‘ (espera no nível superior), melhorias nas classes privadas com campos e métodos.

14. **ECMAScript 2023 (ES14)**: atualizações contínuas com adição de pequenos novos recursos e melhorias de desempenho.

Desde o ECMAScript 2015 (ES6), a ECMAScript tem adotado um ciclo de atualizações anuais, em vez de grandes revisões com longos intervalos.

1.2 Versatilidade e Usos

O JavaScript é uma linguagem incrivelmente versátil. Desde funcionalidades simples, como a criação de galerias de imagens e botões interativos, até aplicações mais complexas como jogos, gráficos animados 2D e 3D, e até sistemas baseados em banco de dados. O JavaScript permite a criação de uma ampla gama de funcionalidades graças à sua capacidade de trabalhar tanto

no *front-end* (lado do cliente) quanto no *back-end* (lado do servidor), por meio de ambientes como o Node.js. Embora o núcleo do JavaScript seja compacto, ele é complementado por uma vasta gama de ferramentas, APIs e bibliotecas que estendem suas capacidades:

- **APIs Web Integradas**: JavaScript oferece acesso a diversas APIs integradas nos navegadores, como a manipulação dinâmica de HTML (DOM), configuração de estilos CSS, e captura e manipulação de mídias, como vídeos da camêra. Além disso, APIs modernas como o WebGL permitem a criação de gráficos 3D, e o Web Audio API habilita manipulações sofisticadas de áudio.

- **APIs de Terceiros**: APIs como as do Twitter, Facebook e Google Maps permitem que os desenvolvedores integrem facilmente funcionalidades e dados de outros serviços em seus sites.

- **Frameworks e Bibliotecas**: Ferramentas como React, Angular, Vue.js e bibliotecas como jQuery e D3.js facilitam o desenvolvimento de aplicações, permitindo que desenvolvedores construam interfaces complexas de maneira eficiente e escalável.

Para se iniciar com o JavaScript tudo o que é necessário é um navegador moderno ou instalar o ambiente Node.js. Ou seja, é muito fácil. Existe uma ferramenta incorporada no Firefox que é muito útil para experimentações com o JavaScript: A 'Console Web'.

A Console *Web* mostra informações sobre a página atualmente carregada, tais como como solicitações de rede, códigos JavaScript e CSS e mensagens de erro. Também inclui uma linha de comando que pode ser usada para executar expressões de JavaScript. Para abrir a console do navegador, utilize a combinação de teclas <Ctrl+Shift+I> no Windows e Linux ou <Cmd-Option-K> no Mac, ou selecione "Mais Ferramentas" no menu "Ferramentas de Desenvolvovimento" do Firefox.

Expressões JavaScript podem ser interpretadas em tempo real no Firefox usando o interpretador fornecido pela Console Web. Ele possui dois modos:

- **Entrada em uma única linha**:

  Para inserir expressões no modo de uma única linha, digite a expressão no *prompt* e pressione *Enter*. Para inserir expressões de várias linhas, pressione *Shift + Enter* após digitar cada linha, e depois pressione *Enter* para executar todas as linhas inseridas.

- **Entrada em várias linhas**:

  Para entrada de várias linhas, clique no ícone de "painel dividido"no lado direito do campo de entrada de uma única linha, ou pressione Ctrl + B (Windows/Linux) ou Cmd + B (macOS). O painel de edição de várias linhas será aberto no lado esquerdo do Console da Web.

2

Características

2.1 Introdução

Embora o JavaScript seja uma linguagem interpretada, ela pode alcançar um desempenho muito bom graças a várias técnicas e tecnologias avançadas. Aqui estão alguns fatores que contribuem para o desempenho eficiente do JavaScript:

1. **Engenho JavaScript Otimizado**: Os navegadores modernos utilizam engenhos JavaScript altamente otimizados, como o V8 do Google Chrome, o SpiderMonkey do Firefox e o JavaScriptCore (ou Nitro) da Apple. Esses motores são projetados para compilar o código JavaScript em código de máquina nativo, o que melhora significativamente a velocidade de execução.

2. Compilação Just-In-Time (JIT): Em vez de interpretar o código linha por linha, muitos engenhos JavaScript utilizam a compilação JIT. Isso significa que o código é convertido em código de máquina durante a execução, permitindo otimizações que melhoram o desempenho. O JIT pode aplicar várias otimizações, como a eliminação de código redundante e a *inlining* de funções.

3. Otimizações Baseadas em Tipos: os motores modernos de JavaScript podem otimizar o código com base na análise dos tipos de dados que o código manipula. Eles ajustam o código e o *layout* da memória para maximizar a eficiência. Por exemplo, se o engenho detecta que uma variável é sempre um número, ele pode otimizar as operações matemáticas para essa suposição.

4. Garbage Collection Eficiente: O gerenciamento de memória é crucial para o desempenho. Os engenhos JavaScript implementam técnicas avançadas de *garbage collection* (coleta de lixo) para gerenciar e liberar memória de forma eficiente. Isso ajuda a minimizar o impacto da alocação e liberação de memória sobre o desempenho do programa.

5. Execução Assíncrona e Concorrência: O JavaScript usa um modelo de concorrência baseado em eventos, onde as operações assíncronas são gerenciadas através de *callbacks*, "promessas"e ‘async/await‘. Isso permite que o JavaScript execute tarefas de longa duração sem bloquear a *thread* principal, melhorando a capacidade de resposta e o desempenho geral.

6. Otimizações de Código e Técnicas de Profiling: Os desenvolvedores podem utilizar técnicas de *profiling* e ferramentas de análise para identificar gargalos e otimizar o código JavaScript. Os engenhos JavaScript também aplicam otimizações de execução com base no perfil de execução do código.

7. Estratégias de Cache: Os engenhos de JavaScript frequentemente usam *caches* para armazenar o código compilado e os resultados das operações. Isso reduz a necessidade de recalcular resultados e melhora o desempenho geral.

O JavaScript possui uma sintaxe intencionalmente similar a linguagens como Java e C++, o que facilita a curva de aprendizado para desenvolvedores familiarizados com essas linguagens. No entanto, seu propósito e funcionamento diferem significativamente. A sintaxe do JavaScript é relaxada para permitir que seja utilizada como um linguagem de *scripting* fácil de ser usada. Por exemplo, uma variável não precisa ter seu tipo declarado, nem os tipos são associados com propriedades, e as funções definidas não precisam ter suas declarações aparecendo textualmente antes de uma chamada para elas.

No contexto de desenvolvimento web, o JavaScript é particularmente útil por sua capacidade de manipular o conteúdo e a estrutura de documentos HTML e estilizar páginas através do CSS. Abaixo estão algumas das funcionalidades específicas que o JavaScript oferece no ambiente do navegador:

- **Modificar Conteúdo HTML**: O JavaScript pode alterar dinamicamente o conteúdo de elementos HTML, como textos, imagens e links. Por exemplo, ao acessar e modificar o DOM (Document Object Model), você pode atualizar o conteúdo de um parágrafo ou substituir o texto de um título de maneira interativa e em tempo real.

- **Modificar Atributos HTML**: A linguagem permite modificar atributos de elementos HTML, como o src de uma imagem ou o href de um link. Isso é útil para criar experiências de usuário dinâmicas, como alterar a imagem exibida ao passar o mouse sobre um ícone ou mudar o destino de um link com base na interação do usuário.

- **Remover Elementos e Atributos HTML Existentes**: JavaScript pode excluir completamente elementos do DOM ou remover atributos específicos de um elemento, permitindo que desenvolvedores controlem quais partes do conteúdo são exibidas ou ocultadas sem a necessidade de recarregar a página.

- **Adicionar Novos Elementos e Atributos HTML**: Além de remover elementos, é possível adicionar novos elementos HTML ao documento de forma dinâmica. Isso pode ser utilizado para criar novos botões, seções de texto ou qualquer outro elemento visual de forma programática. Novos atributos também podem ser adicionados a elementos existentes, possibilitando maior flexibilidade na personalização do comportamento da página.

- **Modificar Estilos CSS**: O JavaScript pode alterar diretamente o estilo de elementos HTML, modificando propriedades CSS, como cores, tamanhos, margens e muito mais. Isso permite criar animações, aplicar efeitos visuais em resposta a eventos, ou até mesmo mudar completamente o layout de uma página com base na interação do usuário.

- **Esconder Elementos HTML**: Com JavaScript, elementos HTML podem ser ocultados da visualização ao ajustar a propriedade display ou visibility do CSS. Isso é útil em cenários como o controle de menus suspensos, onde você deseja ocultar ou mostrar uma lista de opções ao clicar em um botão.

- **Mostrar Elementos HTML Ocultos**: Da mesma forma, elementos previamente ocultos podem ser exibidos novamente em resposta a eventos do usuário, como um clique de botão ou uma ação específica. Isso oferece uma maneira interativa de controlar o fluxo de informações na página, mantendo uma interface limpa e funcional.

Um exemplo simples do uso dessas funcionalidades seria um botão que, ao ser clicado, altera o texto de um parágrafo, esconde uma imagem e exibe uma nova, além de modificar a cor do fundo de um elemento específico. Tudo isso pode ser feito em tempo real, sem recarregar a página.

```
<button onclick="alterarPagina()">Clique aqui</button>
<p id="texto">Texto original.</p>
<img id="imagem" src="imagem1.jpg" alt="Imagem">
<script>
  function alterarPagina() {
    document.getElementById("texto").innerHTML = "Texto modificado!";
    document.getElementById("imagem").style.display = "none";
    document.body.style.backgroundColor = "#f0f0f0";
  }
</script>
```

Este exemplo demonstra como o JavaScript permite alterar o conteúdo, ocultar elementos e modificar estilos com apenas algumas linhas de código. Essas funcionalidades são fundamentais para criar interfaces de usuário dinâmicas e interativas.

2.2 HTML DOM

O HTML DOM (Document Object Model) é uma interface de programação que permite aos desenvolvedores acessar, manipular e modificar a estrutura de documentos HTML e XML de maneira programática. Essencialmente, o DOM representa a página web como uma árvore de nós, onde cada elemento HTML é um nó na árvore, com a estrutura que pode ser vista na Figura 2.1. Isso possibilita o uso de linguagens de programação como JavaScript para interagir com a estrutura e o conteúdo de uma página de forma dinâmica.

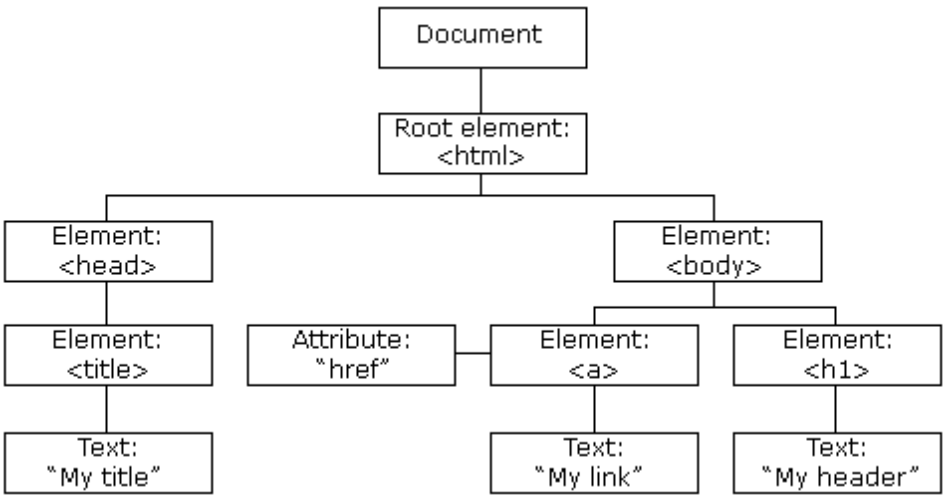

Figura 2.1: DOM HTML

As principais características do HTML DOM são:

- **Modelo de Objeto**: O DOM é um modelo de objeto que representa o documento HTML como uma hierarquia de nós, onde cada elemento, atributo e texto é representado por um objeto específico. Por exemplo, uma tag seria um objeto de nó do tipo Element, enquanto o texto contido dentro de um parágrafo seria um objeto de nó do tipo Text.
- **Acessibilidade e Manipulação**: Através do DOM, você pode acessar e modificar qualquer elemento HTML da página. Isso significa que, por meio de JavaScript, você pode alterar o conteúdo, o estilo e até mesmo remover ou adicionar elementos dinamicamente.
- **Estrutura em Árvore**: O documento HTML é representado como uma estrutura em árvore, onde o nó raiz é o próprio documento, e cada nó descendente representa um elemento HTML. Isso permite percorrer a árvore para acessar elementos específicos com base na sua relação hierárquica (pais, filhos, irmãos, etc.).
- **Interatividade**: O DOM permite que você adicione interatividade à página web. Você pode, por exemplo, associar eventos a elementos (como cliques em botões), alterar propriedades e até mesmo criar animações e transições.

- **Padronização**: O DOM é padronizado pelo W3C (World Wide Web Consortium), o que significa que ele deve funcionar de maneira consistente em todos os navegadores que seguem esse padrão. Isso garante que os scripts que manipulam o DOM se comportem de maneira previsível em diferentes ambientes.

Alguns objetos HTML importantes utilizados no JavaScript são elencados a seguir:

- **NAVIGATOR**: O objeto navegador contém informações sobre o navegador utilizado.
- **DOCUMENT**: Quando um documento HMTL é carregado no navegador, ele se torna um objeto DOCUMENT. O objeto DOCUMENT é o nó raiz do documento HTML, é uma parte do objeto WINDOW e pode ser acessado como window.document.
- **WINDOW**: O objeto WINDOW representa uma janela aberta em um navegador. Se um documento contém "frames", o navegador cria um objeto WINDOW para o documento HTML e um objeto WINDOW adicional para cada "frame".
- **CONSOLE**: O objeto CONSOLE permite acesso à console de depuração do navegador.
- **FORM**: O objeto formulário do JavaScript é uma propriedade do objeto DOCUMENT, que corresponde a um formulário de entrada HTML construído com a etiqueta FORM. Um formulário pode ser enviado chamando o método de submissão (*submit()*) do JavaScript ou clicando no botão enviar do formulário.
- **DATE**: O objeto DATE permite armazenamento básico e recuperação de datas e horários.

Os objetos tem **propriedades** e **métodos** associados, sendo que as **propriedades** são características particulares dos objetos e os **métodos** são funções que podem ser aplicadas para criar, remover, modificar as propriedades associadas aos objetos e os próprios objetos em si.

Por exemplo, um objeto FORM representa um elemento <form> em HTML. Você pode acessar um elemento <form> com o uso do método getElementById():

```
var x = document.getElementById("myForm");
```

Você pode criar um elemento <form> usando o método document.createElement():

```
var x = document.createElement("FORM");
```

Alguns exemplos de propriedades de um elemento <form> são o *name, enctype, target*, etc. Exemplos de métodos aplicáveis a um elemento <form> são *reset()* (reinicia um formulário) e *submit()* (envia um formulário).

O JavaScript permite ainda a captura e gerenciamento de eventos, tais como: *onClick, onSubmit, onMouseOver e onMouseOut*, que serão vistos em mais detalhes no Capítulo 5.

Abaixo está um exemplo simples de como o DOM pode ser utilizado para alterar o conteúdo de um elemento HTML usando JavaScript:

```
<!DOCTYPE html>
<html>
<head>
  <title>Exemplo DOM</title>
</head>
<body>
<p id="paragrafo">Este é um parágrafo.</p>
<button onclick="alterarTexto()">Clique para alterar o texto</button>
<script>
  function alterarTexto() {
    document.getElementById("paragrafo").innerHTML = "O texto foi alterado!
    ";
  }
</script>
</body>
</html>
```

Exemplo DOM

Neste exemplo o documento HTML é representado como uma árvore de elementos (DOM). O *document.getElementById("paragrafo")* é utilizado para acessar o nó do parágrafo específico. O conteúdo do parágrafo é modificado por meio do método *innerHTML*.

2.3 Entrada e saída em JavaScript

A interação com o usuário em JavaScript pode ser realizada de várias formas, dependendo do ambiente em que o código está sendo executado. As principais formas de entrada e saída variam entre o ambiente do navegador e o ambiente de execução no Node.js ou em consoles de desenvolvedores. No navegador, métodos como window.prompt() e window.alert() são utilizados para capturar e exibir informações, respectivamente. No ambiente Node.js ou em consoles de desenvolvedores, o método **console.log()** é o mais comum para saída, enquanto a captura de entrada geralmente requer o uso de módulos adicionais, como o readline. A seguir, detalharemos como isso ocorre em cada um desses ambientes.

2.3.1 No navegador

No ambiente do navegador, JavaScript oferece métodos nativos para interagir com o usuário por meio de caixas de diálogo que aparecem na tela.

- **window.prompt()**: Este método exibe uma caixa de diálogo que solicita ao usuário a entrada de uma *string*. Pode ser utilizado para capturar dados diretamente do usuário, como um nome, uma senha ou qualquer outro valor. A função retorna o valor inserido pelo usuário ou *null* caso o usuário cancele a operação.

  Exemplo:

```
let nome = window.prompt("Digite o seu nome:", "");
if (nome !== null) {
    console.log("Nome digitado: " + nome);
}
```

  Neste exemplo, o navegador exibirá uma caixa de diálogo pedindo ao usuário que insira seu nome. O valor digitado é então armazenado na variável **nome**.

- **window.alert()**: Este método exibe uma caixa de diálogo simples com uma mensagem e um botão "OK". Geralmente é utilizado para alertar o usuário sobre algum evento ou situação, como um erro ou uma confirmação.

  Exemplo:

```
    window.alert("Senha inválida.");
```

  Aqui, o navegador exibirá uma caixa de diálogo informando ao usuário que a senha inserida é inválida.

```
<html>
<head>
<script language="JavaScript">
var nome = window.prompt("qual o seu nome ?", " ");
window.alert("Oi " + nome);
</script>
</head>
<body bgcolor=white>
<h2 align=center>Esta é a minha página.<hr></h2>
<script language="JavaScript">
document.write("<h3>seja bem vindo à minha página, ");
document.write(nome, "</h3><p><hr>");
</script>
</body>
</html>
```

Window Prompt

Outro exemplo simples, combinando os dois métodos anteriores, que faz a personalização da página com os dados informados pelo usuário.

A seguir um exemplo mais completo, que utiliza algumas facilidades do JavaScript, como a possibilidade de capturar eventos com o método *addEventListener()*. Observamos também a referência aos elementos HTML com o uso do método *getElementById()*.

```
<html>
<meta charset="UTF-8">
<head>
<title>Alô Mundo</title>
</head>
<body>

Nome: <input id="nome">
Sobrenome: <input id="sobrenome">
<button id="alo">Diga Alô!</button>
<hr>

<div id="result"></div>
<script>
function diga_alo() {
    var fname = document.getElementById('nome').value;
    var lname = document.getElementById('sobrenome').value;
    var html = 'Alô <b>' + fname + '</b> ' + lname;
    document.getElementById('result').innerHTML = html;
}

document.getElementById('alo').addEventListener('click', diga_alo);
</script>
</body>
</html>
```

Interação com Usuário

2.3.2 Na console do node.js ou navegador

Quando JavaScript é executado fora do navegador, como em um ambiente de console (ou terminal) no Node.js, as formas de entrada e saída são um pouco diferentes. Em vez de janelas de diálogo, você geralmente utiliza métodos que escrevem diretamente na saída padrão (console). A seguir um exemplo de como imprimir mensagens de uma forma bem simples na console.

- **console.log**: Este método imprime uma mensagem no console do navegador ou do terminal. Ele é amplamente utilizado para depuração e exibição de informações enquanto o código está sendo executado.

```
    console.log('Alô!');
```

Isso exibirá a string "Alô!" no console, seja no navegador (console de desenvolvedor) ou no terminal se estiver utilizando Node.js.

Para realizar a entrada de dados há várias possibilidades. A seguir indicamos um exemplo com uso do módulo *prompt*.

```
var prompt = require('prompt');
  prompt.start();

  prompt.get(['usuario', 'email'], function (erro, resultado) {
    if (erro) { return onErr(erro); }
    console.log('Entrada pela console recebida:');
    console.log('  Usuário: ' + resultado.usuario);
    console.log('  Email: ' + resultado.email);
  });

  function onErr(erro) {
    console.log(erro);
    return 1;
  }
```

Entrada com Prompt

Uma outra possibilidade é a utilização do módulo *readline*, que permite capturar entrada diretamente do terminal. Exemplo de uso básico no Node.js:


```
const readline = require('readline');
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('Digite o seu nome: ', (nome) => {
    console.log(`Olá, ${nome}!`);
    rl.close();
});
```

Entrada com Readline

Neste exemplo, utilizamos o módulo **readline** para solicitar uma entrada ao usuário via terminal e, em seguida, imprimir uma mensagem com o valor inserido.

2.4 Inserindo um código JavaScript em um documento HTML

O código JavaScript pode ser inserido em várias posições dentro de um documento HTML, dependendo da necessidade do desenvolvedor e do comportamento desejado. Existem três abordagens principais: inserir o código no head, no body ou em um arquivo externo. Para pequenos *scripts* ou manipulações rápidas, inserir o código no head ou no body pode ser suficiente. No entanto, para projetos maiores ou em produção, utilizar arquivos JavaScript externos é considerado uma boa prática, pois facilita a manutenção e melhora o desempenho. Abaixo, discutimos cada uma dessas abordagens com exemplos práticos.

- **Código JavaScript no HEAD:**

  Colocar o código JavaScript dentro da seção <head> é uma prática comum quando o script precisa ser carregado antes do conteúdo da página. No entanto, deve-se ter cuidado, pois, se o script tentar manipular elementos que ainda não foram carregados (que estão no body), isso pode causar erros. Nesses casos, pode ser útil envolver o código dentro de um evento window.onload para garantir que o DOM esteja totalmente carregado antes que o script seja executado.

```
<!DOCTYPE html>
<html>
<head>
<script>
    function myFunction() {
        document.getElementById("demo").innerHTML =
        "Parágrafo modificado.";
    }
</script>
</head>
<body>
    <h2>JavaScript no Head</h2>
    <p id="demo">Um paragráfo.</p>
    <button type="button" onclick="myFunction()">
    Clique Aqui</button>
</body>
</html>
```

Código JS no Head

Nota: Ao inserir scripts no head, o código será carregado antes do conteúdo do body. Se o script precisar interagir com elementos do body, deve-se garantir que ele seja executado após o carregamento da página.

- **Código JavaScript no BODY:**

Outra abordagem comum é inserir o código JavaScript dentro do corpo (body) do documento HTML. Isso é útil quando o script precisa interagir diretamente com os elementos presentes no body e quando você deseja que o JavaScript seja carregado junto com o conteúdo da página.

```
<!DOCTYPE html>
<html>
<body>
    <h2>JavaScript no  Body</h2>
    <p id="demo">Um parágrafo.</p>
    <button type="button" onclick="myFunction()">
    Clique Aqui.</button>
    <script>
    function myFunction() {
        document.getElementById("demo").innerHTML =
        "Paragrafo Modificado.";
    }
    </script>
</body>
</html>
```

Código JS no Body

Nota: Inserir o script no final do body pode melhorar o desempenho da página, já que o conteúdo HTML é carregado antes do script ser executado, evitando atrasos na renderização da página.

- **Código JavaScript inserido a partir de um arquivo externo:**

  A melhor prática para projetos maiores ou scripts reutilizáveis é manter o código JavaScript em arquivos separados. Isso facilita a manutenção, a reutilização de código e também contribui para um carregamento mais eficiente, já que o navegador pode armazenar em cache o arquivo JavaScript externo.

```
<!DOCTYPE html>
<html
<body>
    <h2>JavaScript Externo</h2>
    <p id="demo">Um Paragrafo.</p>
    <button type="button" onclick="myFunction()">
    Clique Aqui</button>
    <p>(myFunction está armazenado em um
    arquivo externo de nome "myscript.js")</p>
    <script src="myscript.js"></script>
</body>
```

```
</html>
```

Código JS no Arquivo Externo

Arquivo Externo (myscript.js):

```
function myFunction() {
  document.getElementById("demo").innerHTML = "Parágrafo modificado.";
}
```

myscript.js

As vantagens de uso do JavaScript externo são a facilidade de manter e atualizar o código, pois está separado do HTML. O mesmo arquivo JavaScript pode ser utilizado em várias páginas, evitando duplicação de código. Finalmente, os arquivos externos podem ser armazenados em cache pelo navegador, reduzindo o tempo de carregamento em visitas subsequentes.

2.5 Modificando um Documento HTML

Atualizar um documento HTML com JavaScript pode ser feito de várias maneiras, dependendo do que você deseja modificar ou adicionar ao documento. Abaixo estão as principais formas de atualizar um documento HTML utilizando JavaScript:

1. Modificando o Conteúdo de Elementos (innerHTML e textContent)

   - innerHTML: Permite inserir ou substituir o conteúdo HTML de um elemento. Isso inclui a adição de tags HTML no conteúdo, tornando-o mais flexível.

   ```
   document.getElementById("elementoID").innerHTML = "<p>Novo conteú
       do com HTML</p>";
   ```

   - textContent: Substitui o conteúdo de texto de um elemento sem interpretar o HTML. Isso é mais seguro se você quiser evitar a execução de scripts embutidos.

   ```
   document.getElementById("elementoID").textContent = "Novo conteúdo
       de texto";
   ```

2. Alterando Atributos de Elementos (setAttribute)

   Você pode alterar ou adicionar atributos de elementos HTML, como src, href, class, etc., utilizando o método setAttribute().

```
document.getElementById("imagemID").setAttribute("src", "novaImagem.
    png");
```

3. Alterando Estilos CSS (style Property)

   Modifique o estilo de um elemento diretamente através da propriedade style.

```
document.getElementById("elementoID").style.backgroundColor = "blue";
```

4. Manipulando Classes (classList API)

   A API classList permite adicionar, remover ou alternar classes CSS em um elemento.

   - add: Adiciona uma classe ao elemento.

```
document.getElementById("elementoID").classList.add("novaClasse");
```

   - remove: Remove uma classe existente.

```
document.getElementById("elementoID").classList.remove("
    classeExistente");
```

   - toggle: Alterna a presença de uma classe.

```
document.getElementById("elementoID").classList.toggle("
    classeAlternada");
```

5. Inserindo Novos Elementos (createElement e appendChild)

   Crie novos elementos HTML dinamicamente e insira-os no DOM usando createElement() e appendChild().

```
var novoElemento = document.createElement("div");
novoElemento.textContent = "Novo Elemento";
document.body.appendChild(novoElemento);
```

6. Substituindo Elementos (replaceChild)

   Você pode substituir um elemento existente no DOM por um novo elemento.

```
var novoElemento = document.createElement("div");
novoElemento.textContent = "Novo Elemento";
var elementoExistente = document.getElementById("elementoExistente");
document.body.replaceChild(novoElemento, elementoExistente);
```

7. Removendo Elementos (removeChild ou remove)

   Para remover um elemento do DOM, você pode usar removeChild() ou o método remove() (em navegadores modernos).

   - removeChild:

```
var elementoParaRemover = document.getElementById("elementoID");
elementoParaRemover.parentNode.removeChild(elementoParaRemover);
```

   - remove (mais direto):

```
document.getElementById("elementoID").remove();
```

8. Eventos e Interatividade

   Você também pode atualizar o documento em resposta a eventos de usuário, como cliques, toques ou alterações de formulário, usando addEventListener().

```
document.getElementById("botaoID").addEventListener("click", function
    () {
  document.getElementById("elementoID").textContent = "Texto
   atualizado após o clique";
});
```

9. Manipulação de Atributos de Dados (dataset API)

   A API dataset permite acessar e modificar atributos data-* personalizados de um elemento.

```
document.getElementById("elementoID").dataset.valor = "novoValor";
```

10. AJAX e Fetch API

    Utilize AJAX (com XMLHttpRequest) ou a Fetch API para carregar novos dados do servidor e atualizar dinamicamente o conteúdo do documento HTML sem recarregar a página.

    - AJAX:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "dados.html", true);
xhr.onload = function() {
  if (xhr.status === 200) {
    document.getElementById("elementoID").innerHTML = xhr.
   responseText;
  }
```

```
};
xhr.send();
```

- Fetch API:

```
fetch("dados.html")
  .then(response => response.text())
  .then(data => {
    document.getElementById("elementoID").innerHTML = data;
  });
```

11. Templates (template element)

    Você pode utilizar elementos <template> para definir fragmentos de HTML que podem ser clonados e inseridos dinamicamente no documento.

```
var template = document.getElementById("templateID");
var clone = template.content.cloneNode(true);
document.body.appendChild(clone);
```

12. Manipulação de Nós DOM (insertAdjacentHTML, insertBefore, after, before)

    Métodos como insertAdjacentHTML, insertBefore, after e before permitem inserir conteúdo no DOM em posições específicas em relação a outros elementos.

    - insertAdjacentHTML:

```
document.getElementById("elementoID").insertAdjacentHTML("
    beforeend", "<p>Texto inserido</p>");
```

    Essas são algumas das principais formas de atualizar dinamicamente um documento HTML usando JavaScript, permitindo desde simples modificações de texto até a manipulação avançada de elementos e estruturas do DOM.

3

Sintaxe

3.1 Variáveis

No JavaScript há três formas principais de declarar variáveis:

1. **var**: Declara variáveis com escopo de função ou global, dependendo de onde é usada. Pode ser redeclarada e atualizada.

```
var x = 10;
```

2. **let**: Declara variáveis com escopo de bloco. Pode ser atualizada, mas não redeclarada no mesmo escopo.

```
let y = 20;
```

3. **const**: Declara constantes com escopo de bloco. Não pode ser atualizada ou redeclarada.

```
const z = 30;
```

4. Você pode declarar múltiplas variáveis em uma única linha usando vírgulas.

```
let x = 10, y = 20;
```

O nomes em JavaScript são sensíveis à caixa-alta ou baixa: *minhaVariavel* é diferente de *minhavariavel*. A seguir diferentes formas de declarar e atribuir valor a uma variável.

```
var minhaVariavel;
minhaVariavel = 'Beto';
var minhaVariavel = 'Beto';
minhaVariavel = 'Esteves';
```

O escopo de uma variável em JavaScript refere-se ao contexto no qual a variável é definida e onde ela pode ser acessada ou referenciada. Existem três tipos principais de escopo:

1. **Escopo Global**: Variáveis declaradas fora de qualquer função ou bloco são acessíveis em qualquer parte do código.
2. **Escopo de Função**: Variáveis declaradas com 'var' dentro de uma função são acessíveis apenas dentro dessa função.
3. **Escopo de Bloco**: Variáveis declaradas com 'let' e 'const' dentro de um bloco (como os delimitados por '{}') são limitadas ao bloco em que foram definidas.

A seguir algumas observações interessantes sobre variáveis em JavaScript:

1. **Hoisting**: Em JavaScript, as declarações de variáveis com 'var' são "elevadas"para o topo de seu escopo. No entanto, o valor atribuído não é elevado.

```
console.log(x); // undefined
var x = 5;
```

2. **Redeclaração**: Variáveis declaradas com 'var' podem ser redeclaradas sem erro, ao contrário de 'let' e 'const'.
3. **Variáveis globais e locais**: Se uma variável é declarada fora de qualquer função, ela é global. Se for declarada dentro de uma função, será local e só estará acessível dentro daquela função.
4. **Reatribuição**: 'let' permite reatribuição, enquanto 'const' impede a reatribuição de valores, exceto para objetos e *arrays*, onde as propriedades ainda podem ser modificadas.

3.2 Comentários

No JavaScript, você pode realizar comentários de duas formas principais:

- **Comentários de uma linha**: Usando '//', qualquer texto após essa sequência na linha será ignorado pelo JavaScript.

```
// Isto também é um comentário
// mas precisa colocar sempre
// antes de todas as linhas
```

- **Comentários de várias linhas**: Usando '/* */', qualquer texto entre essas sequências será ignorado, permitindo comentários que se estendem por várias linhas.

```
/*
Tudo o que estiver aqui é comentário
*/
```

3.3 Operadores

A seguir apresentamos alguns dos operadores mais utilizados na linguagem JavaScript. Em particular, notem a diferença entre == e ===, onde o segundo operador exige que as variáveis sendo comparadas sejam do mesmo tipo para que a igualdade seja verdadeira e no primeiro caso não.

Operadores Relacionais	**Significado**
== ou ===	Igualdade (**do mesmo tipo**)
!= ou !==	Desigualdade (**do mesmo tipo**)
<, <=	Menor / Menor ou Igual
>, >=	Maior / Maior ou Igual

Operadores Lógicos	**Significado**
&&	E (AND)
||	OU (OR)
!	Negação (NOT)

Operadores Bit a Bit	**Significado**
&	E (AND)
|	OU (OR)
^	Ou Exclusivo (XOR)
~	Negação (NOT)

Para garantir a precedência dos operadores em JavaScript, siga estas práticas:

- **Uso de Parênteses**: A maneira mais segura de garantir a ordem de avaliação é utilizando parênteses. Isso força o JavaScript a realizar as operações na ordem que você deseja.

Tabela 3.1: Operadores Aritméticos

Operadores Aritméticos	**Significado**	**Exemplo**
+, -	Soma / Subtração	a+b
*, /, **	Multiplicação / Divisão / Potência	x * 2; Soma / 3; x = x ** y
%	Resto da divisão	Soma % 3
++, - -	Incremento / Decremento	a++; ++a; b- -; - -b;
=	Atribuição Simples	Media = (a+b+c) / 3;
+=, -=, *=, /=, %=, « =, »=	Atribuição Composta	A -= 1 ; equivalente a A = A - 1; R %= 2; equivalente a R = R % 2;
«, »	Deslocamento à esquerda/direita	x = x « y x = x » y

```
let result = (2 + 3) * 4; // Garante que a soma seja feita antes da
 multiplicação.
```

- **Entendimento das Regras de Precedência**: Conheça a tabela de precedência dos operadores, cosultando o manual da linguagem. Operadores como multiplicação (‘*‘) e divisão (‘/‘) têm precedência mais alta que adição (‘+‘) e subtração (‘-‘).

Mesmo entendendo as regras, o uso de parênteses torna o código mais claro e evita erros.

3.4 Tipos de Dados

O JavaScript é uma linguagem de tipagem dinâmica, o que significa que você não precisa declarar explicitamente o tipo de uma variável. O tipo é inferido automaticamente pelo valor atribuído. Nesta seção relacionamos alguns tipos de dados de uso mais comum em JavaScript, tais como constant, string, number, boolean, array e objeto. Notem que pode haver mais de uma maneira de declarar cada um desses tipos. Veja a Tabela 3.2.

3.5 Arrays

Arrays são um tipo de dados muito importantes em qualquer linguagem, assim como em Javscript. ’Arrays’ em JavaScript são estruturas de dados que armazenam uma coleção de valores em uma única variável. Esses valores podem ser de qualquer tipo, como números, strings, objetos ou até mesmo outros ’arrays’ (’arrays’ aninhados). Os ’arrays’ são indexados por números, onde o primeiro elemento tem índice 0, o segundo índice 1, e assim por diante. Uma maneira de criar e declarar *arrays* em JavaScript pode ser a seguinte:

Tabela 3.2: Tipos de Dados

Tipo	Explanação	Exemplo
constant	Um valor apenas de leitura, de qualquer tipo definido abaixo.	const myConstant = 20;
string	Uma sequência de texto entre plics ou aspas.	var myVariable = 'Beto' let myVariable = "Beto"
number	Os números em JavaScript são todos reais.	var myVariable = 10;
boolean	Um valor verdadeiro/falso, não precisa de aspas.	var myVariable = true;
array	Uma estrutura que permite armazenar diversos valores em uma única referência	var myVariable = [1,'Beto','José',10]; Cada membro do array como myVariable[0], myVariable[1]...
object	Basicamente tudo em JavaScript é um objeto e pode ser armazenado em uma variável.	myVariable = document.querySelector('h1');

```
var a = new Array();      // arquivo array.js
a[0] = 'cao';
a[1] = 'gato';
a[2] = 'frango';
a.length; // 3
```

Uma notação mais conveniente é usar um *array* de literais:

```
var a = ['cao', 'gato', 'frango'];
a.length; // 3
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas[0]); // Saída: maçã
```

Se você acessar um índice de *array* que não existe, o valor *undefined* será retornado:

```
typeof a[90]; // undefined
```

Você pode iterar sobre os elementos de um *array* usando um laço do tipo *for* e o método *a.length*, que retorna o tamanho de um *array*:

```
var a=[0,1,2,3,4,5,6,7,8,9,10];
for (var i = 0; i < a.length; i++) {
    console.log(a[i]);
```

```
    }
```

Programa for.js

Um outro modo de iterar sobre os elementos de um array foi adicionado mais recentemente à definição da linguagem:

```
var a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
a.forEach(function(elemento) {
  console.log(elemento);
});
```

Programa foreach.js

O método **forEach()** é um método de array que executa uma função fornecida uma vez para cada elemento do array. A função anônima **function(elemento)** recebe um parâmetro chamado *elemento*, que representa o valor atual do elemento do array durante a iteração. Dentro da função **console.log(elemento)** o valor do elemento atual é impresso no console.

3.6 Métodos Pré-definidos para Arrays

Existem vários métodos úteis pré-definidos para arrays, como 'push()', 'pop()', 'shift()', 'unshift()', 'map()', 'filter()', entre outros, para manipulação dos dados. Vejamos a seguir alguns em detalhes:

1. **push()**: Adiciona um ou mais elementos ao final de um array e retorna o novo comprimento do array.

```
let frutas = ["maçã", "banana"];
console.log(frutas); // ["maçã", "banana"]
frutas.push("laranja");
console.log(frutas); // ["maçã", "banana", "laranja"]
```

Programa push.js

2. **pop()**: Remove o último elemento de um array e retorna esse elemento.

```
let frutas = ["maçã", "banana", "laranja"];
let ultimaFruta = frutas.pop();
console.log(ultimaFruta); // "laranja"
console.log(frutas); // ["maçã", "banana"]
```

Programa pop.js

3. **reverse()**: Inverte o array.

```
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas);
let inverso = frutas.reverse();
console.log(inverso); // ['laranja', 'banana', 'maçã']
```

Programa reverse.js

4. **shift()**: Remove o primeiro elemento de um array e retorna esse elemento. Isso altera o comprimento do array.

```
let frutas = ["maçã", "banana", "laranja"];
let primeiraFruta = frutas.shift();
console.log(primeiraFruta); // "maçã"
console.log(frutas); // ["banana", "laranja"]
```

Programa shift.js

5. **unshift()**: Adiciona um ou mais elementos ao início do array e retorna o novo comprimento.

```
let frutas = ["banana", "laranja"];
console.log(frutas);
frutas.unshift("maçã");
console.log(frutas); // ["maçã", "banana", "laranja"]
```

Programa unshift.js

6. **concat()**: Junta dois ou mais arrays e retorna um novo array.

```
let frutas = ["maçã", "banana"];
console.log(frutas);
let verduras = ["alface", "cenoura"];
console.log(verduras);
let alimentos = frutas.concat(verduras);
console.log(alimentos); // ["maçã", "banana", "alface", "cenoura"]
```

Programa concat.js

7. **slice()**: Retorna uma cópia de uma parte do array selecionada, sem modificar o array original.

```
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas);
let citrus = frutas.slice(1, 3);
console.log("slice(1,3)");
console.log(citrus); // ["banana", "laranja"]
```

Programa slice.js

O método slice() é chamado no array frutas. O parâmetro '1' indica o índice inicial do elemento a ser incluído no novo array (índice 1 é o segundo elemento). O parâmetro '3' indica o índice final (não inclusivo) do elemento a ser incluído no novo array (índice 3 é o quarto elemento). O slice() extrai os elementos do índice 1 até o índice 2 (excluindo o 3), criando um novo array citrus com as frutas "banana" e "laranja".

8. **splice()**: Adiciona ou remove elementos de um array, modificando o array original. Se elementos forem removidos, um novo array com esses elementos é retornado. A sua sintaxe é:

```
array.splice(inicio, quantidade, item1, item2, ...)
```

Onde:

- inicio: Índice onde começar a modificar.
- quantidade: Número de elementos a serem removidos.
- item1, item2, ...: Elementos a serem adicionados.

```
// Adicionar elementos
  let frutas = ["maçã", "laranja"];
  console.log(frutas);
  frutas.splice(1, 0, "banana"); // Adiciona "banana" no índice 1
  console.log("splice(1,0,'banana')");
  console.log(frutas); // ["maçã", "banana", "laranja"]
// Remover elementos
  frutas.splice(2, 1); // Remove 1 elemento a partir do índice 2
  console.log("splice(1,0)");
  console.log(frutas); // ["maçã", "banana"]
```

Programa splice.js

9. **forEach()**: Como já visto, executa uma função para cada elemento do array.

```
let frutas = ["maçã", "banana", "laranja"];
frutas.forEach(function(fruta) {
  console.log(fruta);
});
// Saída:
// maçã
// banana
// laranja
```

Programa foreach2.js

10. **map()**: Cria um novo array com o resultado da chamada de uma função em cada elemento do array.

```
let numeros = [1, 2, 3];
let dobrados = numeros.map(function(num) {
  return num * 2;
});
console.log(dobrados); // [2, 4, 6]
```

Programa map.js

11. **filter()**: Cria um novo array com todos os elementos que passam em um teste implementado por uma função.

```
let numeros = [1, 2, 3, 4];
console.log(numeros);
let pares = numeros.filter(function(num) {
  return num % 2 === 0;
});
console.log("Filtra os pares e gera um novo vetor");
console.log(pares); // [2, 4]
```

Programa filter.js

O 'filter()' percorre cada elemento do array 'numeros', aplica a função de verificação e cria um novo array 'pares' contendo apenas os elementos que retornaram 'true'. O código acima filtra os números pares do array 'numeros' e armazena os resultados em um novo array 'pares'. O método 'filter()' é uma ferramenta poderosa para criar novos arrays com base em condições específicas.

12. **reduce()**: Aplica uma função a um acumulador e cada elemento do array (da esquerda para a direita), reduzindo-o a um único valor.

```
let numeros = [1, 2, 3, 4, 5];
console.log(numeros);
let soma = numeros.reduce(function(acumulador, valorAtual) {
  return acumulador + valorAtual;
}, 0);
console.log("Retorna a soma de todos os elementos do vetor");
console.log(soma); // 10
```

Program reduce.js

O método **reduce()** é aplicado ao array numeros. Esse método reduz o array a um único valor, aplicando uma função a cada elemento. A função anônima **function(acumulador, valorAtual)** é passada como argumento para o **reduce()**. Essa função recebe dois parâmetros: o primeiro é *acumulador*, que é um valor acumulado que é iniciado com o segundo argumento do **reduce()** (no caso, 0); o segundo é *valorAtual*, que é o valor atual do elemento sendo processado. A cada iteração, a função soma o *valorAtual* ao

acumulador e retorna o resultado. Esse resultado se torna o novo valor do acumulador na próxima iteração. O valor inicial do acumulador (0) é essencial para iniciar a soma.

13. **toString()**: Retorna uma cadeia com a toString() de cada elemento separadas por vírgula.

```
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas.toString()); // "maçã,banana,laranja"
```

Program tostring.js

14. **join(sep)**: Converte o array em uma cadeia – com os valores delimitados com o parâmetro *sep*.

```
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas.join(" - ")); // "maçã - banana - laranja"
```

Program join.js

15. **sort([cmpfn])**: Ordena o array com o uso de uma uma função de comparação (opcional) passada como parâmetro.

```
// Sem função de comparação
let numeros = [3, 1, 4, 2];
console.log(numeros.sort()); // [1, 2, 3, 4] (ordem alfabética)
// Com função de comparação
console.log(numeros.sort((a, b) => b - a)); // [4, 3, 2, 1] (ordem
    decrescente)
```

Programa sort.js

Esses métodos permitem uma ampla variedade de manipulações e operações em arrays, tornando o JavaScript muito flexível para trabalhar com coleções de dados.

3.7 Conversão de Tipos

3.7.1 Conversão Implícita de Tipos

Em expressões envolvendo valores numéricos e *strings* com o operador +, os valores numéricos serão convertidos implicitamente para strings. Por exemplo:

```
X = "A resposta é " + 35; // retorna "A resposta é 35"
Y = 35 + " é a resposta" // retorna "35 é a resposta"
```

Em expressões envolvendo outros operadores a conversão será feita de acordo com o tipo de dado esperado pelo operador. Exemplo:

```
X = "37" - 7 // retorna 30
Y = "37" + 7 // retorna 377
```

Quando dois valores são comparados, números, *strings* e valores lógicos são comparados por valor:

```
3 == "3" // Resultado true
1 && true // Resultado true
```

Usando a conversão implícita podemos usar de um artifício simples para forçar a conversão de número para string e de string para número.

- **Número para string:**

```
i = 25 ;
s = "" + i ; // s recebe "25"
```

- **String para número:**

```
s = "256" ;
j = s - 0 ; // j recebe 256
```

3.7.2 Conversão de Explícita de Tipos

parseInt (str) ou parseInt (str,base): Converte uma string num número inteiro. Str é a cadeia a ser convertida e base é a base em que os números estão codificados. Se a base não for especificada, assume que a string é uma seqüência de algarismos na base decimal (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Caso encontre algum caractere diferente dos esperados, encerra a conversão:

```
num = "3A";
x = parseInt(num);     // x recebe 3
y = parseInt(num,16);     // y recebe 58
```

parseFloat (str): É a função equivalente a parseInt, para converter strings em números reais. Da mesma forma, se encontrar algum caractere não esperado, encerra a conversão.

```
z = parseFloat("3.15"); // z recebe 3.15
```

3.8 Minificação em JavaScript

A minificação, também conhecida como minimização, é o processo de remover todos os caracteres desnecessários do código fonte do JavaScript sem alterar sua funcionalidade. Isto inclui a remoção de espaços em branco, comentários e ponto-e-vírgula, juntamente com o uso de nomes e funções de variáveis mais curtas. A minificação do código do JavaScript resulta em um arquivo de tamanho compacto.

A minificação, também conhecida como minimização, é o processo de otimização que remove todos os caracteres desnecessários de um código-fonte, sem alterar sua funcionalidade ou comportamento. Esse processo é comumente aplicado em JavaScript, CSS, e HTML e envolve a eliminação de espaços em branco, quebras de linha, comentários, e até caracteres como ponto-e-vírgula quando não estritamente necessários. Além disso, a minificação inclui a renomeação de variáveis e funções para nomes mais curtos, o que reduz ainda mais o tamanho do código.

O objetivo da minificação é reduzir o tamanho dos arquivos de código para melhorar o desempenho de carregamento da página web. Um arquivo menor implica menos dados para serem transferidos entre o servidor e o navegador, resultando em tempos de carregamento mais rápidos. Isso é crucial tanto para melhorar a experiência do usuário (especialmente em dispositivos móveis com conexões mais lentas) quanto para melhoria da Otimização para Mecanismos de Busca (SEO), já que os mecanismos de busca valorizam sites com carregamento mais rápido. Aqui está um exemplo de um código antes e depois da minificação:

```
    function greet(name) {
    var greeting = "Hello, " + name + "!";
    console.log(greeting);
}
greet("Alice");
```

Após a minificação:

```
function greet(a){console.log("Hello, "+a+"!")}greet("Alice");
```

Benefícios adicionais da minificação

- Redução de largura de banda: Com arquivos menores, menos dados precisam ser transmitidos, o que diminui o consumo de largura de banda tanto para o servidor quanto para o usuário.
- Melhoria da performance do site: Menos bytes para carregar resultam em uma renderização mais rápida da página, melhorando a experiência do usuário final, principalmente em redes lentas.

- Obfuscação leve: Embora a minificação não seja o mesmo que a ofuscação, o código resultante da minificação é mais difícil de ler para seres humanos, o que pode oferecer um nível básico de proteção contra cópia ou modificação não autorizada. Melhora no SEO: Páginas mais rápidas têm maior probabilidade de classificação mais elevada nos resultados de pesquisa, já que mecanismos como o Google consideram a velocidade de carregamento um fator relevante. Menos solicitações HTTP: Em alguns casos, as ferramentas de minificação combinam vários arquivos em um único arquivo minificado, o que reduz o número de requisições HTTP, resultando em menos sobrecarga no carregamento.

Ferramentas populares para minificação

- UglifyJS: Ferramenta amplamente usada para minificação de JavaScript.
- Terser: Outra ferramenta para minificação de JavaScript, baseada no UglifyJS, com suporte a ES6+.
- CSSNano: Utilizada para minificar arquivos CSS.
- HTMLMinifier: Reduz o tamanho de arquivos HTML, removendo espaços e comentários desnecessários.

4

Sentenças Condicionais e Laços

4.1 if-then-else

O uso de sentenças condicionais, assim como em outras linguagens, também é possível em JavaScript. A seguir um exemplo que apresenta o resultado na console.

```
// Declaração e valor inicial da variável 'saudacao'
var saudacao = 'Bom dia!';
// Condicional para verificar o valor da variável 'saudacao'
if (time < 10) {
    saudacao = "Bom dia!";
} else if (time < 20) {
    saudacao = "Boa tarde!";
} else {
    saudacao = "Boa noite!";
}
console.log(saudacao);
```

saudacao.js

...

```
// Declaração e valor inicial da variável 'nome'
var nome = 'gatinho';
// Condicional para verificar o valor da variável 'nome'
if (nome == 'toto') {
    // Se 'nome' for igual a 'toto', adiciona ' auau' ao final
    nome += ' auau';
} else if (nome == 'gatinho') {
    // Se 'nome' for igual a 'gatinho', adiciona ' miau' ao final
    nome += ' miau';
} else {
    // Se não for nenhuma das opções anteriores, adiciona '!' ao final
```

```
    nome += '!';
}
// Exibe o resultado no console
console.log(nome);
```

pet.js

4.2 Switch

O comando *switch* pode ser usado para fazer uma escolha entre diversos itens, sem que seja necessário o uso do *if-then-else*. A seguir um exemplo com uso do switch.

```
var sinal = 'verde';
    switch (sinal) {
        case 'verde':
            console.log("Sinal verde: Pode seguir!");
            break;
        case 'vermelho':
            console.log("Sinal vermelho: Pare o veículo.");
            break;
        case 'amarelo':
            console.log("Sinal amarelo: Reduza a velocidade.");
            break;
        default:
            // Caso o sinal seja desconhecido, também pare como medida de
   segurança
            console.log("Sinal desconhecido: Mantenha-se parado.");
            Pare();
    }
```

4.3 Laços com while e for

Para iterar diversas vezes podem ser utilizados o *for* e o *while*. Inicialmente, vejamos um exemplo com *while*:

```
while (true) {
  // um laço infinito!
}

var input;
do {
  input = get_input();
```

```
} while (inputIsNotValid(input));
```

O loop ‘for‘ é uma construção utilizada para executar um bloco de código um número determinado de vezes. Ele é especialmente útil quando precisamos iterar sobre elementos de um array, objetos ou realizar cálculos repetitivos.

```
// Estrutura básica do ‘for‘
for (inicialização; condição; incremento) {
    // Código a ser executado em cada iteração
}

for (var i = 0; i < 5; i++) {
  // vai executar 5 vezes
}

for (let value of array) {
  // faça alguma coisa com o valor
}
for (let property in object) {
  // faça alguma coisa com a propriedade do objeto

}
```

- Inicialização: Uma expressão que é executada apenas uma vez, antes do início do loop. Geralmente, é usada para declarar e inicializar uma variável de controle.
- Condição: Uma expressão que é avaliada antes de cada iteração. Se a condição for verdadeira, o corpo do loop é executado. Caso contrário, o loop termina.
- Incremento: Uma expressão que é executada após cada iteração. Normalmente, é usada para atualizar a variável de controle.
- Iterando sobre um array:

```
const frutas = ["maçã", "banana", "laranja"];

for (let i = 0; i < frutas.length; i++) {
  console.log(frutas[i]);
}
```

Neste exemplo, o laço itera sobre cada elemento do array ‘frutas‘, imprimindo o nome de cada fruta no console.

- Criando uma tabela de multiplicação:

```
const numero = 5;

for (let i = 1; i <= 10; i++) {
  const resultado = numero * i;
  console.log(numero + " x " + i + " = " + resultado);
}
```

Este código cria a tabela de multiplicação do número 5.

- Somando os números de 1 a 100:

```
let soma = 0;

for (let i = 1; i <= 100; i++) {
  soma += i;
}

console.log("A soma dos números de 1 a 100 é:", soma);
```

Este exemplo calcula a soma dos números de 1 a 100.

- Iterando sobre as propriedades de um objeto:

```
const pessoa = {
  nome: "João",
  idade: 30,
  cidade: "São Paulo"
};

for (const propriedade in pessoa) {
  console.log(propriedade + ": " + pessoa[propriedade]);
}
```

Este código itera sobre as propriedades do objeto ‘pessoa‘ e imprime o nome da propriedade e seu valor.

Observações:

- A variável de controle ‘i‘ é comumente utilizada, mas você pode usar qualquer nome válido.
- O incremento pode ser positivo ou negativo, permitindo percorrer o array em ordem inversa.
- É importante garantir que a condição de parada seja alcançada para evitar loops infinitos.

Além do ‘for‘, o JavaScript oferece outras formas de iterar sobre arrays e objetos, como ‘forEach‘, ‘map‘, ‘filter‘, ‘reduce‘, etc. Os respectivos exemplos foram apresentados anteriormente na Seção 3.6. A seguir um exemplo mais completo.

```
const produtos = [
  { nome: 'Camiseta', preço: 20 },
  { nome: 'Calça', preço: 50 },
  { nome: 'Tênis', preço: 100 }
];

// Filtrar produtos com preço acima de 30 e calcular o preço total
const produtosCaros = produtos.filter(produto => produto.preço > 30);
const precoTotal = produtosCaros.reduce((total, produto) => total + produto
    .preço, 0);
console.log(produtos);
console.log("Filtra os mais caros e imprime o total");
console.log(produtosCaros);
console.log("Preço total:", precoTotal);
```

Programa produtos.js

Cada um desses métodos tem suas particularidades e é mais adequado para determinadas situações.

Quando usar cada método

- forEach: Ideal para iterar e realizar ações em cada elemento, sem a necessidade de criar um novo array.
- map: Utilizado quando se deseja criar um novo array transformando cada elemento do array original.
- filter: Empregado para criar um novo array com os elementos que atendem a uma determinada condição.
- reduce: Adequado para reduzir um array a um único valor, como a soma, o produto, o maior valor, etc.

Observações Os exemplos acima utilizam **arrow functions**, uma sintaxe mais concisa para definir funções. Veja mais detalhes na Seção 7.7.

Existem outros métodos de array como ‘some‘, ‘every‘, ‘find‘, ‘findIndex‘, entre outros, cada um com sua funcionalidade específica. A escolha do método ideal depende da operação a ser realizada e do tamanho do array. Em alguns casos, o loop ‘for‘ pode ser mais eficiente, especialmente em arrays muito grandes.

5

Eventos

Eventos em HTML sao “coisas” que acontecem com os elementos HTML. Quando o JavaScript é utilizado em páginas HTML, ele pode “reagir” a esses eventos. Um evento HTML pode ser alguma coisa que o navegador faz, pode ser algo que o usuário faz.

Veja a seguir alguns exemplos de eventos em HTML:

- Uma página HTML que acabou de carregar;
- Um campo de entrada HTML que mudou de valor;
- Um botão HTML que foi pressionado.

Frequentemente, quando um evento acontece, você pode querer fazer alguma coisa. O JavaScript permite que você execute código quando eventos são detectados. Veja a seguir:

```
document.querySelector('html').onclick = function() {
    alert('Ai! Não me cutuque!');
}
```

Há muitas maneiras de conectar um evento com um elemento. Aqui o elemento HTML foi selecionado, atribuindo ao manipulador *onclick* a uma função anônima, que contém o código que queremos executar. Note que

```
document.querySelector('html').onclick = function() {};
```

é equivalente a

```
var myHTML = document.querySelector('html');
myHTML.onclick = function() {};
```

O HTML permite que atributos de manipuladores de evento, com código JavaScript, sejam adicionados aos elementos HTML.

- Com aspas simples: **<element event='codigo JavaScript'>**
- Com aspas duplas: **<element event="codigo JavaScript" >**

No exemplo seguinte, um atributo *onclick* (com código) é adicionado ao elemento *button* em HTML:

```
<button onclick="document.getElementById('demo').innerHTML =
Date()">Que horas são?</button>
```

No exemplo acima, o JavaScript muda o conteúdo do elemento com **id="demo"**. No próximo exemplo, o código muda o conteúdo de seu próprio elemento (usando this.innerHTML):

```
<button onclick="this.innerHTML = Date()">Que horas são?</button>
```

Outros usos do 'this' serão vistos mais adiante, na Seção 6.3.

O código JavaScript frequentemente tem várias linhas de código. É cada vez mais comum vermos atributos de eventos chamando funções.

```
<button onclick="displayDate()">
Que horas são?</button>
```

A seguir uma lista de alguns eventos HTML usuais:

Evento	**Descrição**
onchange	Uma mudança em um elemento HTML
onclick	O usuário clicou em um elemento HTML
onmouseover	O usuário movimentou o mouse sobre o elemento HTML
onmouseout	O usuário moveu o mouse para fora do elemento HTML
onkeydown	O usuário pressionou alguma tecla no teclado
onload	O navegador terminou de carregar a página

O manipuladores de evento podem ser utilizados para manipular e verificar a entrada de dados do usuário, ações do usuário e ações do navegador:

- Coisas que devem ser feitas toda vez que uma página é carregada;
- Coisas que devem ser feitas quando uma página é fechada;
- Ações que devem ser realizadas quando um usuário clica um botão;

- Conteúdo que deve ser verificado quando o usuário faz uma entrada de dados.
- E mais...

Outros exemplos possíveis são:

- **'onclick'**: Executado quando um elemento é clicado.

```
    document.getElementById("botao").onclick = function() {
      alert("Botão clicado!");
    };
```

- **'onmouseover'**: Executado quando o mouse passa sobre um elemento.

```
const imagem = document.getElementById("imagem");
imagem.onmouseover = function() {
  imagem.style.border = "2px solid red";
};
```

- **'onkeyup'**: Executado quando uma tecla é liberada.

```
    const inputTexto = document.getElementById("inputTexto");

inputTexto.onkeyup = function() {
  console.log(inputTexto.value);
};
```

- **'onload'**: Executado quando a página ou um recurso específico é totalmente carregado.

```
    window.onload = function() {
      alert("Página carregada!");
    };
```

- **'onsubmit'**: Executado ao enviar um formulário.

```
    document.getElementById("formulario").onsubmit = function() {
      alert("Formulário enviado!");
    };
```

Esses eventos tornam as páginas mais interativas e dinâmicas, permitindo que ações específicas ocorram em resposta ao comportamento do usuário

6

Objetos

6.1 Criando Objetos

Há duas formas básicas para se criar um objeto vazio:

```
var obj = new Object();
```

```
var obj = {};
```

Essas formas são semanticamente equivalentes e a segunda é chamada sintaxe literal do objeto e é mais conveniente. Esta sintaxe também é o núcleo do formato JSON e deve ser preferida todas as vezes. A sintaxe literal do objeto pode ser usada para iniciar um objeto em sua totalidade.

```
var obj = {
  nome: 'Cenoura',
   for: 'Mario',
  details: {
    color: 'laranja',
    size: 12
  }
}
```

Atributos dos objetos podem ser encadeados juntos:

```
obj.details.color; // orange
obj['details']['size']; // 12
```

Os exemplos a seguir criam um protótipo de um objeto, **Pessoa**, e uma instância deste protótipo, **Voce**.

```
function Pessoa(nome, idade) {
  this.nome = nome;
  this.idade = idade;
}
// Define um objeto
var Voce = new Pessoa('Voce', 24);
// Estamos criando uma nova pessoa de nome  "Voce"
// com a idade igual a 24
```

O 'this' se refere ao próprio objeto criado. Maiores detalhes são apresentados na Seção 6.3.

6.2 Propriedades dos Objetos

Uma vez criado, as propriedades de um objeto podem ser criadas de dois modos:

```
obj.nome = 'Simonal';
var nome = obj.nome;
E ...
obj['nome'] = 'Simonal';
var nome = obj['nome'];
```

Essas formas também são semanticamente equivalentes. O segundo método tem a vantagem que o nome da propriedade é fornecido como uma *string*, o que significa que ela pode ser calculada em tempo de execução.

Contudo, o uso deste método previne que algumas máquinas JavaScript utilizem certas otimizações de minimização.

Esta forma também pode ser usada para atribuir propriedades com nomes que são palavras reservadas no JavaScript:

```
obj.for = 'Simonal'; // Erro de sintaxe, porque 'for' é uma
                     // palavra reservada
obj['for'] = 'Simonal'; // Funciona sem problemas
```

6.3 This

Em JavaScript, o 'this' é uma palavra-chave especial que **se refere ao objeto que está sendo executado** no momento. É como um ponteiro que aponta para o objeto atual. No entanto, o valor de 'this' pode variar dependendo do contexto em que a função é chamada. O 'this' é fundamental para:

- Acessar propriedades e métodos de um objeto: Dentro de um método de um objeto, ‘this‘ se refere ao próprio objeto. Isso permite acessar e modificar suas propriedades e chamar outros métodos.
- Criar objetos com construtores: Ao usar a palavra-chave ‘new‘ para criar um objeto a partir de um construtor, ‘this‘ se refere ao novo objeto sendo criado.
- Ligar funções a objetos: Através de métodos como ‘bind‘, ‘call‘ e ‘apply‘, podemos definir o valor de ‘this‘ para uma função antes de executá-la.

O valor de ’this’ muda nas seguintes condições:

- Chamadas de função: Quando uma função é chamada como uma função normal (não como um método de um objeto), ‘this‘ geralmente se refere ao objeto global (‘window‘ no navegador).
- Métodos de objetos: Quando uma função é chamada como um método de um objeto, ‘this‘ se refere ao objeto que chamou o método.
- Construtores: Dentro de um construtor, ‘this‘ se refere ao novo objeto que está sendo criado.
- Métodos como ‘bind‘, ‘call‘ e ‘apply‘:** Esses métodos permitem definir explicitamente o valor de ‘this‘ para uma função.

```
const pessoa = {
  nome: 'João',
  idade: 30,
  saudar: function() {
    console.log(‘Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.‘)
    ;
  }
};

pessoa.saudar(); // Saída: Olá, meu nome é João e tenho 30 anos.

// Usando call para mudar o valor de this
const outraPessoa = { nome: 'Maria', idade: 25 };
pessoa.saudar.call(outraPessoa); // Saída: Olá, meu nome é Maria e tenho 25
    anos.
```

Pontos importantes para lembrar:

- O valor de ‘this‘ é determinado no momento da chamada da função, não quando a função é definida.

- As arrow functions não possuem seu próprio ‘this‘, elas herdam o ‘this‘ do escopo em que são definidas.
- ‘bind‘, ‘call‘ e ‘apply‘ são métodos úteis para controlar o valor de ‘this‘ em funções.

7

Funções

Uma função JavaScript pode ter ’0’ ou mais parâmetros nomeados. O corpo da função pode conter tantas declarações quanto você quiser, e variáveis que são locais para aquela função podem ser declaradas no corpo da função.

A declaração *return* pode ser usada para retornar um valor a qualquer momento, terminando a função.

```
function soma(num1,num2) {
  var resultado = num1 + num2;
  return resultado;
}

soma(4,7);
soma(20,20);
soma(0.5,3);
```

Se nenhuma declaração de retorno for usada (ou um *return* vazio sem nenhum valor), o JavaScript retornará o valor *undefined*.

Os parâmetros enumerados revelam-se mais como orientações do que qualquer outra coisa. Você pode chamar uma função sem passar os parâmetros que espera, caso em que eles serão definidos como *undefined*.

```
soma(); // NaN
// Não se pode realizar a soma em um undefined
```

Você também pode passar mais argumentos do que a função está esperando:

```
soma(2, 3, 4); // 5
// somou os dois primeiros argumentos; 4 foi ignorado
```

7.1 Argumentos de Funções

Isso pode parecer um pouco simples, mas as funções têm acesso a uma variável adicional dentro de seu corpo chamada *argument*, que é um objeto do tipo *array* contendo todos os valores passados para a função. Vamos re-escrever a função soma para receber quantos valores quisermos:

```
function soma() {
    var resultado = 0;
for (var i = 0, j = arguments.length; i < j; i++) {
      resultado += arguments[i];
      }
return resultado;
}
soma(2, 3, 4, 5); // 14
```

Isso realmente não é mais útil do que apenas escrever 2 + 3 + 4 + 5. Vamos criar então uma função que calcule a média:

```
function media() {
var soma = 0;
      for (var i = 0, j = arguments.length; i < j; i++) {
            soma += arguments[i];
        }
  return soma / arguments.length;
  }
  media(2, 3, 4, 5); // 3.5
```

7.2 Operador Spread

O operador *spread* é usado em declarações de função com o formato: ... [variável] e incluirá dentro dessa variável toda a lista de argumentos não capturados com os quais a função foi chamada. Vamos também substituir o laço *for* por um laço *for ... of* para retornar os valores dentro de nossa variável.

```
function media(...argumentos) {
    var soma = 0;
    for (let valor of argumentos) {
        soma += valor;
    }
    return soma / argumentos.length;
}
```

```
    media(2, 3, 4, 5); // 3.5
```

7.3 Funções Anônimas

O JavaScript permite que você crie funções anônimas.

```
var avg = function() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
};
```

7.4 Recursividade em Funções

O JavaScript permite que você chame funções recursivamente. Isso é particularmente útil para lidar com estruturas de árvore, como as encontradas no DOM do navegador.

```
function countChars(elm) {
  if (elm.nodeType == 3) { // TEXT_NODE
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.childNodes[i]; i++) {
    count += countChars(child);
  }
  return count;
}
```

Observe que as funções JavaScript são objetos - como tudo o mais em JavaScript - e você pode adicionar ou alterar suas propriedades, como vimos anteriormente.

7.5 Funções anexadas a Objetos

Vamos criar duas funções para exibir o nome de uma pessoa. Basicamente há duas maneiras pelas quais o nome pode ser exibido: como "primeiro último"ou como "último, primeiro". Usando as funções e objetos que discutimos anteriormente, poderíamos exibir os dados como este:

```
function criaPessoa(primeiro, ultimo) {
  return {
    primeiro: primeiro,
    ultimmo: ultimo,
    Nomecompleto: function() {
      return this.primeiro + ' ' + this.ultimo;
    },
    NomecompletoReverso: function() {
      return this.ultimo + ', ' + this.primeiro;
    }
  };
}
s = criaPessoa('Wilson' ,'Simonal');
s.Nomecompleto(); // "Wilson Simonal"
s.NomecompletoReverso(); // "Simonal, Wilson"
```

Podemos aproveitar a palavra-chave *this* para melhorar nossa função **criaPessoa**:

```
function Pessoa(primeiro, ultimo) {
  this.primeiro = primeiro;
  this.ultimo = ultimo;
  this.Nomecomplero = function() {
    return this.primeiro + ' ' + this.ultimo;
  };
  this.NomecompletoReverso = function() {
    return this.ultimo + ', ' + this.primeiro;
  };
}
var s = new Pessoa('Wilson','Simonal');
```

Introduzimos outra palavra-chave: *new*. Ela cria um novo objeto vazio e, em seguida, chama a função especificada, com este conjunto para esse novo objeto. Por exemplo:

```
function Pessoa(primeiro, ultimo, idade, olhos) {
    this.primeiroNome = primeiro;
    this.ultimoNome = ultimo;
    this.idade = idade;
    this.corOlhos = olhos;
    this.nome = function() {return this.primeiroNome + " " + this.
   ultimoNome;};
}
var s = new Pessoa('Wilson','Simonal', 18, 'blue');
console.log(s.primeiroNome);
console.log(s.ultimoNome);
console.log(s.nome());
```

Esta função pode ainda ser melhorada criando-se as funções do método apenas uma vez, e atribuindo referências a elas dentro do construtor.

```
function pessoaNomeCompleto() {
  return this.primeiro + ' ' + this.ultimo;
}
function pessoaNomeCompletoReverso() {
  return this.ultimo + ', ' + this.primeiro;
}
function Pessoa(primeiro, ultimo) {
  this.primeiro = primeiro;
  this.ultimo = ultimo;
  this.Nomecompleto = pessoaNomeCompleto;
  this.NomecompletoReverso = pessoaNomeCompletoReverso;
}
```

7.6 Protótipos de Funções

Podemos também criar um protótipo de função, que depois pode ser utilizado para criar diversos tipos de objetos.

```
function Pessoa(primeiro, ultimo) {
  this.primeiro = ultimo;
  this.ultimo = ultimo;
}
Pessoa.prototype.Nomecompleto = function() {
  return this.primeiro + ' ' + this.ultimo;
};
Pessoa.prototype.NomecompletoReverso = function() {
  return this.ultimo + ', ' + this.primeiro;
};
var s = new Pessoa('Wilson','Simonal');
console.log(s.primeiro);
console.log(s.ultimo);
console.log(s.Nomecompleto());
console.log(s.NomecompletoReverso());
```

7.7 Arrow Functions

Arrow functions são uma sintaxe mais moderna e concisa introduzida no ECMAScript 6 (ES6) para criar funções em JavaScript. Elas são chamadas de "arrow functions" devido à seta (=>) que é utilizada em sua sintaxe. Essa sintaxe tem as seguintes vantagens:

- **Sintaxe mais curta**: A sintaxe das *arrow functions* é geralmente mais concisa do que a sintaxe tradicional de funções, tornando o código mais limpo e fácil de ler.
- **Léxico ‘this‘**: As arrow functions herdam o valor de ‘this‘ do escopo em que são definidas, o que pode simplificar o tratamento de ‘this‘ em contextos como ‘map‘, ‘filter‘ e outros métodos de array.
- **Expressões de função concisas**: Para funções simples, você pode omitir as chaves e a palavra-chave ’return’, tornando a sintaxe ainda mais concisa.

7.7.1 Estrutura básica

```
(parâmetros) => { corpo da função }
```

```
// Função tradicional
function soma(a, b) {
  return a + b;
}

// Arrow function
const soma = (a, b) => a + b;
```

7.7.2 Características importantes

- Não possuem o próprio ‘this‘: O valor de ‘this‘ dentro de uma arrow function é o mesmo que no escopo onde ela foi definida.
- Não podem ser usadas como construtores: Você não pode usar ‘new‘ para criar objetos com arrow functions.
- Não possuem ‘arguments‘: Se você precisar acessar todos os argumentos de uma função, use o operador rest (‘...args‘)
- Corpo conciso: Se o corpo da função consiste em apenas uma expressão, você pode omitir as chaves e a palavra-chave ‘return

7.7.3 Quando usar arrow functions

- Callbacks curtas: Arrow functions são ideais para callbacks em métodos como ‘map‘, ‘filter‘, ‘reduce‘ e outros.
- Funções anônimas: Quando você precisa de uma função que não será reutilizada, as arrow functions oferecem uma sintaxe mais concisa.

- Quando o valor de ‘this‘ é importante: Se você precisa controlar o valor de ‘this‘ dentro de uma função, as arrow functions podem ser úteis.

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

// Dobrar cada número usando arrow function
const numerosDobrados = numeros.map(num => num * 2);

console.log(numerosDobrados); // [2, 4, 6, 8, 10]
```

8

Node.js

8.1 Introdução

O Node.js é uma plataforma baseada no V8 JavaScript Engine para desenvolvimento de aplicações *server-side*, usando JavaScript tanto no cliente quanto no servidor. Suas principais características incluem:

- **Consistência**: Representações de linguagem e dados no cliente e servidor são idênticas.
- **Escalabilidade**: A arquitetura de única *thread* reduz uso de memória e evita troca de contexto entre *threads*.
- **Desempenho**: Ótimo para tarefas leves, como *streaming*, *websockets*, e filas de E/S, mas não para operações computacionalmente intensivas.

Veja um exemplo simples [1] seguir:

```
var http = require('http');
 http.createServer(function(req,res) {
   res.writeHead(200, { 'Content-Type':
   'text/plain; charset=utf-8' });
   res.end('Olá mundo!\n');
 }).listen(3000);
 console.log('Servidor iniciado em
 localhost:3000. Ctrl+C para encerrar...');
```

Neste exemplo, cria-se um servidor simples que responde "Olá mundo!" para cada requisição recebida na porta 3000 e pode ser executado com os comandos a seguir:

```
$ node arquivo.js
Servidor iniciado em localhost:3000. Ctrl+C para encerrar...
```

[1] https://tableless.com.br/o-que-nodejs-primeiros-passos-com-node-js/

Em outro terminal executar

```
$ curl http://localhost:3000/
> Olá mundo!
```

O Node.js é especialmente útil se a E/S for provavelmente o seu gargalo (ou seja, o servidor não está fazendo muita coisa), como *chat* em tempo real, APIs leves e microsserviços, *streaming* de dados, *websockets*. De um modo geral, o Node.js é ideal para tarefas leves e em tempo real e uma má escolha para processamento pesado, como cálculos matemáticos complexos ou manipulação de grandes arquivos. O seu site pode ser encontrado no endereço: http://www.nodejs.org

8.2 npm

8.2.1 Introdução

O ‘npm‘ (Node Package Manager) é o gerenciador de pacotes oficial para o ambiente de desenvolvimento Node.js. Ele é utilizado para instalar, compartilhar e gerenciar dependências (pacotes) de código, tanto em projetos de front-end quanto back-end, facilitando o trabalho dos desenvolvedores ao lidar com bibliotecas externas. O Node Package Manager (npm) fornece duas funcionalidades principais:

- Acesso a repositórios *online* para pacotes e módulos node.js que podem ser localizados em http://search.nodejs.org
- Utilitários de linha de comando para instalar pacotes node.js, fazendo o gerenciamento de versões e dependências.

8.2.2 Instalação do Node.js e npm

Antes de usar o ‘npm‘, é necessário instalar o Node.js, pois o ‘npm‘ é distribuído junto com ele. Ao instalar o Node.js, o ‘npm‘ é instalado automaticamente. Dependendo do sistema operacional, você pode fazer o *download* de um instalador do site oficial do Node.js ou usar um gerenciador de pacotes específico do sistema, como ’dnf’ no Fedora, ’apt’ no Ubuntu ou ’brew’ no macOS.

Instalação alternativa

O **nvm** permite gerenciar diferentes versões do **Node.js** no mesmo sistema, o que é útil para projetos que exigem versões específicas.

1. Para instalar o **nvm**

- Abra o terminal
- Execute o seguinte comando para baixar e instalar o NVM:

```
$ curl -o- https://raw.githubusercontent.com/
nvm-sh/nvm/v0.39.3/install.sh | bash
```

Nota: Verifique a versão mais recente no repositório oficial do [nvm-sh] (`https://github.com/nvm-sh/nvm`).

- Após a instalação, adicione o **nvm** ao seu shell. Dependendo do shell que você está utilizando, adicione o seguinte no arquivo de configuração do shell (ex.: ‘ /.bashrc‘, ‘ /.zshrc‘, ‘ /.profile‘, etc.):]

```
$ export NVM_DIR="$HOME/.nvm" [ -s "$NVM_DIR/nvm.sh" ]
&& \. "$NVM_DIR/nvm.sh"
```

- Para aplicar as mudanças imediatamente, recarregue o shell com o comando:

```
$ source ~/.bashrc
$ source ~/.zshrc
```

- Verifique se o **nvm** foi instalado corretamente executando:

```
$ nvm --version
        \end{verbatim}
```

2. Agora que o nvm está instalado, você pode instalar a versão desejada do Node.js. Por exemplo, para instalar a versão mais recente da LTS (Long-Term Support):

```
$nvm install --lts
```

Após a instalação, o **nvm** também instala automaticamente o **npm** (gerenciador de pacotes do Node.js). Para verificar as versões instaladas de **node.js** e **npm**, execute:

```
$node --version
$npm --version
```

3. Caso você tenha múltiplas versões do **Node.js** instaladas, pode definir a versão padrão com o comando:

```
$nvm use <versao>
```

Para definir a versão padrão a ser usada sempre que abrir um novo terminal, execute:

```
$nvm alias default <versão>
```

4. Com o **npm** instalado, você pode instalar pacotes globalmente ou localmente para o projeto. Por exemplo:

- Para instalar um pacote globalmente:

```
$ npm install -g <nome-do-pacote>
```

- Para instalar um pacote localmente em um projeto:

```
$ npm install <nome-do-pacote>
```

5. Para instalar uma nova versão do **Node.js**, basta usar o comando **nvm install** seguido da versão desejada, por exemplo:

```
$ nvm install <versão>
```

6. Se precisar mudar entre diferentes versões de **Node.js**, use o comando:

```
$ nvm use <versão>
```

Esse procedimento oferece flexibilidade no gerenciamento de versões do Node.js, facilitando o trabalho em múltiplos projetos com requisitos diferentes.

8.2.3 Comandos básicos do npm

1. 'npm -v': Para verificar qual a sua versão, abra a console (ou execute cmd no windows)

```
$ npm -v
  8.5.1
```

2. 'npm init': Inicia um novo projeto Node.js, criando um arquivo 'package.json' onde são listadas as dependências e informações do projeto. Com a opção '-y', o comando usa as configurações padrão.

```
$ mkdir teste
$ cd teste
$ npm init -y
```

3. 'npm install' ou 'npm i': Instala todas as dependências listadas no arquivo 'package.json' ou instala pacotes específicos. Existem diferentes formas de instalar pacotes:

   - Localmente: Instala pacotes apenas para o projeto atual (cria uma pasta 'node_modules'). Executáveis vão para node_modules/bin/, e as páginas de manual não são instaladas. É o modo padrão.
   - Globalmente: Instala o pacote de forma global, tornando-o acessível em qualquer projeto. Para isso, utilize o comando com o flag '-g'. Isso coloca os módulos em {prefix}/lib/node_modules, e coloca os arquivos executáveis em {prefix}/bin, onde {prefix} usualmente é algo como /usr/local. Isso também instala as páginas de

manual em {prefix}/share/man, se houver.

```
$ npm install express
$ npm install -g create-react-app
```

4. ‘npm install –save‘: Adiciona o pacote ao ‘package.json‘ como uma dependência do projeto. A flag ‘–save‘ foi descontinuada nas versões mais recentes do npm, pois já é o comportamento padrão.

5. ‘npm install –save-dev‘: Instala o pacote como dependência de desenvolvimento, ou seja, ele será necessário apenas durante o desenvolvimento (não em produção).

```
$ npm install eslint --save-dev
```

6. ‘npm uninstall <package>‘: Remove um pacote instalado e o retira das dependências no ‘package.json‘.

7. ‘npm update‘: Atualiza os pacotes instalados para suas versões mais recentes, conforme permitido pela semântica de versão especificada no ‘package.json‘.

8. ‘npm run <script>‘: Executa scripts definidos na seção ‘"scripts"‘ do arquivo ‘package.json‘. Por exemplo, se você definiu um script ‘"start": "node app.js"‘, pode executá-lo com o comando:

```
$ npm run start
```

9. ‘npm audit‘: Verifica as dependências instaladas em busca de vulnerabilidades de segurança e sugere correções.

10. ‘npm outdated‘: Lista pacotes que estão desatualizados no projeto, permitindo que você veja rapidamente quais pacotes precisam ser atualizados.

11. ‘npm ci‘: Instala as dependências usando o arquivo ‘package-lock.json‘, garantindo a instalação exata das versões especificadas. É útil em ambientes de integração contínua (CI).

12. ’npm prune’: Para remover todos os pacotes que o seu projeto não dependa, segundo o informado pelo arquivo package.json, use o comando:

```
$ npm prune
```

13. ’npm list’ ou ’npm ls’: listam todos os pacotes instalados.

```
$ npm list
$ npm ls
```

14. 'npm search': Verifica se tem algum pacote com nome ou que realize funções similares de manipulação e diretório.

```
$ npm search mkdir
```

8.2.4 Configuração do npm

O 'npm' permite várias configurações personalizadas, que podem ser feitas de três maneiras: globalmente, localmente (para o projeto) ou por meio de variáveis de ambiente. Algumas das configurações mais comuns são:

- Configuração de proxies: Se você estiver atrás de um proxy, pode configurá-lo com os comandos:

```
$ npm config set proxy http://proxyurl:port
$ npm config set https-proxy http://proxyurl:port
```

- Configuração de diretório global: Por padrão, os pacotes globais são instalados em um diretório padrão do sistema. Você pode alterar este diretório:

```
$ npm config set prefix /path/to/directory
```

- Configuração de repositório privado: Se sua organização usa um repositório privado para pacotes npm, você pode configurar o registro:

```
$ npm config set registry https://my-private-registry.com
```

- Ver configuração atual**: Para visualizar as configurações atuais do 'npm', use:

```
$ npm config list
```

8.2.5 Arquivos 'package.json' e 'package-lock.json'

- 'package.json': Este arquivo contém as informações do projeto, incluindo dependências, scripts e metadados. É essencial para compartilhar projetos com outros desenvolvedores ou para implantar em produção.

- 'package-lock.json': Este arquivo é gerado automaticamente e registra as versões exatas das dependências instaladas, garantindo consistência entre instalações em diferentes máquinas.

Pacotes e dependências instalados globalmente são armazenados em um diretório do sistema. Em geral, a regra padrão é a seguinte:

- Se você está instalando que você deseja usar apenas no seu programas, usando require ('pacote'), então instale-o localmente, na raiz do seu projeto.
- Se você está instalando alguma coisa que você quer usar no seu *shell*, ou na linha de comando ou algo assim, instale-o globalmente, de modo que os binários possam ser encontrados no caminho definido pela variável de ambiente PATH.

8.3 Gerenciando a Cache

Para atualizar um pacote que já está instalado, o seguinte comando pode ser utilizado:

```
$ npm update johnny-five
```

Quando o **npm** instala um pacote ele guarda uma cópia deste pacote, de modo que na próxima vez que você precisar deste pacote, não seja necessário ir à internet novamente.

O diretório onde esta cópia é guardada é ~/.npm no Posix, ou %AppData%/npm-cache no Windows.

Eventualmente este diretório pode ficar cheio com pacotes velhos, então é necessário que ele seja limpo de vez em quando com o comando:

```
$ npm cache clean
```

Para utilizar um módulo no seu arquivo js, inclua a linha como a seguir:

```
var express  = require ('prompt');
```

As regras de onde "require" encontra os arquivos podem ser um pouco complexas, mas uma regra simples é que se o arquivo não começa com "./" ou "/", então ele é considerado um módulo global (e o caminho de instalação do "Node" local é verificado), ou é uma dependência local na pasta "node_modules" do projeto local.

Se o arquivo começa com "./" é considerado um arquivo relativo ao diretório do arquivo que chamou "require". Se o arquivo começa com "/", ele é considerado um caminho absoluto.

OBSERVAÇÃO: você pode omitir a extensão ".js" e a função "require" irá automaticamente anexá-la se necessário. Veja um exemplo bem simples a seguir:

```
  var prompt = require('prompt');
  prompt.start();
  prompt.get(['usuario', 'email'], function (err, result) {
```

```
  if (err) { return onErr(err); }
  console.log('Entrada na linha de comando recebida:');
  console.log('  Usuário: ' + result.usuario);
  console.log('  Email: ' + result.email);
});
function onErr(err) {
  console.log(err);
  return 1;
}
```

8.4 Semântica de Versão (‘semver‘)

No ‘package.json‘, as versões das dependências seguem o padrão de Semântica de Versão (semver), que é dividido em três partes: ‘major.minor.patch‘ (exemplo: ‘1.2.3‘). Os operadores ^ e ~ são frequentemente usados para especificar faixas de versões:

- ^ **1.2.3**: Permite atualizações sem quebrar a compatibilidade, como ‘1.3.0‘, mas não ‘2.0.0‘.
- ~ **1.2.3**: Permite atualizações de correção dentro da versão especificada, como ‘1.2.4‘, mas não ‘1.3.0‘.

8.5 Publicação de pacotes

Para publicar pacotes no repositório do npm:

1. Crie uma conta no site do npm (https://www.npmjs.com/).
2. Autentique-se no terminal com o comando:

```
$ npm login
```

3. Publique o pacote com o comando:

```
$ npm publish
```

8.6 Dicas e boas práticas

- Bloqueio de versão: Use o ‘package-lock.json‘ para garantir consistência entre diferentes ambientes.
- Automação de scripts: Utilize a seção de scripts no ‘package.json‘ para automatizar tarefas comuns como testes, builds, e linting.

- Segurança: Sempre verifique vulnerabilidades com o comando ‘npm audit‘ e mantenha suas dependências atualizadas.
- Gerenciamento de pacotes globalmente: Use o flag ‘-g‘ com cuidado, preferindo sempre que possível a instalação local de pacotes para evitar conflitos.

9

WebSockets e Socket.IO

9.1 Introdução

Na implementação de mensagens em tempo real em um aplicativo, a escolha do método correto é essencial para garantir a entrega ágil e a confiabilidade que seus usuários esperam.

Duas opções disponíveis são **Socket.IO** e **WebSocket**. O **Socket.IO** é uma biblioteca de mensagens avançada para aplicações JavaScript/NodeJS, enquanto o WebSocket é um protocolo de comunicação em tempo real de nível inferior, usado em bibliotecas como o próprio Socket.IO.

O **WebSocket** é um padrão da indústria que descreve uma forma de clientes e servidores trocarem mensagens em tempo real, que vem acompanhado de uma API que permite acessar esse protocolo. Isso é o diferencial do Socket.IO, que, como veremos mais adiante, é uma biblioteca completa de mensagens.

Vamos apresentar a seguir mais detalhes sobre essas duas ferramentas.

9.2 WebSocket

A API WebSocket, suportada pela maioria dos navegadores modernos, estende o protocolo para clientes web, fornecendo métodos para criar objetos WebSocket, gerenciar conexões, enviar e receber mensagens e lidar com eventos disparados pelo servidor WebSocket.

Mas qual o papel exato do WebSocket na infraestrutura de um aplicativo? Para entender melhor, vejamos suas principais características, vantagens e desvantagens. As principais características do Websocket são:

- **Comunicação bidirecional**: Conexões WebSocket são *full duplex*, o que significa que mensagens podem ser trocadas em ambas as direções (cliente-servidor), sem bloqueios.
- **Conexões persistentes**: O WebSocket mantém a conexão aberta pelo tempo que for necessário, ou até que algum problema de rede a interrompa.

- **Suporte amplo**: Como tecnologia padrão para web, WebSocket é suportado em navegadores, linguagens de *backend* e dispositivos IoT.
- **Foco na entrega de mensagens discretas**: O WebSocket lida com mensagens autossuficientes, como uma mensagem de chat enviada para todos os participantes de um grupo.

As suas principais vantagens são:

- **Baixa latência**: O WebSocket minimiza a latência ao manter uma única conexão aberta, reduzindo o tempo de overhead de novas conexões HTTP.
- **Flexibilidade na implementação**: O WebSocket, sendo um protocolo, impõe poucas restrições sobre como sua aplicação é construída.
- **Suporte a diferentes formatos de mensagens**: A codificação das mensagens fica a critério do desenvolvedor.

As principais desvantagens do WebSocket são:

- **Apenas um protocolo**: O WebSocket é um padrão, não uma implementação. Você precisará escolher uma biblioteca para implementá-lo, como ‘ws‘ ou ‘websockets‘ do Python, ou optar por uma biblioteca mais rica, como Socket.IO.
- **Gerenciamento de infraestrutura**: Você precisa construir e manter a infraestrutura para garantir baixa latência e alta disponibilidade.
- **Reconexões manuais**: O WebSocket não reconecta automaticamente quando uma conexão cai. Esse código precisa ser escrito manualmente.
- **Conexões bloqueadas**: Ambientes como redes corporativas com servidores proxy podem bloquear conexões WebSocket.
- **Escalabilidade complexa**: Escalar WebSocket horizontalmente pode ser desafiador, já que é um protocolo com estado.

9.3 Socket.IO

O Socket.IO é uma biblioteca de mensagens em tempo real para desenvolvedores JavaScript baseada em WebSocket, que habilita comunicação bidirecional baseada em eventos em tempo real. Assim como o WebSocket, oferece mensagens de baixa latência e *full duplex*, mas, além de especificar um padrão, Socket.IO oferece uma biblioteca tanto para o cliente quanto para o *backend*. O **Socket.IO** é uma biblioteca *open-source* que consiste em:

- Uma biblioteca para o servidor Node.js;
- Uma biblioteca de cliente JavaScript para o navegador (ou um cliente Node.js).

Além disso, há implementações disponíveis para outras linguagens, como Java, C++, Swift, Dart, Python, .NET, Rust e PHP. A conexão do **Socket.IO** oferece funcionalidades adicionais para facilitar a implementação de comunicação em tempo real e pode ser estabelecida com diferentes métodos de transporte de baixo nível:

- HTTP long-polling
- WebSocket
- WebTransport

O **Socket.IO** vai escolher automaticamente a melhor opção disponível dependendo das características do navegador e da rede, já que algumas redes bloqueia as conexões WebSocket e/ou WebTransport. Assim como WebSocket, Socket.IO trabalha com mensagens discretas, adequadas para *chats* e outros tipos de mensagens. As vantagens do Socket.IO são:

- **Multiplexação**: O Socket.IO permite dividir uma única conexão em múltiplas lógicas de comunicação.: O Socket.IO tenta reconectar automaticamente quando uma conexão é interrompida.
- **Salas e *rooms***: Permite agrupar clientes em salas e transmitir mensagens para todos com uma única chamada.
- **Multiplexação**: O Socket.IO permite dividir uma única conexão em múltiplas lógicas de comunicação.

O Socket.IO apresentas as seguintes desvantagens:

- **Infraestrutura**: Você ainda precisa construir e manter a infraestrutura de *backend*.
- **Suporte limitado a várias regiões**: Socket.IO só pode operar em um único data center, o que pode aumentar a latência para usuários distantes.
- **Garantias de mensagens limitadas**: O Socket.IO garante apenas a entrega "no máximo uma vez", sem assegurar que todas as mensagens sejam entregues.
- **Não substitui diretamente WebSocket**: Socket.IO não é compatível com WebSocket diretamente.

As principais funcionalidades do Socket.IO são:

1. **Baseado em eventos**: O Socket.IO permite definir e responder a eventos personalizados, como "mensagem enviada"ou "usuário digitando".

2. **Confiabilidade**: O Socket.IO permite a comunicação mesmo quando há *proxies* e balanceadores de carga; ou *firewalls* pessoais e softwares antivírus.

   Para garantir essa confiabilidade, ele utiliza o **Engine.IO**, que primeiro estabelece uma conexão de *long-polling* e tenta atualizar para transportes melhores, como o **WebSocket**, quando possível.

3. **Suporte a Reconexão Automática**: Se o cliente for desconectado, ele tenta se reconectar automaticamente até que o servidor esteja disponível novamente. Existem opções configuráveis para ajustar o comportamento de reconexão

4. Detecção de Desconexão: O **Socket.IO** implementa um mecanismo de "*heartbeat*"(sinal de vida) no nível do **Engine.IO**, que permite ao servidor e ao cliente detectar quando um não está mais respondendo. Isso é feito com temporizadores no servidor e no cliente, configurados com os parâmetros ‘pingInterval‘ e ‘pingTimeout‘, trocados durante a fase inicial do protocolo de conexão.

5. **Suporte a Dados Binários**: é possível emitir qualquer estrutura de dados serializável, incluindo:

   - ArrayBuffer e Blob no navegador
   - ArrayBuffer e Buffer no Node.js

6. **API Simples e Conveniente**:aqui está um exemplo básico de uso da API:

```
io.on('connection', socket => {
  socket.emit('request', /* .. */); // Emite um evento para o socket
  io.emit('broadcast', /* .. */);   // Emite um evento para todos os
   sockets conectados
  socket.on('reply', () => { /* .. */ }); // Escuta por um evento
});
```

7. **Compatibilidade entre Navegadores**: a compatibilidade do **Socket.IO** com diversos navegadores é verificada por meio da plataforma **Sauce Labs**.

8. **Suporte a Multiplexação**: o **Socket.IO** permite a criação de múltiplos **Namespaces**, que atuam como canais de comunicação separados, mas compartilham a mesma conexão subjacente. Isso facilita a organização de assuntos dentro da aplicação.

9. **Suporte a Rooms (Salas)**: dentro de cada *namespace*, é possível criar "salas", que são canais arbitrários aos quais os *sockets* podem se juntar e sair. Com isso, é possível enviar

mensagens para todos os clientes em uma sala específica. Essa funcionalidade é útil, por exemplo, para enviar notificações a grupos de usuários ou a um único usuário em vários dispositivos.

10. **Não é uma Implementação de WebSocket**: apesar de utilizar o **WebSocket** como transporte sempre que possível, o **Socket.IO** adiciona metadados extras aos pacotes (como o tipo de pacote, o *namespace* e o ID de reconhecimento). Por isso, um cliente WebSocket puro não pode se conectar a um servidor **Socket.IO** e vice-versa.

Você pode instalar o **Socket.IO** usando npm ou yarn:

```
 // com npm
$ npm install socket.io

// com yarn
$ yarn add socket.io
```

9.4 Exemplos de Uso

9.4.1 Usando com um servidor HTTP básico em Node.js:

```
const server = require('http').createServer();
const io = require('socket.io')(server);

io.on('connection', client => {
  client.on('event', data => { /* ... */ });
  client.on('disconnect', () => { /* ... */ });
});

server.listen(3000);
```

9.4.2 Uso Standalone (Autônomo):

```
const io = require('socket.io')();

io.on('connection', client => { /* .. */ });

io.listen(3000);
```

9.4.3 Emitindo eventos

A ideia principal por trás do Socket.IO é que você pode enviar e receber qualquer evento que desejar, com qualquer dado que quiser. Qualquer objeto que possa ser codificado como JSON é válido, e dados binários também são suportados.

Vamos fazer com que, quando o usuário digitar uma mensagem, o servidor a receba como um evento de mensagem de chat. A seção de script no arquivo 'index.html' agora deve ficar assim:

```
 <script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();

  var form = document.getElementById('form');
  var input = document.getElementById('input');

  form.addEventListener('submit', function(e) {
    e.preventDefault();
    if (input.value) {
      socket.emit('chat message', input.value);
      input.value = '';
    }
  });
</script>
```

E o arquivo index.js vai imprimir a mensagem:

```
io.on('connection', (socket) => {
  socket.on('chat message', (msg) => {
    console.log('message: ' + msg);
  });
});
```

9.4.4 Broadcast

O próximo objetivo é emitir um evento do servidor para o restante dos usuários. De modo a enviar um evento para todos, o Socket.IO dispõe do método **io.emit()**.

```
io.emit('some event', { someProperty: 'some value', otherProperty: 'other
    value' }); // Isso vai emitir o evento para todos os sockets conectados
```

Se você quiser enviar a mensagem para todos, exceto para o emissor, a flag broadcast pode ser utilizada:

```
io.on('connection', (socket) => {
  socket.broadcast.emit('hi');
});
```

Neste exemplo, por simplicidade, nós enviamos o evento para todos, incluindo o emissor:

```
io.on('connection', (socket) => {
  socket.on('chat message', (msg) => {
    io.emit('chat message', msg);
  });
});
```

E no lado do cliente quando nós capturamos o evento de mensagem de chat, ele será incluído na página. O código completo do cliente em JavaScript é o seguinte:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();

  var messages = document.getElementById('messages');
  var form = document.getElementById('form');
  var input = document.getElementById('input');

  form.addEventListener('submit', function(e) {
    e.preventDefault();
    if (input.value) {
      socket.emit('chat message', input.value);
      input.value = '';
    }
  });

  socket.on('chat message', function(msg) {
    var item = document.createElement('li');
    item.textContent = msg;
    messages.appendChild(item);
    window.scrollTo(0, document.body.scrollHeight);
  });
</script>
```

E isso completa a nossa aplicação de *chat*, com apenas 20 linhas de código!

9.4.5 Usando com o Express:

Desde a versão 3.0 do **Socket.IO**, as aplicações **Express** tornaram-se funções de manipulador de requisições que você passa para uma instância de servidor HTTP. Portanto, você deve passar o servidor HTTP para o **Socket.IO**, e não a função da aplicação **Express**. Além disso, certifique-se de chamar '.listen' no servidor, e não na aplicação.

```
const app = require('express')();
const server = require('http').createServer(app);
const io = require('socket.io')(server);
```

```
io.on('connection', () => { /* .. */ });

server.listen(3000);
```

Por exemplo:

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

// Cria a aplicação Express
const app = express();

// Cria o servidor HTTP
const server = http.createServer(app);

// Inicializa o Socket.IO e conecta ao servidor HTTP
const io = new Server(server);

// Define a rota principal para servir a página HTML
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Evento de conexão: acontece quando um cliente se conecta ao Socket.IO
io.on('connection', (socket) => {
  console.log('Um usuário se conectou');

  // Recebe uma mensagem do cliente e reenvia a todos os clientes
   conectados
  socket.on('chat message', (msg) => {
    console.log('Mensagem recebida: ' + msg);
    io.emit('chat message', msg); // Envia a mensagem para todos os
   clientes
  });

  // Evento de desconexão: acontece quando um cliente se desconecta
  socket.on('disconnect', () => {
    console.log('Um usuário se desconectou');
  });
});

// Configura o servidor para escutar na porta 3000
server.listen(3000, () => {
  console.log('Servidor rodando na porta 3000');
});
```

server.js

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Chat com Socket.IO</title>
  <script src="/socket.io/socket.io.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.
   min.js"></script>
  <style>
    ul { list-style-type: none; padding: 0; }
    li { padding: 8px; margin-bottom: 5px; background-color: #f3f3f3;
   border-radius: 5px; }
    input { padding: 10px; width: 80%; margin-right: 5px; }
    button { padding: 10px; }
  </style>
</head>
<body>
  <h1>Chat com Socket.IO</h1>
  <ul id="messages"></ul>
  <form id="form" action="">
    <input id="message" autocomplete="off" placeholder="Digite uma mensagem
   ..." /><button>Enviar</button>
  </form>

  <script>
    $(function() {
      var socket = io(); // Conecta com o servidor

      // Envia a mensagem quando o formulário é enviado
      $('#form').submit(function(e) {
        e.preventDefault(); // Evita o comportamento padrão de recarregar a
    página
        var msg = $('#message').val();
        socket.emit('chat message', msg); // Envia a mensagem para o
   servidor
        $('#message').val(''); // Limpa o campo de mensagem
        return false;
      });

      // Recebe a mensagem do servidor e exibe na lista
      socket.on('chat message', function(msg) {
        $('#messages').append($('<li>').text(msg));
      });
    });
  </script>
```

```
</body>
</html>
```

index.html

9.4.6 Usando com o Koa:

O **Koa** funciona de maneira semelhante ao **Express**, mas com a função ‘callback()‘ para expor o manipulador de requisições.

```
const app = require('koa')();
const server = require('http').createServer(app.callback());
const io = require('socket.io')(server);

io.on('connection', () => { /* .. */ });

server.listen(3000);
```

9.4.7 Usando com o Fastify:

Para integrar o **Socket.IO** com o **Fastify**, é necessário registrar o plugin ‘fastify-socket.io‘, que cria um decorador chamado ‘io‘.

```
const app = require('fastify')();
app.register(require('fastify-socket.io'));

app.ready().then(() => {
    app.io.on('connection', () => { /* .. */ });
});

app.listen(3000);
```

O **Socket.IO** oferece uma solução robusta para comunicação bidirecional em tempo real, sendo confiável, fácil de usar e compatível com uma ampla gama de navegadores e dispositivos. Seu suporte a *namespaces*, salas, reconexão automática e detecção de desconexões torna-o ideal para aplicações em grande escala que necessitam de interatividade em tempo real, como chats, jogos e monitoramento em tempo real.

10

Firmata

O **Firmata** é um protocolo de comunicação que permite que um computador se conecte a microcontroladores (como o Arduino, NodeMCU, etc.) e os controle diretamente, enviando comandos e recebendo dados em tempo real. Ele é muito utilizado para controlar dispositivos de hardware sem a necessidade de carregar um código personalizado diretamente no microcontrolador, pois o computador age como o cérebro, enquanto o microcontrolador serve como um intermediário para executar as tarefas de *hardware*.

Os comandos podem ser enviados do computador para o microcontrolador com uso do barramento USB ou via WiFi, se o controlador possuir este tipo de interface.

10.1 Funcionamento do Firmata

1. **Protocolo Baseado em MIDI**: O Firmata é baseado no protocolo MIDI, um padrão amplamente utilizado para comunicação em instrumentos musicais digitais. Isso significa que os dados são transmitidos como uma sequência de bytes, com comandos específicos para diferentes ações.

2. **Interatividade em Tempo Real**: Ele permite que você controle pinos de entrada e saída (como LEDs, sensores, motores, etc.) de um microcontrolador a partir de um computador em tempo real. Você pode ler o estado de um sensor ou controlar um motor diretamente de um software no computador, sem reprogramar o Arduino.

3. Código Padrão no Microcontrolador**: O Arduino (ou outro microcontrolador) executa um "sketch"padrão, geralmente chamado de **StandardFirmata**. Esse código permite que o dispositivo interprete comandos que vêm do computador via porta serial (ou outros protocolos de comunicação como Bluetooth ou Wi-Fi).

4. **Software de Controle no Computador**: No lado do computador, você pode usar bibliotecas em várias linguagens de programação (como Python, JavaScript, Processing, etc.) para enviar comandos ao Arduino usando o protocolo Firmata.

10.1.1 Principais Recursos do Firmata

- Controle de Pinos: Controle individual de pinos digitais e analógicos. Você pode configurar um pino como entrada ou saída, ler o valor de um pino de entrada ou escrever valores (HIGH ou LOW) em um pino de saída.
- PWM e Servo: Controle de pinos PWM para modulação de sinal (geralmente usada para controle de brilho de LEDs ou a velocidade de motores) e controle de servos.
- Leitura de Sensores: O Firmata pode ler valores de sensores analógicos e digitais em tempo real e enviá-los para o computador.
- Mensagens Customizadas: Firmata suporta a criação de mensagens personalizadas, o que permite expandir as funcionalidades conforme suas necessidades.

10.1.2 Vantagens do Firmata

O Firmata é ideal para prototipagem rápida, já que você não precisa carregar um novo código no Arduino toda vez que faz uma mudança. O controle é feito diretamente pelo computador. Ele também facilita a integração de microcontroladores com softwares avançados como Max/MSP, Pure Data, Python, JavaScript (Node.js) ou Processing, permitindo que a lógica e o processamento pesado ocorram no computador. O Firmata é compatível com múltiplos sistemas operacionais e pode ser usado em várias plataformas (Windows, macOS, Linux, etc.).

10.1.3 Limitações do Firmata

Como depende de comunicação entre o computador e o microcontrolador, pode haver uma latência perceptível em aplicações que exigem respostas ultra-rápidas. O Firmata é ideal para controle direto de *hardware* simples, mas não é muito eficiente para projetos mais complexos que exigem processamento local no microcontrolador. Nesses casos, é mais eficiente programar o microcontrolador diretamente.

10.1.4 Exemplo de Uso

No caso de um microcontrolador controlado via Firmata e um computador executando uma aplicação em Python, o fluxo seria:

Primeiro, você precisa instalar o Johnny-Five e o Firmata no seu projeto Node.js. Se ainda não tiver, crie um novo projeto Node.js e execute o seguinte comando:

```
$ npm install johnny-five
```

No microcontrolador, você deve carregar o sketch StandardFirmata (disponível na IDE do Arduino) para permitir a comunicação com o Johnny-Five via Firmata.

Este exemplo acende e apaga um LED conectado ao pino 13 do microcontrolador a cada segundo:

```
const five = require("johnny-five");  // Importa a biblioteca Johnny-Five
const board = new five.Board();       // Cria uma nova instância da placa

board.on("ready", function() {
  // Cria um objeto LED conectado ao pino 13
  const led = new five.Led(13);

  // Pisca o LED: Liga por 1 segundo, desliga por 1 segundo
  led.blink(1000);
});
```

11

JSON

11.1 Introdução

O JSON (JavaScript Object Notation) é um formato de troca de dados leve, baseado em texto e de fácil leitura tanto para humanos quanto para máquinas. Amplamente utilizado em APIs e serviços web, o JSON se tornou uma escolha popular para a transferência de dados entre servidores e clientes, devido à sua simplicidade e eficiência. Embora tenha suas raízes na linguagem JavaScript, o JSON é independente de linguagem, sendo compatível com diversas linguagens de programação modernas, como C, C++, Java, Python, PHP, entre outras.

11.2 Características

As suas características principais são:

- **Legibilidade**: O JSON é fácil de ler e escrever, usando uma sintaxe simples e minimalista. Essa legibilidade facilita a depuração e a criação manual de dados.
- **Interoperabilidade**: Como é um formato de texto, pode ser interpretado por praticamente qualquer linguagem de programação, desde que siga as convenções universais de sintaxe.
- **Baseado em JavaScript**: Embora o JSON se baseie em um subconjunto da sintaxe do JavaScript, ele é totalmente independente de qualquer linguagem específica, o que o torna amplamente interoperável.

O JSON utiliza duas estruturas principais, amplamente suportadas em várias linguagens de programação:

- **Coleção de Pares Nome/Valor**: Esta estrutura é frequentemente chamada de "objeto"em JSON. Um objeto é representado por um conjunto desordenado de pares nome/-

valor. Em várias linguagens, isso pode ser comparado a um objeto, registro, estrutura, ou dicionário.

Exemplo de um objeto em JSON:

```
{
  "nome": "João",
  "idade": 30,
  "cidade": "São Paulo"
}
```

- **Lista Ordenada de Valores**: Esta estrutura é conhecida como "array" em JSON. Um *array* é uma coleção ordenada de valores, o que pode ser comparado a um vetor, lista, ou sequência em várias linguagens. Exemplo de um *array* em JSON:

```
["banana", "maçã", "laranja"]
```

Essas duas estruturas (objetos e arrays) permitem a representação de praticamente qualquer dado em JSON, desde simples listas até estruturas de dados mais complexas.

11.3 Comparação com Outros Formatos

O JSON muitas vezes é comparado com o XML, que também é utilizado para troca de dados. No entanto, o JSON é geralmente preferido por ser mais compacto e fácil de ler, enquanto o XML pode ser mais verboso. Além disso, o JSON é mais adequado para uso com JavaScript, uma vez que pode ser facilmente convertido para objetos JavaScript usando o método JSON.parse() e convertido de volta para JSON com JSON.stringify().

Exemplo Prático Suponha que você tenha um serviço web que retorna os detalhes de um usuário em formato JSON:

```
{
  "nome": "Carlos",
  "idade": 35,
  "enderecos": [
    {
      "tipo": "residencial",
      "cidade": "Rio de Janeiro",
      "estado": "RJ"
    },
    {
      "tipo": "trabalho",
      "cidade": "São Paulo",
```

```
      "estado": "SP"
    }
  ],
  "ativo": true
}
```

Com esse formato, o cliente pode facilmente interpretar os dados em diferentes linguagens e usá-los conforme necessário.

O JSON é uma das formas mais populares e eficientes de troca de dados no desenvolvimento web e em APIs. Sua simplicidade, flexibilidade e compatibilidade com diversas linguagens de programação o tornam uma escolha ideal para desenvolvedores que precisam de um formato leve, fácil de ler e altamente interoperável.

11.4 Sintaxe JSON

Em JSON, os dados são apresentados desta forma:

- **Objeto**: Um objeto em JSON é delimitado por chaves e consiste em pares *nome/valor*. Cada par é separado por vírgula, e o *nome* e o *valor* são separados por dois pontos :. Os nomes devem ser *strings* (entre aspas duplas) e os valores podem ser de diferentes tipos, como *strings*, *números*, *arrays*, ou outros objetos (Figura 11.1).

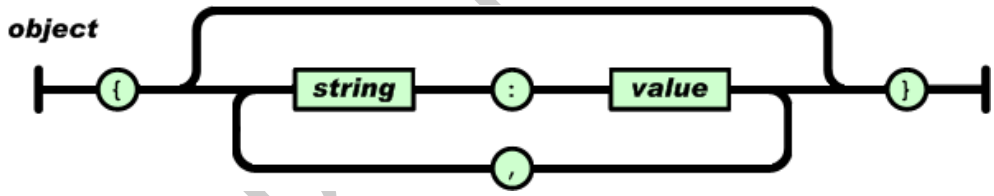

Figura 11.1: object

Exemplo:

```
{
  "produto": "Laptop",
  "preco": 1500.00,
  "disponivel": true
}
```

- **Array**: Um array em JSON é delimitado por colchetes [] e contém uma lista ordenada de valores, separados por vírgula (Figura 11.2).

Exemplo:

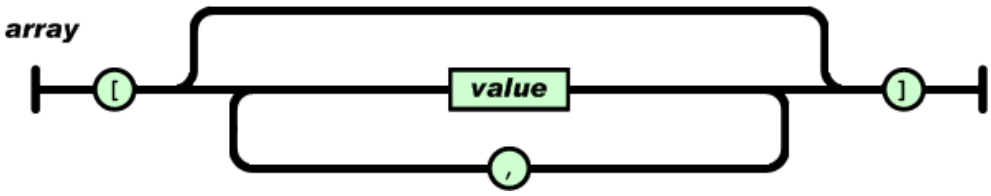

Figura 11.2: array

```
[
  "vermelho",
  "verde",
  "azul"
]
```

- Valores: Os valores em JSON podem ser (Figura 11.3):
  - Strings (entre aspas duplas, como "exemplo"),
  - Números (inteiros ou decimais, sem sufixos especiais),
  - Booleanos (true ou false),
  - null (para representar a ausência de valor),
  - Objetos (delimitados por),
  - Arrays (delimitados por []).

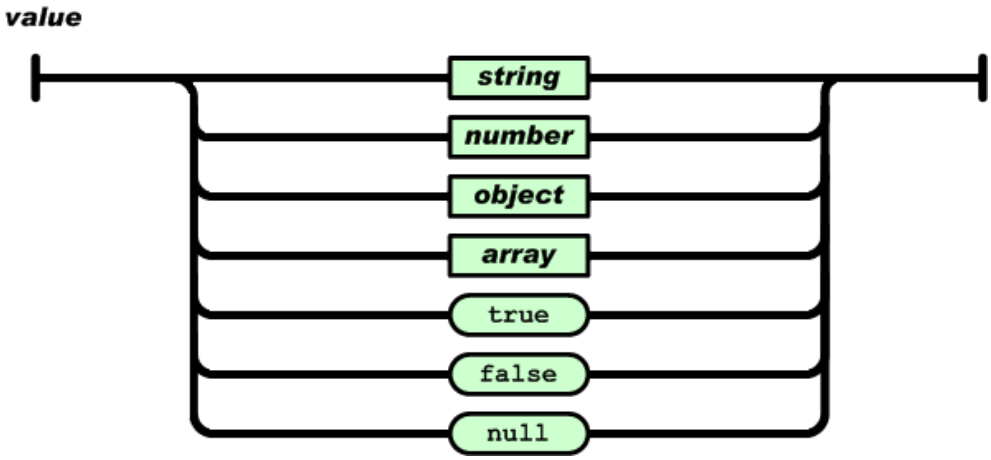

Figura 11.3: value

Exemplo de valores aninhados:

```
{
  "nome": "Ana",
  "idade": 28,
```

```
  "habilidades": ["programação", "design", "gestão"],
  "emprego": {
    "empresa": "TechCorp",
    "cargo": "Desenvolvedora"
  },
  "ativo": true,
  "salario": null
}
```

- Uma *string* é uma coleção de nenhum ou mais caracteres unicode, entre aspas duplas, usando barras invertidas como caractere de escape. Um caractere é representado como um simples caractere de *string*. Uma cadeia de caracteres é parecida com uma cadeia de caracteres em C ou Java. (Figura 11.4)

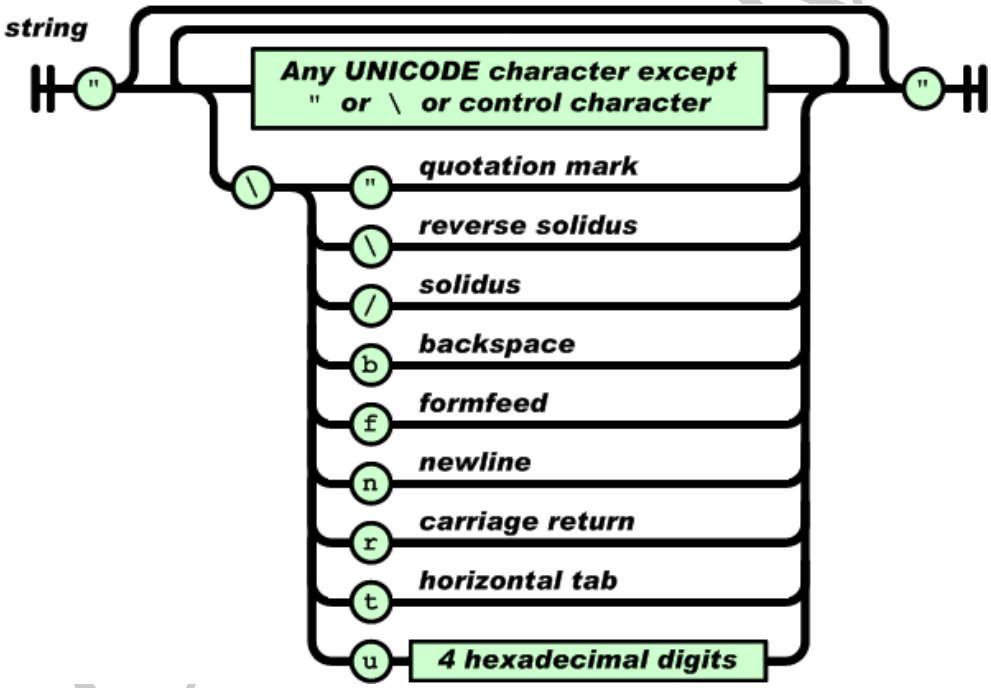

Figura 11.4: string

- Um número é similar a um número em C ou Java, exceto que os números octais ou hexadecimais não são usados. (Figura 11.5)

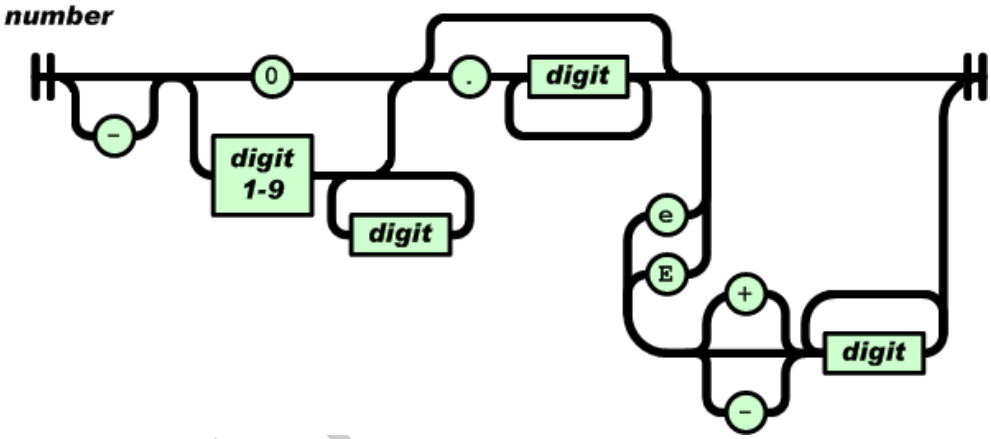

Figura 11.5: number

```
object
    {}
    { members }
members
    pair
    pair , members
pair
    string : value
array
    []
    [ elements ]
elements
    value
    value , elements
value
    string
    number
    object
    array
    true
    false
    null
string
    ""
    " chars "
chars
    char
    char chars
char
    any-Unicode-character-
        except-''-or-\-or-
        control-character
    \''
    \\
    \/
    \b
    \f
    \n
    \r
    \t
    \u four-hex-digits
number
    int
    int frac
    int exp
    int frac exp
```

```
int
    digit
    digit1-9 digits
    - digit
    - digit1-9 digits
frac
    . digits
exp
    e digits
digits
    digit
    digit digits
e
    e
    e+
    e-
    E
    E+
    E-
```

Referências

1. Mozilla Developer Network Web Docs - JavaScript Basics Este guia da MDN Web Docs oferece uma introdução abrangente aos conceitos básicos de JavaScript, ideal para iniciantes. Ele cobre os fundamentos da linguagem, como variáveis, operadores, estruturas de controle, funções e manipulação do DOM. O MDN é uma das principais referências para desenvolvedores e é mantido pela comunidade Mozilla, oferecendo uma documentação bem detalhada e atualizada. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript`

2. Node.js - NPM - Fornece uma introdução ao Node Package Manager (NPM) para desenvolvedores Node.js. Cobre tópicos como a instalação do NPM, gerenciamento de pacotes, e como usá-lo para instalar, atualizar e desinstalar dependências em projetos Node.js. Além disso, há exemplos práticos de como configurar arquivos 'package.json' e usar comandos comuns do NPM para lidar com bibliotecas e ferramentas no ecossistema Node.js. `https://www.tutorialspoint.com/nodejs/nodejs_npm.htm`

3. NCE - UFRJ: Guia de JavaScript O site do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro (NCE-UFRJ) oferece um guia introdutório sobre JavaScript em português. Este material é ideal para iniciantes que buscam aprender os conceitos básicos da linguagem, com exemplos práticos e explicações simples. O guia cobre desde a sintaxe básica até o uso de eventos, formulários e a integração com HTML. `http://www.nce.ufrj.br/ginape/js/`

4. Google's JavaScript Style Guide Este guia de estilo do Google para JavaScript é uma referência importante para desenvolvedores que desejam seguir as melhores práticas de codificação. Ele aborda convenções de nomenclatura, formatação, organização de código e boas práticas gerais, promovendo um código mais legível e consistente. O guia é amplamente utilizado em projetos profissionais para garantir a qualidade do código. `https://google.github.io/styleguide/jsguide.html`

5. JSON.org - Documentação em Português Este site oficial do JSON fornece uma visão geral da sintaxe JSON e suas especificações. É uma referência fundamental para desenvolvedores que utilizam JSON como formato de troca de dados em suas aplicações. A

página em português explica as estruturas de dados suportadas, como objetos e arrays, e fornece exemplos claros para facilitar a compreensão. `http://www.json.org`

6. JavaScript: The Definitive Guide Escrito por David Flanagan, este livro é considerado a "bíblia"do JavaScript, abrangendo desde os fundamentos até tópicos avançados. A obra é altamente recomendada para desenvolvedores que desejam uma compreensão profunda da linguagem. Ele cobre detalhes sobre o núcleo do JavaScript, APIs de navegador, manipulação do DOM e muito mais, com exemplos práticos que auxiliam na implementação. A última edição inclui tópicos modernos como ECMAScript 6+.

7. JavaScript Cookbook Escrito por Shelley Powers, este livro oferece soluções práticas para problemas comuns enfrentados no desenvolvimento com JavaScript. O JavaScript Cookbook segue um formato de "receitas", com exemplos que podem ser aplicados diretamente em projetos. O livro cobre desde a manipulação do DOM até o uso de APIs modernas, tornando-o uma excelente referência para quem já possui algum conhecimento na linguagem e quer aprimorar suas habilidades.

8. JavaScript Bible Escrito por Danny Goodman, Michael Morrison e outros, este é um dos livros mais antigos e abrangentes sobre JavaScript. A JavaScript Bible cobre a linguagem em detalhes, desde os fundamentos até tópicos avançados, incluindo técnicas de depuração, manipulação de eventos, e integração com outras tecnologias da web. É uma referência útil para desenvolvedores que desejam um conhecimento extenso e detalhado do JavaScript.

9. JavaScript Pocket Reference Também escrito por David Flanagan, este pequeno livro é um guia de referência rápida para desenvolvedores que precisam de uma consulta rápida e precisa sobre JavaScript. O JavaScript Pocket Reference fornece uma visão geral concisa da sintaxe da linguagem, APIs e métodos comumente usados, sendo ideal para consulta durante o desenvolvimento ou revisão de código.