



{ Javascript
Aula 05
<Módulo 06/>

Controle de tempo

A habilidade de gerenciar o tempo de forma precisa e dinâmica é essencial na criação de aplicações web interativas.

JavaScript, como uma linguagem de programação do lado do cliente, oferece poderosas funcionalidades para controle temporal, permitindo aos desenvolvedores criar experiências mais envolventes e responsivas.



Seja para criar animações, agendar tarefas assíncronas, ou controlar eventos em tempo real, compreender e utilizar eficientemente as capacidades de controle de tempo do JavaScript é fundamental.

Nesta aula, vamos aprender sobre os recursos disponíveis para manipulação de tempo no Javascript.

Métodos para controle de tempo

`setTimeout`

O método `setTimeout` cria um temporizador e executa uma função `callback` quando o cronômetro expira.

Sua sintaxe é mostrada abaixo:

```
const timeoutID = setTimeout(function ([param1], [param2], ...), delay, [param1], [param2], ...)
```



Por dentro do assunto!

O termo **método** significa que `setTimeout` representa uma ação atrelada a algum objeto. Este objeto é o objeto `window` do Javascript. Dessa forma, o método `setTimeout` pode também ser chamado da seguinte forma:

```
const timeoutID = window.setTimeout(function ([param1], [param2], ...), delay, [param1], [param2], ...)
```

Note que o método `setTimeout` recebe três tipos de parâmetros:

- `function ([param1], [param2], ...)` é a função `callback`, chamada pelo `setTimeout` quando o cronômetro expira
- `delay`: é o tempo, em milisegundos, decorrido desde a chamada do método `setTimeout` até a execução da função `callback`
- `[param1], [param2], ...`: são os parâmetros que devem ser passados à função `callback` quando for chamada pelo método `setTimeout`

Um `timeoutID` é retornado pela chamada do método `setTimeout`. Esse id é um identificador inteiro positivo que pode ser utilizado para cancelar a execução do `setTimeout`.



Por dentro do assunto!

Para refrescar a memória, uma **Callback** é uma função passada a outra função como argumento, como mostrado abaixo:

```
// greeting greets a newly arrived person.
function greeting(name) {
    alert(`Olá, ${name}!`);
}

// welcomePerson welcomes a person and greets his arrival.
function welcomePerson(greetingCallback) {
    const personName = prompt("Por favor, me diga seu nome!")

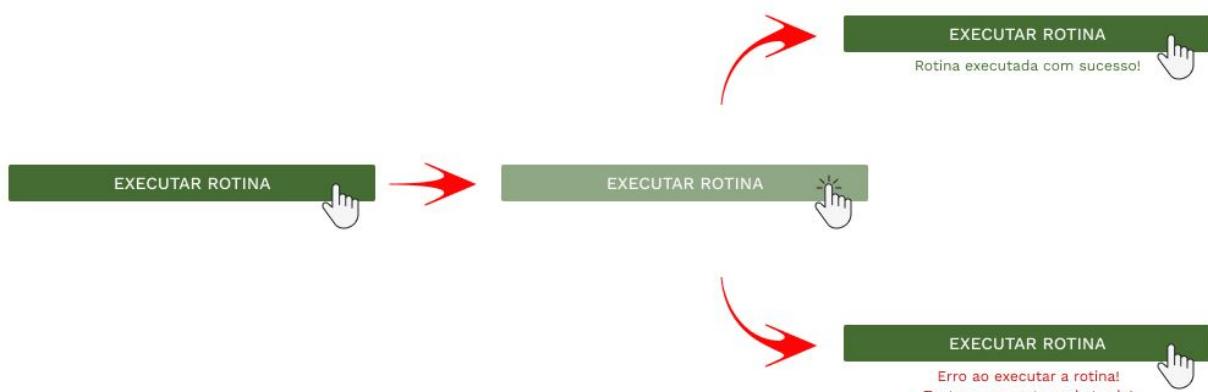
    greetingCallback(personName)
}
```

A função `greeting` é utilizada como callback na função `welcomePerson`.

Como exemplo, imagine que você esteja configurando um botão que, quando clicado chame uma rotina qualquer. Também imagine que a rotina está suscetível a retornar erros. O botão deverá se comportar da seguinte forma:

- se a execução da rotina ocorrer com sucesso, o botão deverá mostrar ao usuário a mensagem "Rotina executada com sucesso!", e fazer com que a mensagem desapareça depois de 3 segundos
- se a execução da rotina resultar em erro, o botão deverá mostrar ao usuário a mensagem "Erro ao executar a rotina! Tente novamente mais tarde!", e fazer com que a mensagem desapareça depois de 5 segundos

A imagem abaixo ilustra melhor a situação:



As seguintes funções poderiam ser utilizadas para implementar essa funcionalidade:

```
// routine executes the routine.
function routine() {
    ...
}

// showSuccessMessage shows the message of success in executing the routine.
function showSuccessMessage() {
    ...
}

// showErrorMessage shows the error message in executing the routine.
function showErrorMessage() {
    ...
}

// hideMessage hides the displayed message.
function hideMessage() {
    ...
}

document.querySelector('button').addEventListener('click', function() {
    try {
        routine();
        showSuccessMessage();
        setTimeout(hideMessage,3000);
    } catch {
        showErrorMessage();
        setTimeout(hideMessage,5000);
    }
}))
```

Cancelamento da execução do `setTimeout`

O cancelamento da execução do `setTimeout` pode ser realizado com o método `clearTimeout`, da seguinte forma:

```
clearTimeout(timeoutID);
```

`setInterval`

O método `setInterval` cria um temporizador e executa uma função repetidamente, com um tempo de espera fixo entre cada execução.

Sua sintaxe é mostrada abaixo:

```
const intervalID = setInterval(function ([param1], [param2], ...), delay, [param1], [param2], ...)
```

Note que o método `setInterval` recebe três tipos de parâmetros:

- `function ([param1], [param2], ...)` é a função `callback` que é chamada pelo `setInterval` a cada expiração do cronômetro
- `delay`: é o tempo de espera, em milisegundos, entre cada chamada da função `callback` pelo `setInterval`
- `[param1], [param2], ...`: são os parâmetros que devem ser passados à função `callback` quando for chamada pelo método `setInterval`

Além disso, um `intervalID` é retornado pela chamada do método `setInterval`. Esse id é um identificador inteiro positivo que pode ser utilizado para cancelar a execução do `setInterval`.

Um bom exemplo de uso para o `setInterval` é um painel que se responsabiliza em mostrar a cotação mais recente de uma moeda, tal como na figura abaixo:



fonte: google.com

Essa cotação pode ser atualizada em um intervalo de tempo fixo, por exemplo, a cada 30 segundos.

As seguintes funções poderiam ser utilizadas para implementar a funcionalidade, considerando que as informações necessárias são o **valor da cota** e o **instante de tempo** em que foi gerada:

```
// getQuote requests the value of the quote.
function requestQuote() {
    ...
}

setInterval(function () {
    const quote = requestQuote();

    document.querySelector('#quote-value').textContent = quote.value;
    document.querySelector('#quote-date').textContent = quote.timestamp;
}, 30000);
```

OBS: As funções acima não representam por completo o fluxo de uma **requisição web** e o tratamento necessário para as informações recebidas. Essas funções apenas exemplificam a possível solução para o problema.

Cancelamento da execução do `setTimeout`

O cancelamento da execução do `setInterval` pode ser realizado com o método `clearInterval`, da seguinte forma:

```
clearInterval(intervalID);
```

O construtor Audio

Para conseguir fazer os exercícios da aula, você precisará conhecer um construtor específico do Javascript: o construtor [Audio](#).

O construtor [Audio](#) retorna um elemento de áudio ([HTMLAudioElement](#)) que pode ser anexado ao documento ou utilizado fora da tela:

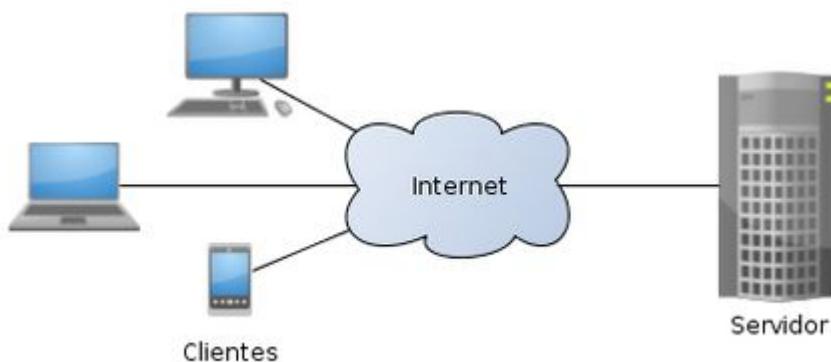
```
const audio = new Audio(url);
```

O parâmetro opcional [url](#) indica a fonte do arquivo de áudio.

Os métodos [play](#) e [pause](#) do construtor podem ser utilizados, respectivamente, para iniciar e pausar a execução do áudio.

Single Page Application (SPA)

O modelo atual da *internet* é o chamado **Cliente-Servidor**. Uma série de **dispositivos clientes** comunicam-se por uma **rede de computadores** com fornecedores de serviços, **servidores**.



fonte: pt.wikipedia.org

Um **dispositivo cliente** pode ser qualquer tipo de aparelho conectado à internet: *notebook*, *desktop*, *smartphone*, etc. Já os **servidores**, geralmente, são computadores dedicados para lidar com muitas **requisições de informações**.

A comunicação entre um dispositivo cliente e um servidor, segue o modelo de **requisições e respostas**, ou seja, o cliente solicita ou envia informação, e o servidor devolve uma resposta.



fonte: pt.wikipedia.org

Antes do advento do Javascript, em meados de 1996, a web era até então composta de **documentos estáticos**. Isto é, o navegador solicitava informação de um servidor web (documento HTML, arquivo de texto etc.), recebia a informação e apresentava ao usuário. Não era possível ter toda a dinâmica que as páginas web tem atualmente.

A principal ferramenta que possibilitava essa dinâmica, do lado cliente, eram os **formulários HTML**.

Você já notou que, um formulário HTML configurado com um botão tipo `submit` e um atributo `action`, quando o usuário clica no botão, a página é redirecionada? Se não percebeu, faça você mesmo essa verificação!

Ao clicar no botão `submit`, o formulário faz uma requisição web ao endereço configurado no atributo `action` do formulário. A informação recebida é então renderizada no navegador numa nova página web. Assim eram as requisições web!

Com o advento do Javascript, uma nova forma de fazer requisições web foi criada: **AJAX (Asynchronous JavaScript And XML)**.



AJAX é uma técnica de desenvolvimento web na qual um cliente web busca conteúdo do servidor, fazendo **solicitações HTTP assíncronas** e usa o conteúdo obtido para atualizar partes relevantes da página, sem exigir um carregamento completo da página, como os formulários HTML até então exigiam.

Por dentro do assunto!

Solicitações HTTP assíncronas são operações realizadas pelo JavaScript sem travar a execução do código.

Com o **AJAX** foi possível que qualquer parte de uma página web fosse alterada sem a necessidade de recarregamento. Isso foi uma enorme evolução na web, que tornou as páginas **dinâmicas** e **responsivas** e abriu um leque de possibilidades. Hoje, é possível fazer infinitas tarefas pela Internet, de forma dinâmica e intuitiva.

Neste ponto, surge um novo conceito: uma página que não precisa ser recarregada para oferecer os serviços que se propõe a oferecer é normalmente chamada de **Single Page Application (SPA)**.

Uma **SPA** é uma implementação de aplicativo na web que carrega apenas um único documento e, em seguida, atualiza o conteúdo do corpo desse único documento por meio de **APIs JavaScript**, quando um conteúdo diferente deve ser mostrado.

Vamos aprender a construir uma **SPA** utilizando JavaScript.

Por dentro do assunto!

Uma **API JavaScript** é uma implementação utilizada para a construção de páginas web.

Por exemplo, o **Document Object Model (DOM)** é uma API que conecta páginas web (documentos HTML) a scripts ou linguagens de programação, e representa a estrutura de um documento HTML.

Numa API Web, pode haver muitas interfaces, que nada mais são do que contratos de ditam como uma API deve ser utilizada. Um exemplo bem conhecido é a interface

window na API DOM. Essa interface dita, por exemplo, que se quisermos buscar um elemento HTML no DOM da página, podemos utilizar o método querySelector do objeto document:

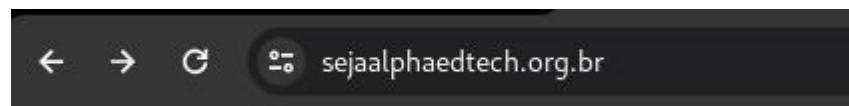
```
const element = window.document.querySelector("selector");
```

Há muitas outras APIs que podem ser utilizadas. Vamos conheceremos a API Fetch, utilizada atualmente para fazer requisições a servidores web.

A API History

Um ponto importante a entender, é que uma SPA busca se comportar como uma página web comum. Isso significa que, comportamentos comuns a páginas web devem ser mantidos ao desenvolvermos esse tipo de aplicação.

Uma característica comum aos sistemas web é a semântica da URL exibida na barra de endereços do navegador:



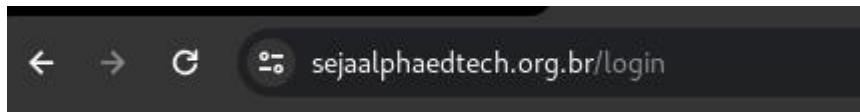
A URL exibida deve buscar expressar o que representa a página web apresentada. Na imagem acima, a URL indica que o conteúdo da página provém um meio de entrar em uma org chamada alphaedtech.

Ao clicar no botão "Acessar Portal" na página mostrada, é exibido um modal de login:



The image shows a light gray modal window with a white background. At the top center, it has a green button labeled "ENTRAR". Below this, there is a label "Email ou Usuário" followed by a text input field containing the placeholder "E-mail ou usuário". Next, there is a label "Senha" followed by an empty text input field. Below the password field is a small checkbox labeled "Manter conectado". At the bottom of the modal is a green "Login" button. To the right of the "Login" button, the text "Cadastre sua senha" is visible.

Nesse momento, a URL exibida na página é alterada:

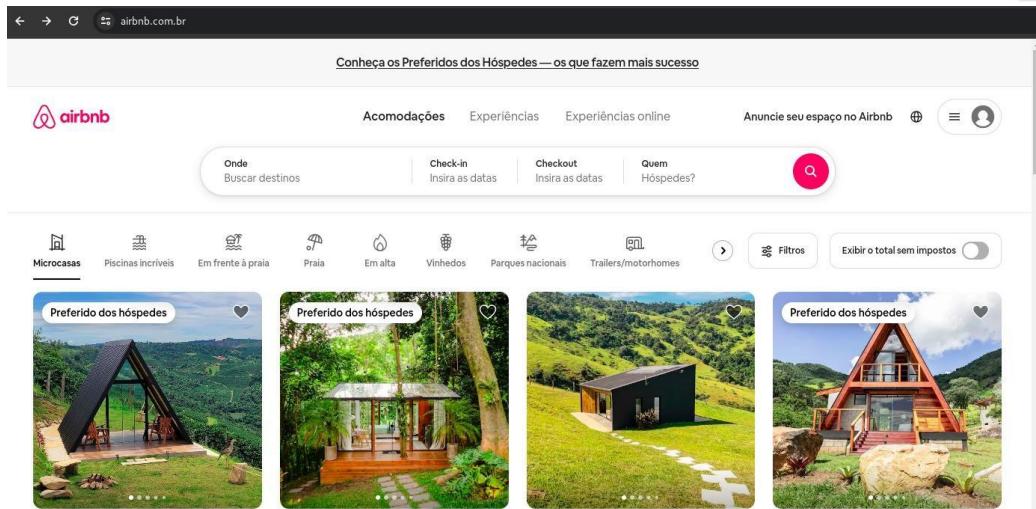


Agora, a **URL** demonstra claramente que o conteúdo mostrado representa uma forma de fazer login no sistema da organização.

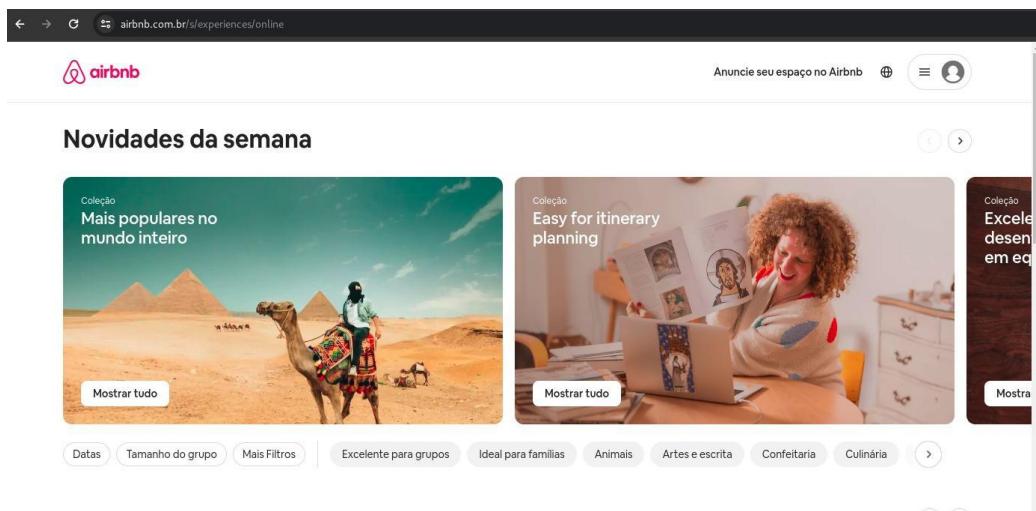
Note que a ação de clicar no botão "Acessar Portal" resultou num recarregamento da página *web*. Isso indica que a aplicação não é uma **SPA**.

Mas qual é o comportamento esperado para uma **SPA** aqui?

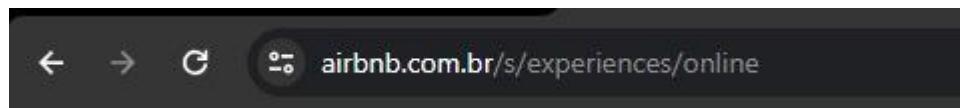
Vamos acessar o site da empresa Airbnb (<https://www.airbnb.com.br/>)!



Na página mostrada, clique no link "Experiências Online" e fique atento ao que acontece com a página *web*:



Conseguiu notar que não houve recarregamento da página pelo navegador, mesmo que quase todo o conteúdo da página tenha sido alterado? Além disso, observe que a url foi alterada de acordo com a nova informação apresentada:



Esse é comportamento de uma **SPA**, o conteúdo relevante é alterado, mas não é necessário recarregamento da página.

Outro detalhe importante é que, se desejármos voltar à página anterior, a **SPA** deverá mostrar o conteúdo apresentado para a última url acessada e alterar a url mostrada na barra de endereços, de acordo com o novo conteúdo.

Para configurar esse comportamento, precisamos entender como o navegador lida com a barra de endereços.

Imagine que você tenha um local dedicado para armazenar os endereços acessados por um usuário durante uma sessão do navegador. Vamos chamá-lo de **history**:



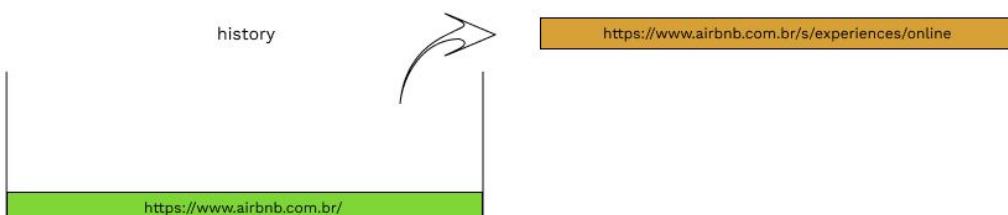
Inicialmente, o **history** da seção está vazio. Quando o usuário acessa a primeira página na seção, um endereço é adicionado ao **history**:



Se o usuário decidir ir a outra página a partir da primeira, mais um endereço é adicionado ao **history**:



E, para o caso em que o usuário deseje voltar à página anterior, o endereço que estiver armazenado mais acima no `history` será utilizado para redirecionar à página anterior:



Essa é a lógica representativa do tratamento de endereços feito pelo navegador!

Por dentro do assunto!

*Essa estrutura que chamamos de `history` é amplamente utilizada na área da computação. Trata-se da estrutura de dados conhecida como **stack**, ou **pilha**, traduzindo para Português.*

*Umas **stack** é uma estrutura capaz de armazenar uma série de elementos. Esses elementos são inseridos e removidos da **stack** por um único ponto de acesso.*

*O que caracteriza uma **stack** é esse fluxo de "empilhar" e "desempilhar" os elementos armazenados numa sequência conhecida como **FILO** (*First In, Last Out*), ou seja, o primeiro elemento a entrar na **stack** será o último a sair.*

Para que possamos controlar o comportamento da barra de endereços do navegador, podemos fazer uso da [API History](#). Essa API possui métodos dedicados ao controle de navegação, que podem ser utilizados para simular o redirecionamento de páginas em uma SPA, sem a necessidade de recarregamento da página. Para isso, utilizamos o objeto `history` da interface `window`:

```
window.history
```

Esse objeto possui, entre outros, os seguintes métodos:

- `back()`: **move para trás** no histórico de navegação, como se o usuário estivesse clicando no botão "Voltar" na barra de navegação, mas sem recarregamento
- `forward()`: **move para frente** no histórico de navegação, como se o usuário estivesse clicando no botão "Avançar" na barra de navegação, mas sem recarregamento
- `go(index)`: move para uma página específica do histórico de navegação, sem recarregamento:
 - o campo `index` especifica a distância entre a página atual e a página que deve ser mostrada, por exemplo:
 - `index = 1` representa a próxima página do histórico
 - `index = -1` representa a página anterior
 - `index = -2` representa a página anterior a esta
 - o número de páginas do histórico pode ser determinada pelo atributo `length` do objeto `history`

- o número de páginas do histórico pode ser determinada pelo atributo `length` do objeto `history`
- `pushState(stateObj, title, url)`: adiciona uma entrada no histórico de navegação:
 - ao ser chamado:
 - o endereço mostrado na barra de endereços é alterado, de acordo com o parâmetro `url`
 - a página não é recarregada
 - é possível acessar o objeto `stateObj` através do evento `popstate` ou do atributo `window.history.state`
 - o parâmetro `title` não é utilizado pelos navegadores
 - se o usuário clicar no botão "Voltar" do navegador após uma chamada ao método `pushState`, a `url` anterior será carregada
- `replaceState()`: modifica a entrada atual do histórico de navegação:
 - ao ser chamado:
 - o endereço mostrado na barra de endereços é alterado, de acordo com o parâmetro `url`
 - a página não é recarregada
 - é possível acessar o objeto `stateObj` através do evento `popstate` ou do atributo `window.history.state`
 - o parâmetro `title` não é utilizado pelos navegadores
 - ao modificar a entrada atual, não será possível acessá-la novamente pelo botão "Voltar" do navegador

Com esses métodos, é possível criar um mecanismo de navegação para uma **SPA**, sem causar recarregamento de página.

Por exemplo, imagine que você está atualmente na página principal do seu aplicativo web, construído no modelo **SPA**: `/home`. No conteúdo da página, existe um menu com os seguintes *links*: `about`, `contact` e `comments`:

- se o usuário clicar no *link* `about`, o método `pushState` é utilizado para alterar o histórico de navegação, adicionando uma url à `stack`:

```
history.pushState({}, "", "/about");
```

- estando na seção `about`, caso o usuário quisesse voltar, um botão "voltar", ou o próprio botão "Voltar" do navegador, é utilizado para voltar ao último registro do histórico de navegação:

```
history.back();
```

O evento `popstate`

O evento `popstate` é um evento Javascript que é disparado sempre que uma entrada do histórico de navegação é alterada.

Esse evento é disparado apenas após uma ação no navegador, tal como um clique no botão "Voltar" ou uma chamada ao método `history.back()`.

Roteamento de páginas

Considerando que em um aplicativo **SPA** o conteúdo apresentado deve estar de acordo com o endereço mostrado na barra de navegação, podemos utilizar essa informação para criar um **sistema de roteamento de páginas** em uma **SPA**.

Um **sistema de roteamento de páginas** é uma lógica que permite que o conteúdo a ser renderizado pelo navegador seja definido partir do **path** especificado na barra de endereços, como mostrado no exemplo abaixo:

```
const router = {
    "/": function () { return HomePage },
    "/about": function () { return aboutPage },
    "/contacts": function () { return contactsPage },
    "/comments": function () { return commentsPage }
};

const home = router["/"]();
```

No exemplo, de acordo com o valor passado como propriedade no objeto **router**, uma função será executada, retornando o elemento html que deve ser renderizado naquele momento. A **home page**, referenciada pelo path **"/"**, será retornada no final da execução do exemplo.

Eventos customizados

No Javascript, quase tudo pode ser feito com eventos.

Se precisamos executar uma rotina a partir de um clique de botão, usamos o evento **click**. Se precisamos detectar uma solicitação de envio de um formulário, usamos o evento **submit**.

Há casos em que precisamos configurar os nossos próprios eventos no Javascript. Para isso, podemos utilizar o construtor **CustomEvent**:

```
const event = new CustomEvent(name, options);
```

O construtor receber os seguintes parâmetros:

- **name**: nome do evento que criado
- **options**: um objeto que poderá ser acessado no ouvinte do evento

Por exemplo, considere a criação de um evento chamado **to-speak**, que configure uma mensagem a ser falada:

```
const toSpeak = new CustomEvent("to-speak", { message: "Olá!" });
```

Podemos configurar um ouvinte que processe o evento, da seguinte forma:

```
document.addEventListener("to-speak", function (event) {
  console.log(event.message);
});
```

Observe que, ao ser processado, o ouvinte terá acesso ao objeto `options` contido no evento, e acessa a propriedade `message` definida na criação do evento.

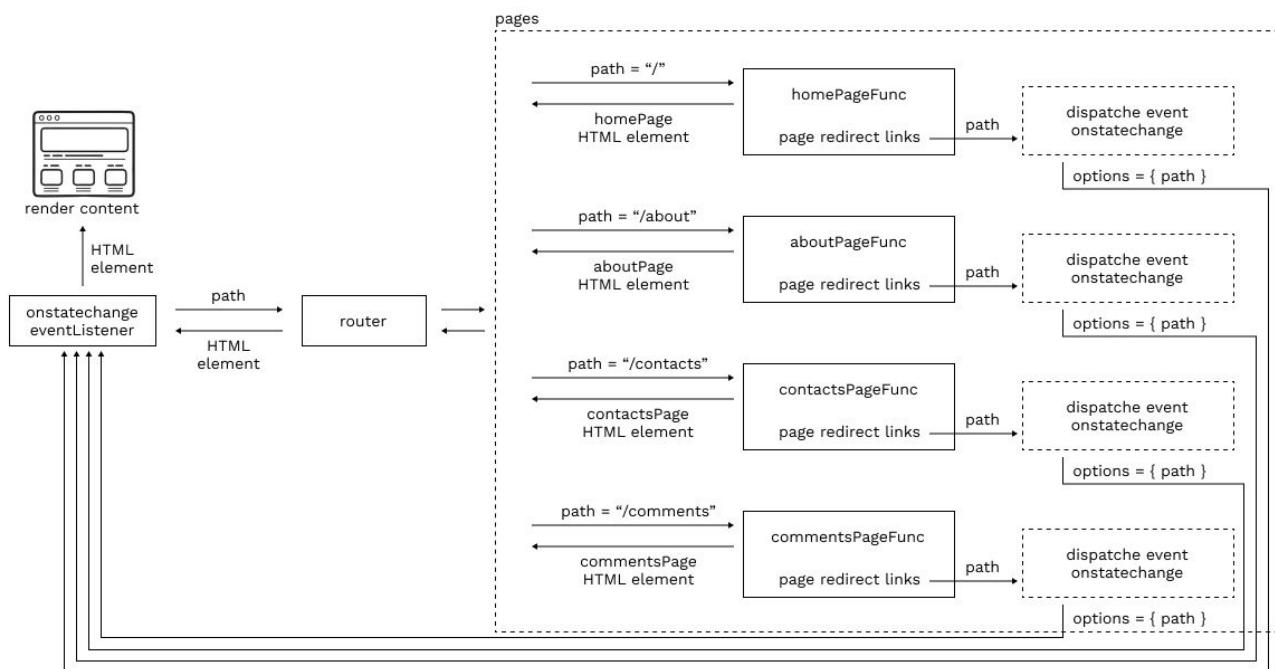
Um evento customizado não pode ser executado por uma ação direta do usuário no navegador. Para executar um evento customizado, devemos utilizar o método `dispatchEvent` do elemento HTML ao qual o evento foi atribuído:

```
document.dispatchEvent(toSpeak);
```

Utilizaremos eventos customizados para identificar alterações no histórico de navegação de uma **SPA**.

Eventos customizados

O seguinte esquema mostra como uma **SPA** pode ser construída com Javascript:

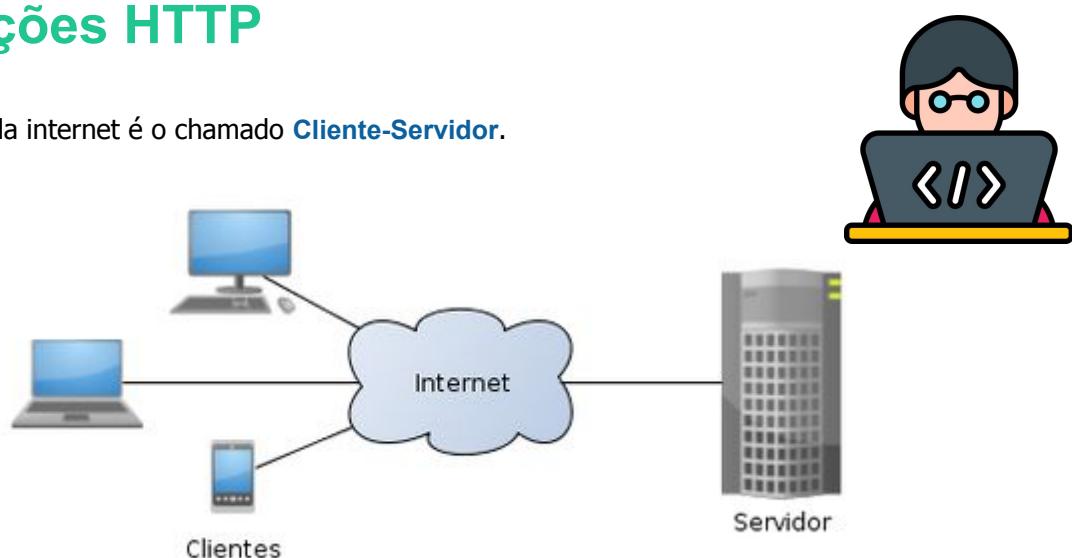


O ouvinte para um evento `onstatechange` deve se responsabilizar em obter o `path` da página que deve ser renderizada, solicita o elemento HTML da página a partir do `router` e renderiza o conteúdo obtido.

Nas páginas renderizadas, os `links` de redirecionamento dispatcham eventos customizados `onstatechange`, e mandam, no parâmetro `options`, o `path` da página que deve ser apresentada. Esse `path` será então utilizado pelo ouvinte do evento para obter o elemento HTML da página.

Requisições HTTP

O modelo atual da internet é o chamado **Cliente-Servidor**.



fonte: pt.wikipedia.org

Neste modelo, a comunicação entre um dispositivo cliente e um servidor, segue o modelo de **requisições** e **respostas**, ou seja, o cliente solicita ou envia informação, e o servidor devolve uma resposta.



fonte: pt.wikipedia.org

O **AJAX (Asynchronous JavaScript And XML)**, foi por muito tempo utilizado para o controle de requisições HTTP. Mas, o que há de mais atual para esse tipo de operação, é a **API Fetch**.

O protocolo HTTP

O **Hypertext Transfer Protocol (HTTP)**, é um protocolo de comunicação utilizado para sistemas de informação de **hipermídia, distribuídos** e **colaborativos**. Ele é a base para a comunicação de dados da **World Wide Web**.

Esse protocolo divide a comunicação entre dois pontos da rede em duas partes: **requisição** e **resposta**.

Uma **requisição HTTP** é estruturada com as seguintes informações:



fonte: mazer.dev

- **método**: define o tipo de requisição, sendo os mais comuns:
 - **GET**: requisições que buscam informações no servidor
 - **POST**: requisições que enviam informações ao servidor
 - **PUT**: requisições que alteram informações no servidor
 - **DELETE**: requisições que deletam informações no servidor
- **URI**: é o endereço do servidor ao qual a requisição é destinada
- **versão**: referência a versão do protocolo HTTP utilizado na comunicação
- **cabeçalho**: armazena uma série de metadados úteis para a requisição
- **corpo**: consiste na mensagem que se deseja mandar ao servidor (texto, objetos, arquivos, etc.)

Uma resposta HTTP contém as seguintes informações:



fonte: mazer.dev

- **versão**: referência a versão do protocolo HTTP utilizado na comunicação
- **status**: um conjunto "código descrição" que descreve o resultado da operação no servidor
- **cabeçalho**: armazena uma série de metadados úteis para a resposta
- **corpo**: consiste na mensagem que o servidor devolveu ao cliente como resultado da requisição (texto, objetos, arquivos, etc.)

- **corpo:** consiste na mensagem que o servidor devolveu ao cliente como resultado da requisição (texto, objetos, arquivos, etc.)

O campo de **status** da resposta para uma requisição HTTP detêm um significado muito importante. Cada código retornado é referenciado a um tipo de resultado na operação executada pelo servidor. Abaixo é listado alguns desses **status**:

- **200 OK:** representa a execução com sucesso da requisição no servidor
- **400 Bad Request:** significa que as informações enviadas pelo cliente ao servidor são inválidas
- **403 Forbidden:** indica que o cliente não tem autorização para acessar o recurso solicitado na requisição
- **404 Not Found:** significa que o recurso que o cliente buscou no servidor não foi encontrado (tal como em requisições à URIs inexistentes)
- **500 Internal Server Error:** indica que ocorreu um erro inesperado no servidor

A API Fetch

A **API Fetch** é uma interface do JavaScript utilizada para manipular requisições e respostas **HTTP**. Ela fornece o método global `fetch`, que provê uma maneira fácil e lógica para buscar recursos de forma assíncrona através da rede.

A sintaxe do método `fetch` é mostrada abaixo:

```
fetch(url, options);
```

Esse método recebe os seguintes parâmetros:

- **url:** representa o endereço web do servidor ao qual a requisição HTTP deve ser direcionada
- **options:** um objeto contendo as configurações para a requisição web, entre outras, método que deve ser utilizado (GET, POST, etc.), cabeçalhos, corpo, etc.

Como exemplo, imagine que queremos fazer uma consulta à página principal do Google. Podemos utilizar a seguinte estrutura:

```
fetch("https://agilemanifesto.org")
    .then(response => response.text())
    .then(content => {
        console.log(content);
    });
}
```

O seguinte conteúdo seria printado no console do navegador (recortado devido ao tamanho):

- `fetch("https://agilemanifesto.org/");` é a chamada ao método `fetch`, com a indicação do endereço do servidor, onde o conteúdo desejado está hospedado
- `.then(response => response.text());` método do `fetch` que espera a resposta do servidor, com o conteúdo desejado
- o parâmetro `response` representa a resposta da requisição, e contém, entre outros, os seguintes parâmetros:
 - `response.status`: código de status da resposta da requisição
 - `response.statusText`: descrição do status da resposta da requisição
 - `response.headers`: cabeçalhos da resposta da requisição
 - o método `text()` é utilizado para converter o corpo da resposta da requisição **HTTP** num texto Javascript
 - existem outros métodos similares a esse, por exemplo, se o corpo da requisição consistir num objeto JavaScript, é possível utilizar o método `json()` para convertê-lo
 - o resultado dessa conversão é acessado no próximo `then`
 - `.then(content => { console.log(content); })`; método do `fetch` que **espera** a conversão do corpo da resposta pelo método `text()`, no `then` anterior
 - o parâmetro `content` contém o resultado da conversão do corpo da requisição
 - a operação `console.log(content)` imprime o conteúdo do corpo da requisição no `console` do navegador

Por dentro do assunto!

*O método `fetch` retorna uma estrutura conhecida como **Promise**, sendo o método `then` pertencente a essa estrutura.*

Veremos a mesma com mais detalhes. Nesse momento, atente-se a apenas entender como se deve utilizar o método `fetch` para buscar informações em servidores web.

Observe que, dentro do segundo método `then`, é possível fazer o que for necessário com o conteúdo da resposta da requisição:

```
fetch("https://agilemanifesto.org")
  .then(response => response.text())
  .then(content => {
    const data = content;
    console.log(data);
  });

```

Porém, é importante entender que o parâmetro `content` só existe dentro do escopo do `then`. Assim, a seguinte estrutura não terá acesso ao conteúdo da resposta da requisição:

```
fetch("https://agilemanifesto.org")
  .then(response => response.text())
  .then(content => {
    const data = content;
    console.log(data);
  });

console.log(data); // undefined
```

Por dentro do assunto!

A execução do `fetch` consiste no que chamamos **JavaScript Assíncrono**.

A execução de **JavaScript Assíncrono** não se dá no fluxo principal do código. Assim, o código a seguir terá, provavelmente, a seguinte saída:

```
fetch("https://agilemanifesto.org")
  .then(response => response.text())
  .then(content => {
    const data = content;
    console.log(data);
  });

console.log(2);
```

```
2
1
```

Isso acontece por que, uma requisição web **não é instantânea**. Há alguns delays no processamento de uma requisição web, entre eles: o tempo de ida da requisição do cliente ao servidor, e o tempo de volta da resposta do servidor para o cliente.

Por conta disso, e de acordo com o **JavaScript Assíncrono**, a execução do `console.log(1)` será feita depois da execução do `console.log(2)`.

Em outras palavras, o fluxo principal do JavaScript não irá esperar que o `fetch` receba a resposta do servidor, para então continuar a sua execução.

APIs Web

Uma **API Web** é um sistema que se destina a disponibilizar informações, ou para executar tarefas.

Existem várias APIs úteis para uso. Dentre elas, utilizaremos nessa aulas as seguintes APIs:

- **API CEP**: <https://docs.awesomeapi.com.br/api-cep>
- **Google Maps**: <https://www.google.com/maps/search/?api=1¶meters>

A **API CEP** disponibiliza uma série de endpoints destinados a retornar informações atreladas a um CEP.

Observe que a API não irá possuir informações para todos os CEPs do Brasil.

Já o **Google Maps**, possui uma API que disponibiliza visualização de mapas.

Promises

Vamos aprofundar nosso conhecimento sobre **operações assíncronas**, ou seja, sobre execução de código de forma assíncrona.

Para relembrar, uma **operação assíncrona** é aquela que não ocorre de forma **instantânea**. Por exemplo, uma requisição **HTTP** feita com uso de `fetch`, no JavaScript, leva um determinado tempo para ser concluída, pois a requisição tem que passar por toda infraestrutura da internet para chegar ao servidor, assim como para que a sua resposta volte ao dispositivo do cliente.



fonte: pt.wikipedia.org

Uma operação assíncrona possui as seguintes características:

- existe um ponto de partida, mas não se sabe **quando** uma resposta será obtida
- também não se sabe se a resposta da operação será de **sucesso** ou de **falha**
- se o resultado da operação for de **sucesso**, a **resposta esperada** estará disponível para uso pelo sistema
- caso o resultado da operação seja de **falha**, um **motivo de falha** estará disponível para uso pelo sistema

Esse comportamento é implementado pelo objeto `Promise` do JavaScript.

Por definição, uma `Promise` é um **proxy** (um ponto, ou um alvo, onde deve ser concentrado um determinado valor), onde o **valor esperado**, mas não inicialmente conhecido, de uma **operação assíncrona** será concentrado.

O **valor esperado** mencionado acima pode ser um **sucesso**, indicando que a operação ocorreu como esperado, ou uma **falha**, indicando que um erro ocorreu durante a operação assíncrona.

No caso de **falha**, o **valor esperado** é considerado como um **motivo de falha**, ou seja, um **erro**.

Uma `Promise` pode estar num desses três estados:

- `pending`: estado inicial, nem **cumprida** nem **rejeitada**;
- `fulfilled`: significa que a operação foi **concluída com sucesso**;
- `rejected`: significa que a operação **falhou**.

Uma `Promise` é considerada **resolvida** se possuir estado **cumprida** ou **rejeitada**.

then e catch

Os métodos `then` e `catch` são utilizados para tratamento das respostas da operação assíncrona.

O método `then` recebe dois parâmetros, `onResolved` e `onRejected`:

```
fetch("https://cep.awesomeapi.com.br/json/05424020"
    .then(
        (response) => { return response.json() }, // onResolved
        (error) => { ... "tratamento do erro" ... } // onRejected
    )
    .then(
        (json) => { console.log(json) } // onResolved
        (error) => { ... "tratamento do erro" ... } // onRejected
    );
)
```

Os métodos `then` e `catch` são utilizados para tratamento das respostas da operação assíncrona.

O método `then` recebe dois parâmetros, `onResolved` e `onRejected`:

- o método `onResolved` será executado caso a operação resulte em **sucesso**.
- o método `onRejected` será executado caso a operação resulte em **falha**.

É mais comum que se utilize o método `then` sem o `onRejected`.

```
fetch("https://cep.awesomeapi.com.br/json/05424020"
    .then(
        (response) => { return response.json() } // onResolved
    )
    .then(
        (json) => { console.log(json) } // onResolved
    );
)
```

Para o tratamento dos erros que podem ocorrer na execução assíncrona, utiliza-se geralmente o método `catch`, que recebe apenas um parâmetro, `onRejected`.

```
fetch("https://cep.awesomeapi.com.br/json/05424020"
    .then(
        (response) => { return response.json() } // onResolved
    )
    .then(
        (json) => { console.log(json) } // onResolved
    )
    .catch(
        (error) => { ... "tratamento do erro" ... } // onRejected
    );
)
```

Manipulação do pipeline

É importante observar que, para que um método `then` seja chamado, uma `Promise` deve ser retornada pelo método anterior.

- no exemplo da página anterior, o método `response.json()` retorna uma nova `Promise`
- dessa forma, o segundo `then` captura essa `Promise` e realiza os procedimentos esperados
- se o `return` for removido, o segundo `then` não irá capturar a execução da promise retornada pelo método `response.json()`

É possível manipular o `pipeline` de execução com os métodos `Promise.resolve(<value>)` e `Promise.reject(<error>)`. Cada método resultará, respectivamente, numa `Promise` cumprida ou rejeitada.

```
fetch("https://cep.awesomeapi.com.br/json/05424020"
    .then(
        (response) => {
            if (response.ok) {
                return response.json();
            };
            return new Promise.reject("Erro na requisição dos dados!");
        }
    )
    .then(
        (json) => {
            return new Promise.resolve(json);
        }
    )
    .then(
        (json) => {
            return new Promise.resolve(json);
        }
    )
    .then(
        (json) => {
            return new Promise.resolve(json);
        }
    )
    .then(
        (json) => {
            return new Promise.resolve(json);
        }
    )
    .then(
        (json) => {
            return new Promise.resolve(json);
        }
    )
    .then(
        (data) => {
            console.log(data)
        }
    )
    .catch(
        (error) => { ... "tratamento do erro" ... }
    );
}
```

O objeto Promise

Até o momento, utilizamos o método `fetch` para exemplificar o comportamento de uma `Promise` em JavaScript.

Porém, o método `fetch` não é o único a retornar uma `Promise`. Existem diversas situações que necessitam do comportamento caracterizado por **Promises**. E para isso, às vezes se torna necessário **construir as nossas próprias Promises**.

Para criar **Promises**, com o comportamento que necesitamos, podemos utilizar o objeto `Promise` do JavaScript. Tal objeto **retorna uma Promise**, sendo possível então utilizar os métodos `then` e `catch` para gerir o `pipeline` de execução da mesma.

```
const value = 2;

const minhaPromise = new Promise((resolve, reject) => {
    if (x > 5) {
        resolve({ status: "OK" });
    } else {
        reject({ err: "ERROR" });
    }
});

minhaPromise
    .then(result => {
        console.log(result);
    })
    .catch(err => {
        console.log(err);
    });

// output expected:
// { err: "ERROR" }
```

Os parâmetros `resolve` e `reject`, recebidos na Promise `minhaPromise` são `callbacks` destinados a resolver ou rejeitar a execução da operação assíncrona.

A função `async`

O JavaScript oferece outra forma de criar **Promises**, utilizando o objeto `AsyncFunction`, resultado da declaração `async function`.

```
async function nome([param1, param2, ... paramN]) {
    instruções
}
```

Essa declaração permite que uma função **retorne uma Promise** como resultado. No exemplo a seguir, a função assíncrona **example** retorna uma **Promise resolvida**, pois nenhuma **exceção** é lançada em seu corpo.

```
async function example() {
    return 1;
}

const p = example();

p.then(result => {
    console.log("OK: ", result);
})
.catch(err => {
    console.log("ERR: ", err);
});

// output expected:
// OK: 1
```

Em contrapartida, no exemplo a seguir, a função assíncrona **example** retorna uma **Promise rejeitada**, pois no seu corpo uma **exceção** é lançada.

```
async function example() {
    throw 0;
}

example()
    .then(result => {
        console.log("OK: ", result);
    })
    .catch(err => {
        console.log("ERR: ", err);
    });

// output expected:
// ERR: 0
```

A função await

A declaração **await** pode ser usada dentro de funções assíncronas para **pausar** a execução do código, **esperando a resolução de uma Promise**.

No exemplo abaixo, a constante **result** espera a resolução da **Promise** retornada pela função **example**, e é preenchida com o valor retornado pela função, caso a **Promise** seja resolvida com sucesso.

```

async function example() {
    return 1;
}

async function init() {
    const result = await example();
    console.log("OK: ", result);
}

// output expected:
// OK: 1

```

É importante ressaltar que, se a função `example` **lançar uma exceção**, será necessário **capturar a exceção** lançada utilizando a estrutura `try ... catch`.

```

async function example() {
    throw 0;
}

async function init() {
    try {
        const result = await example();
        console.log("OK: ", result);
    } catch(err) {
        console.log("ERR: ", err);
    };
}

init();

// output expected:
// ERR: 0

```

Promise.all

O método `Promise.all()` pode ser utilizado para esperar por diversas promises ao mesmo tempo.

Esse método retorna uma única `Promise`, que será **resolvida** (ou **rejeitada**) quando **todas as promises** passadas como parâmetro forem resolvidas (ou rejeitadas).

```

async function example1() {
    return 1;
}

async function example2() {
    return 2;
}

async function example3() {
    return 3;
}

```

```
async function init() {
    const results = await Promise.all([ example1(), example2(), example3() ]);
    console.log("OK: ", results);
}

init();

// output expected:
// OK: [ 1, 2, 3 ]
```

Classes

Em desenvolvimento de sistemas, geralmente nos deparamos com a necessidade de criar uma função ou módulo que armazene uma série de informações e realize algumas ações com essas informações.



Por exemplo, imagine que o sistema em que você trabalhe necessite de um objeto que armazene dois números e possibilite a execução de várias operações com esses números.

Você poderia implementar o seguinte objeto:

```
const calc = {
    num1: null,
    num2: null,
    add: function() { return this.a + this.b },
    sub: function() { return this.a - this.b },
    mul: function() { return this.a * this.b },
    div: function() { return this.a / this.b }
}
```

Caso o sistema necessite do objeto, basta definir os atributos `num1` e `num2` do objeto e utilizar os métodos em seguida:

```
calc.num1 = 1;
calc.num2 = 2;

calc.add() // 3
calc.sub() // -1
calc.mul() // 2
calc.div() // 0.5
```

E se precisamos de duas calculadoras? Basta criar outro objeto semelhante, certo?

```
const calc2 = {
    num1: null,
    num2: null,
    add: function() { return this.a + this.b },
    sub: function() { return this.a - this.b },
    mul: function() { return this.a * this.b },
    div: function() { return this.a / this.b }
}
```

Quando lidamos com **Programação Orientada a Objetos (POO)**, nos deparamos com o termo **Classe**.

Uma **Classe** consiste num **tipo abstrato de dados** que **encapsula** dados e procedimentos que descrevem o conteúdo e o comportamento de entidades do mundo real, representadas por objetos.

No exemplo acima, uma Classe pode ser criada para abstrair a lógica necessária a uma calculadora, num objeto.

Nessa aula, iremos aprender a utilizar **Classes** em Javascript.

Por dentro do assunto!

Programação orientada a objetos (POO) é um paradigma de programação baseado no conceito de **objetos**, que podem conter dados na forma de **campos**, também conhecidos como **atributos**, e códigos, na forma de **procedimentos**, também conhecidos como **métodos**.

Closures

Um termo importante em programação, que já foi comentado em aulas anteriores é o **Escopo Léxico**, que representa a localização de um elemento dentro do código.

No exemplo abaixo, podemos ver três tipos de escopos diferentes:

```
let x = 2; // escopo global

{
    let y = 3; // escopo de bloco
}

function example() {
    let z = 4; // escopo de função
}
```

Uma **closure** é uma função que reconhece o escopo onde está posicionada.

No exemplo abaixo, a função `incrementar` reconhece o valor da variável `count` toda vez que é executada:

```
function contador() {
    let count = 0;

    function incrementar() {
        count++;
        console.log(count);
    }

    return incrementar;
}
```

```
const contar = contador();
contar() // 1
contar() // 2
contar() // 3
```

O resultado é que, após a execução da função `contador`, a função `incrementar` é armazenada na variável `contar`, sendo que o valor da variável `count` fará parte do escopo da função `incrementar`, possibilitando o incremento dessa variável a cada chamada da função.

Em resumo, a função `incrementar` "se lembra" do escopo em que foi criada ao utilizar a variável `count`.

Funções construtoras

O princípio proporcionado por **closures** permite que seja criado um comportamento muito comum em **Programação Orientada a Objetos (POO)**: o comportamento que define **Funções Construtoras**.

Uma **Função Construtora** é uma função que **retorna um objeto**, contendo **atributos** e **métodos**. Com **closures**, é possível **instanciar** um **objeto diferente** para cada chamada da função.

No exemplo abaixo, a cada chamada da função `Pokemon`, uma nova instância, ou seja, um novo objeto é criado, e cada uma é totalmente **independente** dos demais.

```
function Pokemon(_name, _attack) {
    let name = _name;
    let power = _attack;

    function speak() {
        return name + "!!!";
    }

    function attack() {
        return power;
    }

    return {
        speak: speak,
        attack: attack
    };
}

const p = Pokemon;
console.log(Pokemon);

const pikachu = new Pokemon("Pikachu", "Choque do trovão");
const bulbasaur = new Pokemon("Bulbasaur", "Chicote de cipó");
const charmander = new Pokemon("Charmander", "Lança chamas");

pikachu.speak(); // Pikachu!!!
charmander.speak(); // Bulbasaur!!!
bulbasaur.attack(); // Chicote de cipó
charmander.attack(); // Lança chamas
```

Funções construtoras

Uma **Classe** em Javascript é uma **abstração** para criação de objetos, utilizando protótipos, que por sua vez, consistem em mecanismos para que objetos possam **herdar** recursos de outros objetos.

Por dentro do assunto!

Por definição, abstração significa:

"Ação de abstrair, de analisar isoladamente um aspecto, contido num todo, sem ter em consideração sua relação com a realidade."

*O conceito de abstração é muito utilizado na programação. E resumo, significa **levar em consideração apenas o essencial, descartando tudo o que não for importante para o objetivo a ser atingido** .*

*Podemos fazer uma comparação com o retorno de uma função ou método, que podem analisar uma quantidade enorme de informação, mas no final, **somente o que é importante para o sistema deve ser retornado** .*

Para declarar uma **Classe** em Javascript, utiliza-se a palavra reservada `class`:

```
class Retangulo {
    constructor(_altura, _largura) {
        this.altura = _altura;
        this.largura = _largura;
    }
}

const ret = new Retangulo(100, 120);
console.log(ret);

// expected output:
// { altura: 100, largura: 120 }
```

A função `constructor` é uma rotina executada sempre que um objeto é instanciado por uma classe.

Observe que o resultado da chamada de uma classe é um objeto que possui os atributos definidos na instância do mesmo pela classe.

Da mesma forma como com **funções construtoras**, cada objeto instanciado com classes é totalmente independente dos demais objetos instanciados:

```

class Pokemon {
    constructor(_name, _power) {
        this.name = _name;
        this.power = _power;
    }

    speak() {
        return this.name + "!!!";
    }

    attack() {
        return this.power;
    }
}

const pikachu = new Pokemon("Pikachu", "Choque do trovão");
const bulbasaur = new Pokemon("Bulbasaur", "Chicote de cipó");
const charmander = new Pokemon("Charmander", "Lança chamas");

pikachu.speak(); // Pikachu!!!
bulbasaur.attack(); // Chicote de cipó
charmander.attack(); // Lança chamas

```

Encapsulamento

Uma propriedade muito importante em programação é o **Encapsulamento**, que consiste em **isolar** do usuário externo os **detalhes** não importantes para o mesmo.

Quando se trabalha com **Classe**s em Javascript, é comum que os usuários não tenham acesso direto aos atributos da classe, como no exemplo abaixo:

```

const Retangulo = class {
    constructor(altura, largura) {
        this.altura = altura;
        this.largura = largura;
    }
};

const ret = new Retangulo(100, 120);

console.log(ret.altura); // acesso direto ao atributo "altura"
console.log(ret.largura); // acesso direto ao atributo "largura"
// expected output:
// 100
// 120

ret.altura = 150; // acesso direto ao atributo "altura"
ret.largura = 80; // acesso direto ao atributo "largura"

console.log(ret.altura); // acesso direto ao atributo "altura"
console.log(ret.largura); // acesso direto ao atributo "largura"
// expected output:
// 150
// 80

```

Para evitar esse "acesso direto", devemos tornar os atributos que se deseja proteger como privados e adicionar **getters** e **setters** para possibilitar o acesso e alteração nos valores dos atributos:

```
const Retangulo = class {
    #altura // atributo privado
    #largura // atributo privado

    constructor(altura, largura) {
        this.#altura = altura;
        this.#largura = largura;
    }

    set altura(_altura) {
        this.#altura = _altura;
    }

    set largura(_largura) {
        this.#largura = _largura;
    }

    get altura() {
        return this.#altura;
    }

    get largura() {
        return this.#largura;
    }
};

const ret = new Retangulo(100, 120);

console.log(ret.altura); // acesso pelo método get altura
console.log(ret.largura); // acesso pelo método get largura

// expected output:
// 100
// 120

ret.altura = 150; // acesso pelo método set altura
ret.largura = 80; // acesso pelo método set largura

console.log(ret.altura); // acesso pelo método get altura
console.log(ret.largura); // acesso pelo método get largura

// expected output:
// 150
// 80
```

As cláusulas **get** e **set**, são usadas para criar métodos de classes que retornam ou atribuem valor, respectivamente, a uma propriedade de um objeto definido por classes.

Umas das vantagens dessa sintaxe é a possibilidade de adicionar validações aos métodos **set** para permitir que apenas valores válidos sejam dados aos atributos protegidos:

```

const Retangulo = class {
    #altura // atributo privado
    #largura // atributo privado

    constructor(altura, largura) {
        this.#altura = altura;
        this.#largura = largura;
    }

    set altura(_altura) {
        if (_altura <= 0) {
            return false;
        };

        this.#altura = _altura;
        return true;
    }

    set largura(_largura) {
        if (_largura <= 0) {
            return false;
        };

        this.#largura = _largura;
        return true;
    }

    get altura() {
        return this.#altura;
    }

    get largura() {
        return this.#largura;
    }
};

```

Subclasses

Outro comportamento bastante comum em **Programação Orientada a Objetos (POO)**, é a criação de classes como **extensões** de outras classes.

Por exemplo, imagine que você esteja trabalhando num sistema bancário. Nesse sistema pode existir uma classe **Account**, que representaria uma conta bancária:

```
class Account {  
    #number;  
    #owner;  
    #balance;  
  
    constructor(_number, _owner) {  
        this.#number = _number;  
        this.#owner = _owner;  
        this.#balance = 0;  
    }  
  
    get number() {  
        return this.#number;  
    }  
  
    get owner() {  
        return this.#owner;  
    }  
  
    get balance() {  
        return this.#balance;  
    }  
  
    greeting() {  
        return "Bem vindo a sua conta, " + this.#owner + "!";  
    }  
  
    withdraw(_amount) {  
        if (this.#balance < _amount) {  
            throw "not enough funds";  
        };  
  
        this.#balance -= _amount;  
    }  
  
    deposit(_amount) {  
        this.#balance += _amount;  
    }  
}
```

Mas, e se for preciso criar mais uma conta, que deve representar uma conta de rendimentos? Seria necessário criar mais uma classe, que implementasse também os atributos e métodos comuns a conta padrão e a conta de rendimentos?

```

class InvestmentAccount {
    #number;
    #owner;
    #balance;
    #tax;

    constructor(_number, _owner, _tax) {
        this.#number = _number;
        this.#owner = _owner;
        this.#balance = 0;
        this.#tax = _tax;
    }

    get number() {
        return this.#number;
    }

    get owner() {
        return this.#owner;
    }

    get balance() {
        return this.#balance;
    }

    get tax() {
        return this.#tax;
    }

    greeting() {
        return "Bem vindo a sua conta, " + this.#owner + "!";
    }

    withdraw(_amount) {
        if (this.#amount < _amount) {
            throw "not enough funds";
        };

        this.#balance -= _amount;
    }

    deposit(_amount) {
        this.#balance += _amount;
    }

    profits() {
        this.#balance *= this.#tax;
    }
}

```

Para que todo esse retrabalho seja evitado, podemos empregar o princípio da **Herança**.

O princípio de **Herança** permite que uma classe **herde** os atributos e métodos de outra classe. Isso é possível com a utilização da declaração `extends`, seguida do nome da **classe pai**, na declaração da nova classe.

O método `super` pode ser utilizado para acessar os atributos e métodos da **classe pai**.

Segundo esse princípio, a conta de rendimentos pode ser reescrita da seguinte forma:

```
class InvestmentAccount extends Account {
    #tax;

    constructor(_number, _owner, _tax) {
        super(_number, _owner);
        this.#tax = _tax;
    }

    get tax() {
        return this.#tax;
    }

    profits() {
        this.#balance *= this.#tax;
    }
}
```

Dessa forma, através da nova classe, é possível acessar tanto os atributos e métodos da mesma, quanto os pertencentes à **classe pai**:

```
const iAccount = new InvestmentAccount(1, "Lucas", 0.02);

iAccount.deposit(200);

console.log(iAccount.balance);
console.log(iAccount.tax);

console.log(iAccount.greeting());

// output expected:
// 200
// 0.02
// Bem vindo a sua conta, Lucas!
```

Pode-se dizer então, que a classe `InvestmentAccount` é uma **subclasse** da classe original, `Account`.

Polimorfismo

Agora, imagine que a conta de rendimentos deva ter uma “saudação” diferente da conta comum.

Naturalmente, um método chamado `greeting` deva ser adicionado à classe `InvestmentAccount`.

```

class InvestmentAccount extends Account {
    #tax;

    constructor(_number, _owner, _tax) {
        super(_number, _owner);
        this.#tax = _tax;
    }

    get tax() {
        return this.#tax;
    }

    greeting() {
        return "Bem vindo a sua conta de investimentos, " + this.#owner + "!";
    }

    profits() {
        this.#balance *= this.#tax;
    }
}

```

Instanciando os dois tipos de conta, observamos o seguinte comportamento:

```

const account = new Account(1, "Lucas");
const iAccount = new InvestmentAccount(1, "Lucas", 0.02);

console.log(account.greeting());
console.log(iAccount.greeting());

// output expected:
// Bem vindo a sua conta, Lucas!
// Bem vindo a sua conta de investimentos, Lucas!

```

Observe que a classe de rendimentos terá um novo comportamento para o método `greeting`, ou seja, o método `greeting` da classe original foi sobreescrito.

Esse comportamento, onde uma **subclasse** pode **sobrescrever** o comportamento de sua **classe pai**, é conhecido como **Polimorfismo**.

Programação Orientada a Objetos (POO)

Programação Orientada a Objetos (POO) é um **paradigma de programação** baseado no conceito de "**objetos**", que podem conter dados na forma de **campos**, também conhecidos como **atributos**, e códigos, na forma de **procedimentos**, também conhecidos como **métodos**.

Esse paradigma possui algumas **características** importantes:

- **Abstração:** capacidade de abstrair tudo o que não é essencial para um objetivo a ser seguido
- **Encapsulamento:** capacidade de tornar valores privados, para que usuários não consigam alterar seus valores diretamente
- **Herança:** princípio onde uma classe pode herdar atributos e métodos de outra classe
- **Polimorfismo:** capacidade de uma subclasse em poder sobrescrever o comportamento de sua classe pai, de acordo com a sua necessidade