



INSTITUTO DE GESTÃO E TECNOLOGIA
DA INFORMAÇÃO

JavaScript Avançado I

Bruno Augusto Teixeira

2022

JavaScript Avançado I

Bootcamp Desenvolvedor(a) React

Bruno Augusto Teixeira

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1.	Manipulação de Eventos.....	5
1.1.	Arquitetura de Eventos.....	9
	Event Bubbling.....	10
1.3.	Eventos da Interface UIEvent	17
1.4.	Eventos da Interface MouseEvent.....	19
1.5.	Eventos da Interface KeyboardEvent.....	22
1.6.	Eventos da Interface FocusEvent.....	23
Capítulo 2.	JavaScript: Funções.....	26
2.1.	Escopes.....	26
2.2.	Closures.....	30
2.3.	Prototypes.....	32
2.4.	IIFE – Immediately Invoked Function Expression	34
2.5.	Proxy	35
2.6.	Curry.....	36
Capítulo 3.	JavaScript Assíncrono.....	39
3.1.	Promises.....	39
3.2.	Promises API	44
Capítulo 4.	ECMAScript	47
5.1.	Contexto histórico.....	47
5.2.	ECMAScript 2015	48
5.3.	ES7.....	62

5.4. ES8.....	63
5.5. ES9.....	67
Referências.....	70

Capítulo 1. Manipulação de Eventos

O desenvolvimento de sistemas web, cada vez mais, vem demandando sistemas mais dinâmicos e interativos. A utilização de eventos nos sistemas é um recurso poderoso para torná-los mais interessantes e atrativos, embora não seja o requisito primordial para que o sistema web seja considerado de qualidade. Eventos permitem sinalizar algum acontecimento no sistema, como, por exemplo, um clique do usuário, uma tecla pressionada, conclusão do download de uma imagem etc.

Todos os nós da árvore DOM (não está limitado somente ao DOM) podem gerar esses sinais e sempre que um sinal é gerado, um evento é acionado. Caso a aplicação possua um código para escutar (*listen*) esse evento, ações serão executadas. Então, ao disparar um evento, é necessário que exista um escutador de eventos (*event listeners*) e é preciso definir um manipulador de eventos (*event handler*) a ser executado.

Manipuladores de evento

Existem diversas formas de adicionar um ouvinte de eventos às páginas Web, vejamos, a seguir, os exemplos.

1. Manipuladores de eventos in-line

Uma das formas mais antigas de atribuir manipuladores de eventos é direto no atributo HTML, conforme Figura 1.

Figura 1 – Exemplo 1 definição de evento.

```
<button onclick="function btAlert() { window.alert('Botão clicado!');}">Clique aqui</button>
```

Uma outra alternativa é separar o código JavaScript do elemento, e atribuir ao manipulador de eventos a função que deve ser invocada, conforme Figura 2.

Figura 2 – Exemplo 1 definição de evento.

```
<button onclick="btAlert()">Clique aqui</button>
<script>
function btAlert() {
    window.alert("Botão clicado!");
}
</script>
```

Os manipuladores de evento in-line, a princípio, são simples de serem utilizados, sobretudo, se estivermos desenvolvendo algo rápido. No entanto, a sua utilização não é considerada uma boa prática por diversos motivos; um deles é a legibilidade do código.

2. Propriedades do manipulador de eventos

Os elementos HTML possuem propriedades para vários manipuladores de eventos. No exemplo da Figura 3, obtemos a referência de um botão que foi definido na página por meio do método *Document.querySelector()*. Ao obtermos a referência deste button, recebemos um objeto que possui diversas propriedades de manipuladores de eventos nas quais podem ser disparadas. Para o exemplo, foi utilizada a propriedade *onclick* referente ao clique do mouse. Assim, adicionamos uma função anônima no manipulador de eventos para o evento *onclick*, que, por sua vez, irá alertar uma mensagem quando houver um clique no botão.

Figura 3 – Exemplo 1 definição de evento.

```
var btn = document.querySelector('button');

btn.onclick = function() {
    window.alert("Botão clicado!");
}
```

Podemos, também, definir a propriedade do manipulador de eventos para ser igual a um nome de função, conforme Figura 4.

Figura 4 – Exemplo 1 definição de evento.

```
var btn = document.querySelector('button');

function btAlert() {
    window.alert("Botão clicado!");
}
btn.onclick = btAlert;
```

3. `addEventListener()` e `removeEventListener()`

Este mecanismo é mais novo para atribuição nos manipuladores de evento, foi definido a partir do DOM Nível 2 e forneceu aos navegadores as funções de `addEventListener()` e `removeEventListener`. O `addEventListener()` não é suportado no Internet Explorer 8 e versões anteriores, e no Opera 6.0 e versões anteriores. No entanto, para estas versões específicas de navegadores, é possível utilizar o `attachEvent()` para anexar manipuladores de eventos.

O exemplo das Figuras acima pode ser feito da seguinte forma:

Figura 5 – Exemplo 1 definição de evento.

```
var btn = document.querySelector('button');

function btAlert() {
    window.alert("Botão clicado!");
}

btn.addEventListener('click', btAlert);
```

A sintaxe para o `addEventListener` é `alvo.addEventListener(_type, _function, _useCapture);`, na qual **`_type`** representa o nome do evento, **`_function`** a função que será disparada quando o evento ocorrer e **`_useCapture`** um booleano para especificar se o evento deve ser executado na captura ou na fase borbulhante (será estudado na seção 2.4). Esse mecanismo é mais robusto e tem algumas vantagens sobre os outros mecanismos. A primeira é a função de `removeEventListener()`, que possibilita remover um listener adicionado anteriormente.

A segunda vantagem é a possibilidade de adicionar vários manipuladores para a um mesmo evento. Sendo assim, adotaremos esta sintaxe para os exemplos doravante.

Argumentos de Evento

Para uma maior robustez aos manipuladores de eventos, as funções fornecem um parâmetro que, geralmente, é especificado com o nome **event**, **evt**, ou simplesmente **e**. Este parâmetro é chamado de objeto de evento (*event object*) e é passado para os manipuladores de eventos para fornecer recursos e informações extras referentes à interface do objeto de evento. Um evento disparado por um clique do mouse terá propriedades diferentes no seu objeto de evento, quando comparado a um evento disparado por uma tecla do teclado, por exemplo. A maioria dos eventos terão seus próprios comportamentos especializados, mas existem algumas propriedades comuns a todos os eventos referente à interface *Event*. As mais populares dessas propriedades comuns são:

- **type**. O tipo de evento que se está lidando;
- **target**. O elemento que disparou o evento.

No exemplo da Figura 6, a função do manipulador de evento apresenta o tipo de evento que foi disparado e o componente que disparou, esse recurso é muito útil quando se precisa atribuir o mesmo manipulador de eventos a vários elementos, sem precisar selecioná-los *a priori*.

Figura 6 – Exemplo 1 definição de evento.

```
var btn = document.querySelector('button');

function btAlert(e) {
    window.alert("O evento: " + e.type + " foi disparado pelo: " + e.target.id);
}

btn.addEventListener('click', btAlert, false);
```

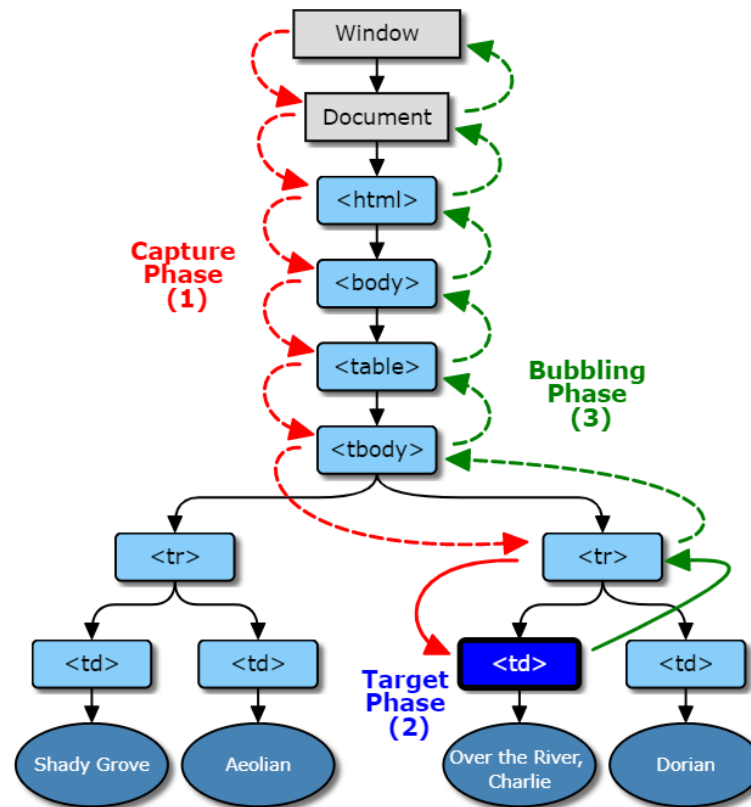

As próximas seções deste capítulo apresentarão a arquitetura dos eventos, e uma seção para cada interface de evento mais utilizadas.

1.1. Arquitetura de Eventos

O fluxo de eventos possui três fases *Capturing Phase*, *Target Phase* e *Bubbling Phase*, conforme apresentado no diagrama da Figura 7. Os objetos de eventos irão propagar na árvore de eventos do DOM para um destino de acordo com o mecanismo que fora atribuído.

1. **Capturing Phase** – Fase de captura, a ação do evento ocorre inicialmente nos ancestrais do destino até o pai do destino.
2. **Target Phase** – Fase na qual o objeto de evento tem como destino.
3. **Bubbling Phase** – Fase de borbulhamento, a ação do evento ocorre inicialmente no elemento que disparou o evento, depois em seu pai, e depois em todos os outros ancestrais.

Figura 7 – Diagrama representativo do Fluxo de Eventos.



Fonte: World Wide Web Consortium, (MIT, ERCIM, Keio, Beihang).

Event Bubbling

O *Event Bubbling*, em tradução livre “borbulhamento de evento”, ocorre dentro do processo chamado propagação de eventos. A propagação de eventos é um mecanismo que define como os eventos se propagam ou caminham pela árvore DOM para chegar ao seu destino (*target*) e o que acontece depois. O fluxo de propagação de eventos é disparado assim que ocorre um evento e, a partir disso, a função correspondente ao evento é então acionada em cada parte de cada estágio de cada fluxo. Embora existam três fases no fluxo de eventos nas versões mais recentes dos navegadores, a fase *target*, que ocorre quando o evento chega ao elemento de destino

que gerou o evento, não é tratada separadamente; os eventos registrados para ambas as fases de *capturing* e *bubbling* são executados nessa fase e, assim, o fluxo de eventos, neste caso, é definido somente em duas fases: *capturing* e *bubbling*.

Bubbling

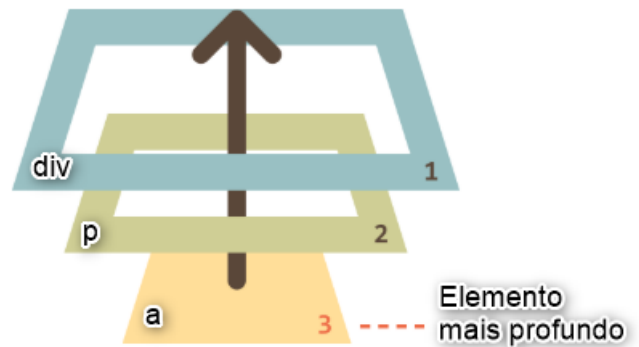
O princípio da fase de borbulhamento é simples. Quando um evento acontece num determinado elemento, primeiramente, se executa a ação correspondente a este elemento, em seguida, é executada a ação do elemento pai e assim por diante nos seus ancestrais até chegar no último nó da árvore DOM. O exemplo da Figura 8 define um hyperlink, elemento `<a>`, com um evento de *onclick* aninhado ao elemento `<p>` (nó pai) que também foi definido um evento de *onclick*, e em seu ancestral `<div>` idem.

Figura 8 – Exemplo de borbulhamento de evento.

```
<html lang="pt">
<head>
  <title>JavaScript Borbulhamento</title>
  <style type="text/css">
    div, p, a{
      padding: 15px 30px;
      display: block;
      border: 2px solid #000;
      background: #fff;
    }
  </style>
</head>
<body>
  <div onclick="alert('Borbulhamento: ' + this.tagName)">DIV
    <p onclick="alert('Borbulhamento: ' + this.tagName)">P
      <a href="#" onclick="alert('Borbulhamento: ' + this.tagName)">A</a>
    </p>
  </div>
</body>
</html>
```

Quando o usuário clicar no hyperlink, esse evento de clique passará pelo elemento `<p>` que contém o link, o elemento `<div>`, ..., o elemento `<html>` e, por fim, no nó do documento. O elemento `<a>` é o elemento *target*, que por padrão será executada a ação deste elemento. A Figura 9 ilustra como o evento borbulha do fundo para o topo.

Figura 9 – Exemplo de borbulhamento de evento.



Fonte: Adaptado de Google.

Capturing

A fase de captura (*capturing*) é o oposto da fase de borbulhamento, o evento propaga na árvore DOM a partir do objeto *Window* até o elemento alvo. No exemplo da Figura 10, definimos um hyperlink, elemento `<a>`, aninhado ao elemento `<p>` (nó pai) que, por sua vez, está aninhado ao elemento `<div>`. No JavaScript, é atribuído um evento de click aos elementos da página

```
elem.addEventListener("click", showTagName, true);
```

e, nessa atribuição, foi definido que será um evento de captura

```
elem.addEventListener("click", showTagName, true);
```

Figura 10 – Exemplo de captura de evento.

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript Captura</title>
    <style type="text/css">
      div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
      }
    </style>
  </head>
  <body>
    <div id="wrap">DIV
      <p class="hint">P
        <a href="#">A</a>
      </p>
    </div>
    <script>
      function showTagName() {
        alert("Captura: " + this.tagName);
      }

      var elems = document.querySelectorAll("div, p, a");
      for(let elem of elems) {
        elem.addEventListener("click", showTagName, true);
      }
    </script>
  </body>
</html>
```

O evento de captura não é suportado em todos os navegadores e é raramente utilizado; versões do Internet Explorer anteriores à 9.0 não suportam este recurso. Apesar de não ser comumente utilizado, esse processo pode ser útil para dar aos elementos *parent* **autonomia** sobre os eventos que ocorrem nos seus descendentes. Isso permite desacoplar uma funcionalidade comum, genérica, que precisa ser implementada em um componente como um todo, das funcionalidades específicas dos seus subcomponentes.

Interromper propagação

O processo de borbulhamento ocorre por default nos navegadores, entretanto, é possível interromper o processo de propagação de eventos caso seja necessário evitar que os ancestrais sejam notificados sobre o evento. Por exemplo, suponha que você

tenha elementos aninhados e cada elemento tenha um manipulador de eventos onclick que exibe uma caixa de diálogo de alerta. Normalmente, quando você clica no elemento interno, todos os manipuladores serão executados de uma só vez, desde que o evento borbulhe até a árvore DOM. O exemplo da Figura 11 adiciona os eventos de click em cada elemento da página, entretanto, a ação do evento interrompe sua propagação com a chamada do método `event.stopPropagation();`.

Figura 11 – Exemplo de captura de evento.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Interromper Propagacao de Eventos</title>
    <style type="text/css">
      div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
      }
    </style>
  </head>
  <body>
    <div id="wrap">DIV
      <p class="hint">P
        <a href="#">A</a>
      </p>
    </div>
    <script>
      function showAlert(event) {
        alert("Item clicado: " + this.tagName);
        event.stopPropagation();
      }
      var elems = document.querySelectorAll("div, p, a");
      for(let elem of elems) {
        elem.addEventListener("click", showAlert);
      }
    </script>
  </body>
</html>
```

O método `stopPropagation()` interrompe a notificação do evento aos ancestrais do elemento, entretanto, caso o elemento possua várias ações para o mesmo evento, todos eles são executados. É possível interromper a propagação de outras ações do mesmo evento, que foram atribuídas no elemento por meio do comando `event.stopImmediatePropagation();`, conforme exemplificado na Figura 12.

Figura 12 – Exemplo de captura de evento.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Interromper imediatamente a propagacao</title>
    <style type="text/css">
      div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
      }
    </style>
  </head>
  <body>
    <div onclick="alert('Clicado: ' + this.tagName)">DIV
      <p onclick="alert('Clicado: ' + this.tagName)">P
        <a href="#" id="link">A</a>
      </p>
    </div>

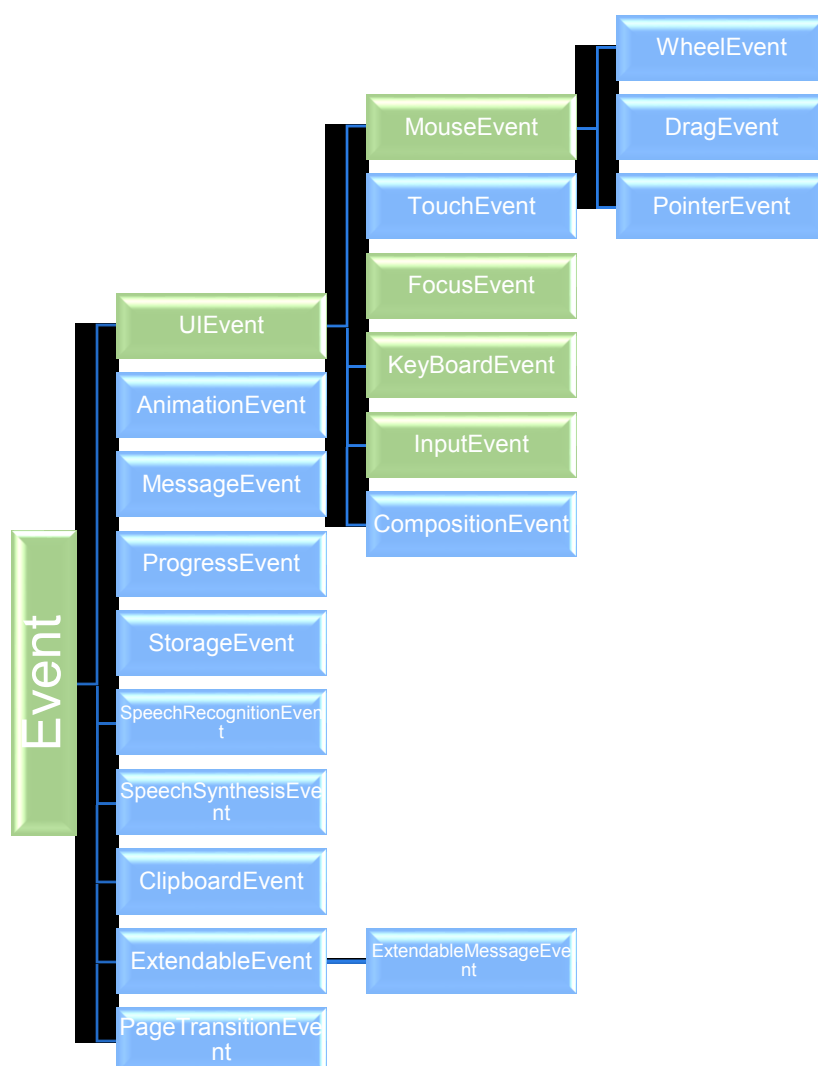
    <script>
      function alertaA() {
        alert("Alerta A!");
        event.stopImmediatePropagation();
      }
      function alertaB() {
        alert("Alerta B!");
      }

      var link = document.getElementById("link");
      link.addEventListener("click", alertaA);
      link.addEventListener("click", alertaB);
    </script>
  </body>
</html>
```

Conforme mencionamos, os eventos são disparados para notificar algum acontecimento na árvore DOM. Esses eventos são representados por um objeto (*object event*) que é baseado na interface *Event*. Essa interface possui suas propriedades e métodos específicos, porém, os eventos podem ter propriedades ou métodos

customizados de acordo com cada interface para se obter informações adicionais sobre o que aconteceu. A árvore abaixo representa as principais interfaces que herdam da interface *Event*, mas existem diversas interfaces de eventos específicas para cada navegador. Nesta apostila, serão apresentadas apenas as interfaces especificadas pelo DOM, conforme destacado pela cor verde na Figura 13.

Figura 13 – Árvore de eventos da interface *Event*.



1.3. Eventos da Interface UIEvent

São eventos da interface que estão associados à interface de usuário, tais como *load*, *unload*, *abort*, *error*, *select*, *resize*.

Propriedades da interface

view – identifica a janela que gerou o evento.

detail – traz algum detalhe adicional dependendo do tipo de evento.

Eventos da Interface

select – O evento é disparado depois que algum texto foi selecionado em um elemento.

Sintaxe	<code>object.addEventListener("select", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	<code><input type="file"></code> , <code><input type = "password"></code> , <code><input type="text"></code> e <code><textarea></code>

error – O evento é disparado se ocorrer um erro ao carregar algum elemento.

Sintaxe	<code>object.addEventListener("error", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	<code></code> , <code><input type="image"></code> , <code><object></code> , <code><link></code> e <code><script></code>

abort – O evento é disparado quando o carregamento de um áudio / vídeo é abortado pelo usuário e não ocorreu erro.

Sintaxe	<code>object.addEventListener("abort", myScript);</code>
Borbulhamento	Não

Cancelável	Não
HTML tags suportadas	<audio> e <video>

unload – O evento é disparado quando a janela do navegador foi fechada ou a página foi recarregada.

Sintaxe	object.addEventListener("unload", myScript);
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	<body>

load – O evento é disparado quando um objeto foi carregado. Sua utilização pode ser feita para verificar o tipo e a versão do navegador, para que a página seja adequada à versão do usuário. Uma outra utilização é para o tratamento de cookies.

Sintaxe	object.addEventListener("load", myScript);
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	<body>, <frame>, <iframe>, , <input type="image">, <link>, <script> e <style>

resize – O evento é disparado quando a janela do navegador é redimensionada.

Sintaxe	object.addEventListener("resize", myScript);
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	<body>

1.4. Eventos da Interface MouseEvent

Eventos desta interface estão especificamente associados para o uso com dispositivos de entrada de ponteiro, como um mouse ou um trackball. Os eventos desta categoria são **click**, **dblclick**, **mousedown**, **mouseenter**, **mouseleave**, **mousemove**, **mouseout**, **mouseover**, **mouseup** e **contextmenu**. Os eventos **mouseover/mouseout** e **mouseenter/mouseleave** possuem o mesmo comportamento, entretanto, os eventos **mouseenter/mouseleave** não consideram as transições entre os elementos aninhados.

Propriedades da interface

screenX, *screenY*, *clientX*, *clientY*, *altKey*, *ctrlKey*, *shiftKey*, *metaKey*, *button*, *buttons*, *relatedTargets*

Ordem dos eventos

mousedown -> mouseup -> click

Eventos da Interface

click – O evento é disparado quando o usuário clica em um elemento.

Sintaxe	<code>object.addEventListener("click", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code>

dblclick – O evento é disparado quando o usuário dá um duplo click em um elemento.

Sintaxe	<code>object.addEventListener("dblclick", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim

HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>.
-----------------------------	--

mousedown – O evento é disparado quando um usuário pressiona um botão do mouse sobre um elemento.

Sintaxe	<code>object.addEventListener("mousedown", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>.

mouseenter – O evento é disparado quando o ponteiro do mouse é movido para um elemento. Este evento é frequentemente utilizado junto com o evento **mouseleave**, que ocorre quando o ponteiro do mouse é movido para fora de um elemento.

Sintaxe	<code>object.addEventListener("mouseenter", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>.

mouseleave – O evento é disparado quando o ponteiro do mouse é movido para fora de um elemento. Este evento é frequentemente utilizado junto com o evento **mouseenter**, que ocorre quando o ponteiro do mouse é movido para dentro de um elemento.

Sintaxe	<code>object.addEventListener("mouseleave", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>.

mousemove – O evento é disparado quando o ponteiro está se movendo enquanto está sobre um elemento.

Sintaxe	<code>object.addEventListener("mousemove", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code> .

mouseout – O evento é disparado quando o ponteiro do mouse é movido para fora de um elemento ou de um de seus filhos. Este evento é frequentemente utilizado junto com o evento **mouseover**, que ocorre quando o ponteiro é movido para um elemento ou para um de seus filhos.

Sintaxe	<code>object.addEventListener("mouseout", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code> .

mouseover – O evento é disparado quando o ponteiro é movido para um elemento ou para um de seus filhos. Este evento é frequentemente utilizado junto com o evento **mouseout**, que ocorre quando o ponteiro do mouse é movido para fora de um elemento ou de um de seus filhos.

Sintaxe	<code>object.addEventListener("mouseover", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code> .

mouseup – O evento é disparado quando um usuário libera um botão do mouse sobre um elemento.

Sintaxe	<code>object.addEventListener("mouseup", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code> .

contextmenu – O evento é disparado quando um usuário clica com um botão direito do mouse para abrir um menu de contexto.

Sintaxe	<code>object.addEventListener("contextmenu", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML

1.5. Eventos da Interface *KeyboardEvent*

Eventos da interface *KeyboardEvent* se referem aos eventos do teclado e possuem os seguintes eventos: **keydown** e **keyup**. Vale ressaltar que desta interface dependem do dispositivo de entrada e como está configurado no sistema operacional. Esses eventos, geralmente, são direcionados ao elemento que tem o foco.

Ordem dos Eventos de Teclado

Os eventos desta interface ocorrem **sempre** em uma ordem definida em relação à outra **keydown**, **beforeinput**, **input** e **keyup**.

Propriedades da Interface

key, *code*, *location*, *altKey*, *shiftKey*, *ctrlKey*, *metaKey*, *repeat*, *isComposing*

Eventos da Interface

keydown – O evento é disparado quando uma tecla é pressionada no teclado.

Sintaxe	<code>object.addEventListener("keydown", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code> .

keyup – O evento é disparado quando uma tecla do teclado que foi pressionada é liberada.

Sintaxe	<code>object.addEventListener("keyup", myScript);</code>
Borbulhamento	Sim
Cancelável	Sim
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code> .

1.6. Eventos da Interface FocusEvent

Eventos que estão associados ao foco, tais como **focusin**, **focusout**, **blur**, **focus**.

Propriedades da interface

relatedTarget – identifica um secundário target dependendo do tipo de evento.

Ordem dos Eventos de Foco

Os eventos desta interface ocorrem em uma ordem definida em relação à outra. A sequência típica de eventos, quando um foco é deslocado entre elementos (esta ordem pressupõe que nenhum elemento está inicialmente focado), é definida desta forma:

Usuário muda o foco

- 1- **focusin** - é disparado antes que o elemento alvo receba o foco.
- 2- **focus** - é disparado depois que o elemento alvo recebe o foco.
Usuário muda o foco
- 3- **focusout** - é disparado antes do primeiro elemento perder o foco.
- 4- **focusin** - é disparado antes do segundo elemento receber o foco.
- 5- **blur** - é disparado depois do primeiro elemento perder o foco.
- 6- **focus** - é disparado antes do segundo elemento receber o foco.

Eventos da Interface

focusin - O evento é disparado quando o elemento está prestes a obter o foco.

Sintaxe	<code>object.addEventListener("focusin", myScript);</code>
Borbulhamento	Sim
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>

focusout - O evento é disparado quando o elemento está prestes a perder o foco.

Sintaxe	<code>object.addEventListener("focusout", myScript);</code>
Borbulhamento	Sim
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>

blur - O evento é disparado quando o elemento perde o foco.

Sintaxe	<code>object.addEventListener("blur", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <base>, <bdo>, , <head>, <html>, <iframe>, <meta>, <param>, <script>, <style> e <title>

focus – O evento é disparado quando o elemento recebe o foco.

Sintaxe	<code>object.addEventListener("focus", myScript);</code>
Borbulhamento	Não
Cancelável	Não
HTML tags suportadas	Todos os elementos HTML, com exceção: <code><base></code> , <code><bdo></code> , <code>
</code> , <code><head></code> , <code><html></code> , <code><iframe></code> , <code><meta></code> , <code><param></code> , <code><script></code> , <code><style></code> e <code><title></code>

Capítulo 2. JavaScript: Funções

2.1. Escopes

Escopos definem a acessibilidade e o ciclo de vida de variáveis na aplicação. A acessibilidade determina as partes da aplicação nas quais tais variáveis podem ser acessadas, ao passo que o ciclo de vida é o período de existência das variáveis durante a execução da aplicação. Em JavaScript, existem dois tipos de escopo, Escopo Global e Escopo Local.

Escopo Global

As variáveis declaradas fora de qualquer função automaticamente são definidas como variáveis globais. Essas variáveis podem ser modificadas por qualquer função na aplicação. Vejamos, na Figura 14, os exemplos de declaração de variáveis globais.

Figura 14 – Exemplos de declaração de variáveis global.

```
1 //Exemplos de declaracao de variavel global
2 var sModelo = "Fusca";
3
4 sMarca = "Volkswagen";
5
6 var iAno;
7 iAno;
```

Uma variável que não foi declarada e é inicializada dentro de funções também é considerada variável global, conforme exemplo da Figura 15.

Figura 15 – Exemplos de declaração de variáveis global.

```
function printCor () {
    //cor sera considerada uma variavel global porque não foi
    //declarada com a palavra reservada "var" dentro da funcao
    sCor = "Preto";
    console.log(sCor);
}

printCor (); //Preto

//cor esta no escopo global, portanto podemos acessa-la
console.log(sCor); //Preto
```

O exemplo acima demonstra a atribuição da variável “sCor” dentro da função “printCor()”, assim, como a variável “sCor” não foi declarada utilizando com “var”, esta se torna global e pode ser acessada pelo comando da última linha “console.log(sCor);”. É importante destacar que a variável “sCor” só entrará no escopo global após a chamada da função “printCor()”, portanto, caso esta função não tivesse sido chamada, o último comando “console.log(sCor);” retornaria *undefined*.

Uma boa prática no desenvolvimento de JavaScript é a não poluição do escopo global, uma vez que todas as variáveis globais são vinculadas ao objeto de escopo global *window*. No exemplo da Figura 16, as variáveis declaradas podem ser acessadas conforme abaixo:

Figura 16 – Exemplo de acesso às variáveis globais.

```
10 console.log(window.sModelo); // Fusca
11
12 console.log("sMarca" in window) // true
13
14 console.log("iAno" in window) //true
```

Escopo Local

Variáveis de escopo local ou variáveis com escopo a nível de função são todas as variáveis que são declaradas dentro das funções. Essas variáveis são acessíveis somente na função na qual foi declarada ou pelas funções que foram declaradas dentro desta função. O exemplo da Figura 17 demonstra a criação de uma variável local “sProprietario” na função “criarCarro”, que só é acessível dentro desta função. A função “printProprietario” utilizou a variável local criada na função “criarCarro”, entretanto, ao executar esta função, será gerada uma exceção, pois a variável tem escopo local e não é acessível por outras funções.

Figura 17 – Exemplo de declaração de variável local.

```
function criarCarro () {  
    var sProprietario = "Jose";  
}  
  
function printProprietario () {  
    console.log(sProprietario);  
}  
  
criarCarro();  
printProprietario();
```


As variáveis locais sempre possuem prioridade com relação às variáveis globais, assim, se uma variável local for declarada dentro de uma função com o mesmo nome de uma variável global já declarada, o valor atribuído à variável local é considerado.

Figura 18 – Exemplo de prioridade de variável local sobre global.

```
var sCor = "Prata";  
  
function printCor () {  
    var sCor = "Preto";  
    console.log(sCor);  
}  
  
printCor (); //Preto
```

No JavaScript, há um conceito chamado **Hoisting**, que permite a utilização de variáveis e funções antes de suas declarações, diferentemente de outras linguagens nas quais todas as variáveis e funções devem ser declaradas antes de sua utilização. Essa possibilidade é permitida pelo fato do compilador do JavaScript mover todas as declarações de variáveis e funções para o topo da aplicação, esse movimento é chamado de Hoisting. O Hoisting é aplicado apenas para as declarações de variáveis, portanto, caso a variável seja inicializada, ela não é movida.

Figura 19 – Exemplo Hoisting de variável.




```
//Declaracao movida para o topo (Hoisting)
iAno = 2009;

function printAno () {
    console.log("Ano: " + iAno);
}

var iAno;
```

O Hoisting também é aplicado para a declaração de funções, conforme exemplo abaixo:

Figura 20 – Exemplo Hoisting de função.



```
//Declaracao movida para o topo (Hoisting)
printAno();

function printAno () {
    var iAno = 2009;
    console.log("Ano: " + iAno);
}
```

Assim como o Hoisting não é aplicado para variáveis inicializadas, nas funções em expressão também não é aplicado.

Figura 21 – Exemplo restrição de Hoisting de função.

```
printAno(); //Exception

var sAno = function printAno () {
    var iAno = 2009;
    console.log("Ano: " + iAno);
}
```

2.2. Closures

As closures (fechamentos) permitem que uma função aninhada acesse qualquer variável fora do seu escopo, ou seja, as funções que foram definidas dentro de um contexto léxico acessam as variáveis deste mesmo contexto. Assim, as closures possuem três escopos:

- Acesso às variáveis definidas dentro do seu escopo.
- Acesso às variáveis definidas dentro do seu escopo léxico.
- Acesso às variáveis globais.

A imagem abaixo apresenta um exemplo simples do funcionamento da closure:

Figura 22 – Exemplo de closure.

```
function IMC() {  
  var altura = 1.80;  
  function calcula() {  
    var peso = 70;  
    console.log("IMC: " + peso/(altura*altura));  
  }  
  return calcula;  
}  
  
var imc = IMC();  
imc();
```

No exemplo acima, podemos observar que temos duas funções:

- Uma função externa **IMC()** que tem a variável **altura** e retorna a função **calcula**;
- Uma função interna **calcula()** que tem a variável **peso** em seu escopo e acessa a variável **altura** do escopo externo à sua declaração.

Ao atribuir a função **IMC()** à variável **imc**, o compilador do JavaScript retorna o corpo da função **calcula** e encerra o escopo da variável **altura**, assim **imc** possuiria o seguinte valor:

Figura 23 – Saída da função com clousure.

```
function calcula() {
    var peso = 70;
    console.log("IMC: " + peso/(altura*altura));
}
```

Como o compilador do JavaScript lida com o acesso à variável **altura** caso seja executada a função atribuída à variável **imc**? **Closure!!**

Em outras palavras, a função interna **calcula()** preserva a cadeia de escopo da função que a declarou no momento em que ela for executada, assim, a função **calcula()** pode acessar as variáveis da função que a declarou.

Clousures possuem diversas aplicações práticas, uma delas é a simulação de variáveis ou funções privadas (encapsulamento), o que não é suportado pela linguagem JavaScript.

Figura 24 – Simulação de visibilidade com clousures.

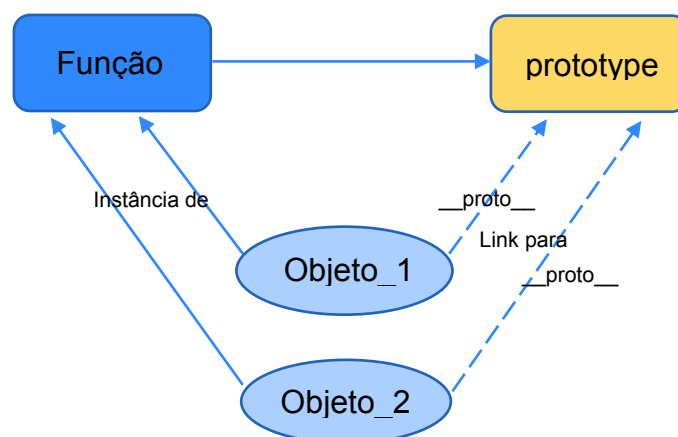
```
function Carro() {
    this.proprietario = "Marcos"
    var ano = 2019;
    this.getAno = function() {
        return ano;
    }
    this.setAno = function(a) {
        ano = a;
    }
}
var carro = new Carro();
console.log(carro.proprietario); // Acessível
console.log(carro.ano); // undefined
console.log(carro.getAno()); //2019
```

Um outro exemplo da utilização de clousures é em **Accumulator Generator**, em outras palavras, podemos dizer que é uma técnica que pode ser utilizada para a contabilização de eventos.

2.3. Prototypes

Protótipo é um objeto no qual toda função criada recebe automaticamente a propriedade *prototype* que, por sua vez, aponta para um novo objeto em branco. Esse objeto é quase idêntico a um objeto criado a partir de um objeto literal ou pelo construtor `Object()`, com a exceção de que sua propriedade `constructor` aponta para a função que fora utilizada, e não para o objeto embutido `Object()`. A partir do protótipo, é possível adicionar membros a esse objeto em branco e permitir que outros objetos herdem os novos membros. Os protótipos, quando bem utilizados, podem reduzir o tempo de inicialização dos objetos e o consumo de memória da aplicação (CROCKFORD).

Figura 25 – Diagrama de prototype em JavaScript.



O exemplo acima ilustra que cada objeto criado pelo literal **new** possui uma propriedade `__proto__` que aponta para o prototype da função utilizada para criar o

objeto, portanto, os objetos não possuem a propriedade **prototype**, apenas a propriedade **__proto__**, conforme exemplo abaixo:

Figura 26 – Exemplo de visualização ao objeto prototype.

```
var carro = function(){
}
console.log(carro.prototype); // [object Object] { ... }
var carro = {sProprietario: 'Maria'}
// __proto__ está obsoleto apenas didático não utilize em aplicações
console.log(carro.__proto__); // [object Object] { ... }
```

Para as classes acessarem o prototype, utiliza-se a sintaxe **<função>.prototype**, que possibilita a modificação e/ou inclusão de novos membros no protótipo no qual as instâncias da classe apontam. Vejamos, na imagem abaixo, a utilização do prototype para a inclusão de uma nova função no objeto **carro**. Quando é feita a inclusão direta na função sem o uso do **prototype (1)**, podemos observar que somente o objeto Carro possui a nova função, na saída do console em **(2)**, o objeto possui a função, enquanto que, na saída **(3)**, a nova função não foi herdada, portanto, haverá uma exceção na aplicação. Com a utilização do **prototype (4)**, todas as instâncias da classe recebem a nova função ou atributo conforme saída **(5)**.

Figura 27 – Exemplo de utilização do prototype.

```
var Carro = function (sProprietario){
    this.sProprietario = sProprietario;
}

Carro.imprimeDono = function () { console.log ("Dono: " + this.sProprietario); }; 1

Carro.prototype.imprimeProprietario = function () { console.log ("Proprietario: " + this.sProprietario); }; 4

var carro = new Carro("Maria");

Carro.imprimeDono(); 2 // "Dono: undefined"
carro.imprimeDono(); // Erro: imprimeDono não está definido 3
carro.imprimeProprietario(); 5 // "Proprietario: Fulano"
```

O **prototype** possui duas aplicações práticas, a primeira é a adição e busca por propriedades e métodos num objeto e a segunda finalidade é para implementação de herança em JavaScript.

2.4. IIFE – Immediately Invoked Function Expression

As funções no JavaScript podem ser declaradas de duas formas:

Function Declaration

Padrão para declaração de funções utilizando a palavra reservada **function**, seguido pelo nome da função (ex: “myFunction”), os () com os respectivos parâmetros e {} com o corpo da função.

```
function myFunction () {  
    /* código */  
}
```

Function Expression

Declaração com a expressão para atribuição de uma função a uma variável, de modo que o nome da função é a variável.

```
let myFunction = function() {  
    /* código */  
};
```

O JavaScript permite que tais funções possam ser imediatamente disparadas, conceito definido como IIFE ou *Immediately Invoked Function Expression*. A sintaxe para a chamada imediata dessas funções é definida como:

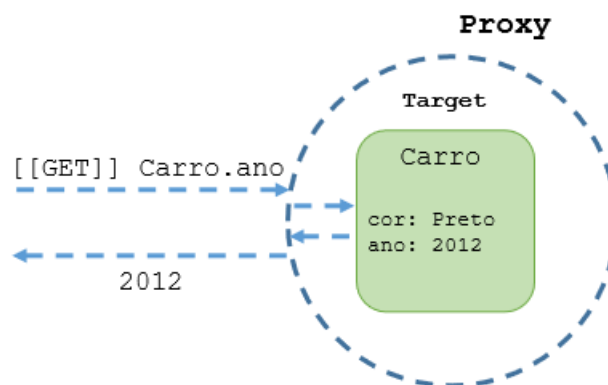
```
( function () {} ) ();
```

O principal objetivo para a invocação imediata de uma função é a privacidade, uma vez que toda função é definida no escopo global da aplicação, sendo possível o acesso ao código da função diretamente pela variável de escopo global.

2.5. Proxy

O Proxy foi uma das funcionalidades adicionada ao ES6. O Proxy permite que operações a serem realizadas num objeto possam ser interceptadas, conforme ilustrado na Figura 28, onde é feita uma requisição para obter a propriedade ano do objeto Carro, o objeto Proxy intercepta essa requisição com a possibilidade de executar operações nesta requisição.

Figura 28 – Representação de um Proxy.



O exemplo da Figura 29 apresenta a criação de um objeto Carro:

Figura 29 – Criação de um objeto.

```
let Carro = {
  proprietario: "Fernanda",
  ano: 2016
};
```

O objeto Proxy é criado com dois parâmetros, o objeto alvo, **target**, e o objeto **handler**, conforme Figura 30.

Figura 30 – Criação do Proxy para o objeto Carro.

```
let carroProxy = new Proxy(Carro, handler);
```

O objeto handler pode possuir vários interceptadores, denominados como **traps**. No exemplo da Figura 31, foi criado o objeto handler com um trap para interceptar as operações de get de propriedades do objeto alvo, de modo que seja validado se a propriedade existe e, caso contrário, uma mensagem padrão é retornada.

Figura 31 – Definição dos traps no objeto handler.

```
const handler = {
  get(target, property, receiver) {
    console.log(`GET ${property}`);
    if(property in target) {
      return target[property];
    }
    return "Propriedade inexistente"
  }
}
```

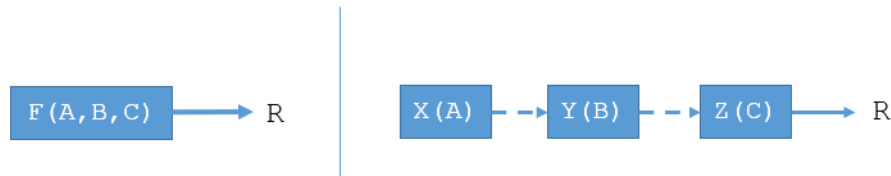
Os traps não estão limitados somente às operações de GET. Existem outros diferentes traps que podem ser declarados para interceptar operações no objeto, sendo eles: set, has, apply, construct, ownKeys, deleteProperty, defineProperty, isExtensible, preventExtensions, getPrototypeOf, setPrototypeOf e getOwnPropertyDescriptor.

2.6. Curry

Currying é definido como um processo para transformar uma função que espera vários parâmetros em uma função que espera um único parâmetro e retorna outra função curried. Conforme a Figura 32, uma função F(A, B, C) que recebe três parâmetros, ao sofrer currying, resulta em uma função que recebe um parâmetro e

retorna uma função que recebe um parâmetro, que, por sua vez, retorna uma função que recebe um parâmetro e retorna o resultado da função original.

Figura 32 – Processo de Proxy de uma função.



Na Figura 33, temos o exemplo de uma função que recebe três parâmetros e retorna um `console.log()`.

Figura 33 – Exemplo de uma função.

```
function log(date, type, message) {
  console.log(`[${date.getHours()}:${date.getMinutes()}] [${type}] ${message}`);
}
```

Para realizar o currying desta função, transformamos em uma função que recebe somente um parâmetro (`date`) e retorna uma função curried que recebe somente o segundo parâmetro (`type`), que, por sua vez, retorna uma função curried que recebe o terceiro parâmetro (`message`) e retorna o valor da função original, `console.log()`, conforme Figura 34.

Figura 34 – Exemplo de uma função.

```
const logCurrying =
  date =>
    type =>
      message =>
        console.log(`[${date.getHours()}:${date.getMinutes()}] [${type}] ${message}`);
```

O processo de currying é muito aplicado em casos que o mesmo parâmetro é informado para a função, neste caso, são criadas funções *curried* para reaproveitamento de código. O exemplo da Figura 35 cria uma nova função a partir da função acima que foi curried. Desta forma, tem uma nova função *logNow* que, por padrão, já recebe o primeiro parâmetro.

Figura 35 – Exemplo de uma função.

```
let logNow = logCurrying(new Date());  
logNow("DEBUG")("Exemplo de currying com parametro fixo");
```

Capítulo 3. JavaScript Assíncrono

3.1. Promises

Promises (promessas) são objetos, disponibilizados a partir da ES6, que possibilitam o desenvolvimento assíncrono no JavaScript. Esse desenvolvimento é simplificado, pois com a *promises* é possível adicionar callbacks (sucesso ou falha) diretamente ao objeto ao invés da tradicional passagem de callbacks para uma função de evento. Embora as *promises* possibilitem o desenvolvimento de métodos assíncronos, durante sua execução, seu comportamento é como um método síncrono, o valor final não é retornado, o método retorna uma promessa de fornecer o valor em algum momento no futuro (método assíncrono).

Vejamos um exemplo abaixo obtido do MDN como passados os callbacks para as funções, sem a utilização de *promises*:

Figura 36 – Exemplo de callbacks para eventos.

```
function sucessoCallback(result) {  
  console.log("Sucesso.... " + result);  
}  
  
function falhaCallback(error) {  
  console.log("Falhou " + error);  
}  
  
acelerar(sucessoCallback, falhaCallback);
```

Fonte: Adaptado de MDN.

A partir das *promises*, é possível adicionar callbacks de modo simplificado às funções:

Figura 37 – Exemplo de callbacks para eventos.

```
//Exemplo 1 de passagem dos callbacks
const promise = acelerar();
promise.then(sucessoCallback, falhaCallback);

//Exemplo 2 de passagem dos callbacks
acelerar().then(sucessoCallback, falhaCallback);
```

Fonte: Adaptado de MDN.

As *promises* possuem três estados mutuamente exclusivos (pendente, realizada, rejeitada):

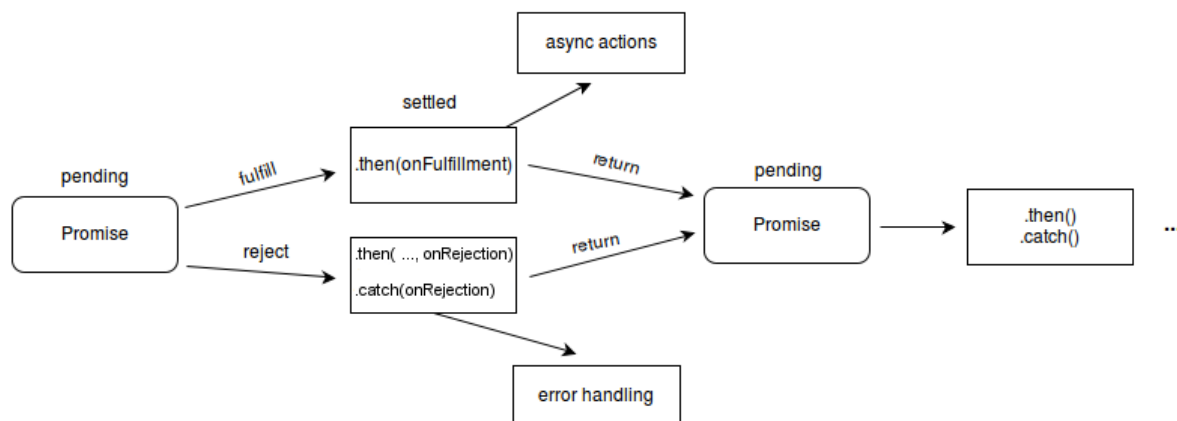
- **pending** (pendente) – estado inicial, a ação ainda não foi rejeitada ou realizada.
- **fulfilled** (realizada) – a ação relacionada à promessa que foi concluída.
- **rejected** (rejeitada) – a ação relacionada à promessa que foi rejeitada.

➔ Podemos dizer, também, que a *promise* está **settled** (estabelecida) se ela não está pendente, ou seja, se ela está realizada ou rejeitada. É importante destacar que *settled* não é um estado, apenas um modo conveniente de representar um dos dois estados.

As *promises* podem ser descritas por dois fatos que também são mutuamente exclusivos (resolvido ou não resolvido).

- **resolved** (Resolvida) – a promessa resolvida é quando tudo foi realizado dentro das suas ações ou a promessa está aguardando uma outra promessa (encadeamento de promessas).
- **unresolved** (Não Resolvida) - promessa que ainda não foi resolvida e poderá ser realizada ou rejeitada.

Figura 38 – Fluxograma dos estados das promessas.



Fonte: MDN.

O diagrama acima apresenta o fluxo dos estados das *promises* e seu estado inicial é definido como *pending*. Após a realização das suas ações, a promessa pode atingir o estado de *fulfill* ou *reject*; nesses estados, dizemos que a promessa está *settled* e já pode disparar as ações correspondentes ao sucesso ou rejeição com o valor retornado. O valor retornado pelos métodos *.then()* e *.catch()* pode ser *number*, *string*, ..., e até mesmo um objeto *promise*, com isso, é possível realizar o encadeamento de *promises*.

Conforme supracitado, as *promises* são objetos e objetos possuem propriedades e métodos, vejamos a seguir:

Propriedades	
Promise.length	Propriedade length sempre será 1 (número de argumentos do construtor).
Promise.prototype	Representa o <i>prototype</i> da função do construtor.

Métodos	
Promise.all([promise1, promise2, ...]);	Recebe um array de <i>promises</i> e retorna uma <i>promise fulfilled</i> quando todas as <i>promises</i> dos argumentos recebidos estiverem <i>fulfilled</i> ou

	retorna uma <i>promises rejected</i> assim que uma das <i>promises</i> estiverem no estado <i>rejected</i> .
Promise.race([promise1, promise2, ...]);	Recebe um array de <i>promises</i> e retorna uma <i>promise</i> no estado <i>fulfilled</i> ou <i>rejected</i> assim que uma das <i>promises</i> recebidas como parâmetro estiverem <i>fulfilled</i> ou <i>rejected</i> .
Promise.resolve(value);	Retorna um objeto <i>promise</i> que foi resolvido a partir de um valor.
Promise.reject(value);	Retorna um objeto <i>promise</i> que foi rejeitado por alguma razão.

Sintaxe da promise

A criação de uma *promise* é feita em duas etapas, a primeira cria o objeto *Promise* por meio de seu construtor com a palavra reservada *new* e define-se as condições de sucesso e de falha da promise. A segunda parte descreve o que a promessa deve executar em caso de sucesso com a função **resolve()** e o que fazer quando não tiver sucesso com a função **reject()**. Vejamos um exemplo:

Figura 39 – Exemplo de criação de promises.

```
//Primeira parte
//criar o Promise e definir as condições de sucesso e falha
let p = new Promise( (resolve, reject) => {
  let statusRequisicao = false;
  if (statusRequisicao)
    resolve('Sucesso!');
  if (!statusRequisicao)
    reject('Falha!');
})

//Segunda parte
//Define as acoes quando o status for realizado (fulfilled)
p.then( (mensagem) => {
  console.log(mensagem);
}).catch( (mensagem) => {
  console.log(mensagem);
})
```

No exemplo acima, foi criado um objeto *promise* **p** que sempre receberá como parâmetro uma função com dois parâmetros, **resolve** e **reject**. O **resolve** é chamado quando temos uma resposta de sucesso ativando o **.then**, e o **reject** é chamado quando temos um erro disparando o **.catch**. No caso, se o valor da variável **statusRequisicao** é falso, um erro é disparado (**reject**) e, caso seja verdadeiro, o valor é retornado pelo **resolve**. A chamada do *promise* é feita na segunda parte do código, caso o estado da *promise* seja realizada, a ação da função **.then** é executada e caso o estado seja rejeitado, a função no **.catch** é executada. A figura 40 apresenta uma outra forma de atribuir a ação referente ao **reject** da *promise* no próprio **then**, eliminando **.catch()**

Figura 40 – Exemplo de criação de promises.

```
//Alternativa para chamada do reject pelo then
p.then( (mensagem) => {
  console.log(mensagem);
}, (mensagem) => {
  console.log(mensagem);
})
```

Cadeia de Promises

Em algumas situações, necessitamos da execução de duas ou mais funções assíncronas consecutivamente, na qual cada função subsequente só começa quando a anterior é bem-sucedida (**resolved**), pois a próxima depende do valor da anterior. Essa situação pode ser denominada como **encadeamento de promises**.

No exemplo abaixo, criamos duas *promises* e a segunda *promise* **p_2** foi encadeada à execução da primeira *promise*, e quando a *promise* **p_1** estiver realizada, a **p_2** recebe o resultado da **p_1** e realiza suas ações.

Figura 41 – Exemplo de criação de promises.

```
let p_1 = () => { return new Promise( (resolve, reject) => {  
  let statusRequisicao = false;  
  if (statusRequisicao)  
    resolve('Sucesso!');  
  if (!statusRequisicao)  
    reject('Falha!');  
})  
};  
  
let p_2 = (mensagem) => {  
  return new Promise( (resolve, reject) => {  
    resolve('Sucesso na primeira promise');  
  });  
};  
  
p_1().then(p_2)  
  .then((mensagem) => {  
    console.log(mensagem);  
  }).catch( (mensagem) => {  
    console.log(mensagem);  
  });
```

3.2. Promises API

Promises Simultâneas (all)

Diferentemente da Cadeia de *promises*, podemos ter situações nas quais precisamos aguardar o resultado de múltiplas *promises*, entretanto, nenhuma depende da outra para executar. Para esses casos, podemos utilizar o método ***Promise.all*** para que um conjunto de promises executem simultaneamente e o retorno será somente após a realização de todas as promises, ou imediatamente, caso alguma delas seja rejeitada. No exemplo da Figura 42, foram criadas 3 promises (**p_1**, **p_2** e **p_3**), na **p_1** foi simulado um atraso de 5000 ms (5 segundos), enquanto que a **p_2** é realizada sem atraso e na **p_3** foi simulado um atraso de 1000ms. Ao executar simultaneamente esse conjunto de promises, nas quais nenhuma depende do resultado da outra, podemos observar que, embora a **p_2** seja realizada imediatamente, o resultado só será retornado após a execução da **p_1** e **p_2**. **A ordem do resultado será sempre a sequência do Array de promises ["Sucesso P1", "Sucesso P2", "Sucesso P3"]**.

Figura 42 – Exemplo de execução simultânea.

```
let p_1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    let statusRequisicao = true;
    if (statusRequisicao)
      resolve('Sucesso P_1');
    if (!statusRequisicao)
      reject('Falha P_1');}, 5000);
});

let p_2 = new Promise((resolve, reject) => {
  resolve('Sucesso P2');
});

let p_3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Sucesso P3");
  }, 1000);
});

Promise.all([p_1,p_2,p_3])
  .then(mensagem =>
    {console.log(mensagem);})
  .catch(mensagem =>
    {console.log(mensagem);});

//["Sucesso P1", "Sucesso P2", "Sucesso P_3"]
```

Corridas de Promises (race)

Enquanto no método **Promise.all** é necessário aguardar a realização de todas as promises, no método **Promise.race** o conjunto de promises executam simultaneamente e retorna imediatamente a promise que realizar ou rejeitar primeiro. No exemplo da Figura 43, a promise **p_2** é realizada mais rápido comparada às promises **p_1** e **p_3**.

Figura 43 – Exemplo de execução simultânea.

```
let p_1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    let statusRequisicao = true;
    if (statusRequisicao)
      resolve('Sucesso P_1');
    if (!statusRequisicao)
      reject('Falha P_1');}, 5000);
});

let p_2 = new Promise((resolve, reject) => {
  resolve('Sucesso P2');
});

let p_3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Sucesso P3");
  }, 1000);
});

Promise.race([p_3, p_2, p_1])
  .then(mensagem =>
    {console.log(mensagem);})
  .catch(mensagem =>
    {console.log(mensagem);});

//Sucesso P2
```

Capítulo 4. ECMAScript

O ECMAScript, também conhecido como JavaScript, é padronizado pela ECMA International na especificação 262 e, atualmente, encontra-se na sua 9ª versão. A seguir, serão apresentadas como a especificação da linguagem evoluiu ao longo deste tempo e quais foram as principais mudanças entre as suas versões.

5.1. Contexto histórico

A linguagem do JavaScript, criada em 1995, foi originalmente denominada desta forma com o intuito de capitalizar o sucesso do Java. A partir disto, a Netscape submeteu o JavaScript para a organização que padroniza informações, conhecida como ECMA International for Standardization. A organização, por sua vez, gerou um novo padrão de linguagem, conhecido como ECMAScript, ou seja, o ECMAScript é um padrão, enquanto o JavaScript é a implementação mais popular desse padrão. Esse padrão começou em 1997 com a sua primeira versão e, em 1998, foi lançada a sua segunda, terceira em 1999. Durante um longo período, a linguagem não recebeu novas padronizações e atualizações.

Em 2007, com a crescente vertente para a criação de aplicações mais dinâmicas e a popularização do AJAX, a linguagem necessitava de uma atualização para não cair em desuso. A partir disto, pequenas e grandes mudanças foram propostas na padronização da linguagem, tais como: módulos, novas sintaxes, classes, herança etc. Porém, membros do comitê consideraram que as mudanças eram muito grandes na versão e poderia comprometer a utilização da linguagem pela comunidade. Com esse dilema, a versão 4 nunca foi lançada e, com isso, criaram uma nova versão em 2009, conhecida como ECMAScript 2009, ou ES5, ou ECMAScript 5. Nesta versão, foi adicionado o “strict mode” com o objetivo de fornecer uma verificação de erros mais

completa e desenvolvimentos menos propensos a erros. Além disto, muitas ambiguidades nas especificações da 3ª edição foram resolvidas, recursos como getters e setters foram adicionados, suporte de biblioteca para JSON e reflexão mais completa sobre propriedades do objeto.

5.2. ECMAScript 2015

Em alguns lugares, lemos ECMAScript 6, ECMAScript 2015, ES6, ES2015; tudo é a mesma coisa e, para simplificar, usaremos sempre ES6 quando referirmos a essa versão. Conhecida como a próxima geração do JavaScript, devido às mudanças que reergueram a utilização da linguagem, nesta versão, as principais mudanças foram:

- Funções de Seta (Arrow Functions)
- Classes
- Template strings
- Destructuring
- Default + Rest + Spread
- let + const
- Modules
- iterators + for..of
- math + number + string + array
- module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- promises
- binary and octal literals

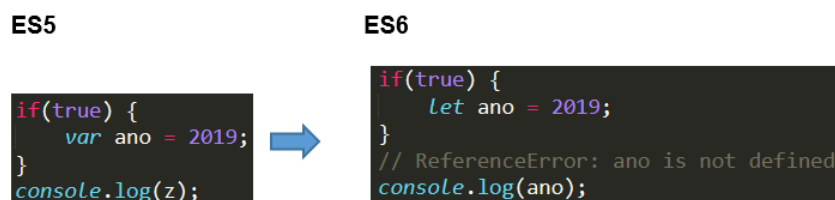
- reflect api
- tail calls
- unicode
- enhanced object literals
- generators

Apresentaremos as principais funcionalidades a seguir:

Let + Const

Na ES5, as variáveis eram declaradas em qualquer lugar do código e poderiam ser sobrescritas sem nenhum problema. No ES6, duas novas palavras reservadas foram criadas para a declaração de variáveis: **let** e **const**. A principal diferença entre **let** e **var** é o escopo do bloco. Vejamos o exemplo abaixo, no qual a variável foi declarada dentro do bloco de chaves e com a utilização do **var**, seu acesso fora do bloco é permitido, enquanto que com a utilização do **let**, a variável não foi definida.

Figura 44 – Exemplo 1 de declaração let.



A declaração utilizando **let** não permite a redeclaração de variáveis, enquanto que **var** permite.

Figura 45 – Exemplo 2 de declaração let.

```

let ano = 1;
let ano = 2; // Uncaught SyntaxError: Identifier 'ano' has already been declared

```

A utilização do **const** na declaração de variáveis tem o mesmo comportamento de **let**, com a diferença de que a variável tem seu valor somente leitura, ou seja, não pode ter seu valor redefinido, conforme exemplo abaixo:

Figura 46 – Exemplo de declaração const.

```
const PI = 3.141593;
PI = 3.0; //Uncaught TypeError: Assignment to constant variable
```

Uma exceção no **const** é o caso do exemplo abaixo, no qual foi declarado um objeto **const** e sua propriedade foi modificada.

Figura 47 – Exemplo de declaração const em objetos.

```
const Ferrari = {
  proprietario: 'Fernando'
};

Ferrari.proprietario = 'Pedro';
```

A utilização de **const** em objetos permite a modificação das suas propriedades, mas a referência à variável não pode ser alterada, ou seja, ela não pode receber um novo valor. O exemplo abaixo modifica o valor de referência da variável.

Figura 48 – Exemplo de declaração const em objetos.

```
const Ferrari = {
  proprietario: 'Fernando'
};

//Uncaught TypeError: Assignment to constant variable
Ferrari = {
  proprietario: 'Pedro'
};
```

Arrows Functions

As funções de seta ou (Arrow Functions) é uma simplificação para a declaração de funções no JavaScript. Considere o exemplo a seguir:

Figura 49 – Comparação de declaração de funções ES5/ES6.

ES5 <pre>var sum = function(a, b) { return a + b; }</pre>	➡	ES6 <pre>const sum = (a, b) => { return a + b; }</pre>
---	---	---

A principal diferença é na eliminação da palavra reservada **function** com a utilização da seta **=>**. Caso a função possua uma instrução, é possível simplificar ainda mais a sintaxe eliminando a palavra reservada **return** e as chaves **{}**, conforme abaixo:

Figura 50 – Funções seta.

```
const sum = (a, b) => a + b;
```

Com a remoção do **return** e as chaves, o que estiver após a **=>** será o valor retornado pela função. Se a função possuir apenas um argumento, podemos remover os parênteses, conforme abaixo:

Figura 51 – Funções seta.

```
const pow = a => a * a;
```

Por outro lado, caso a função não possua argumentos, colocamos os parâmetros vazios:

Figura 52 – Funções seta.

```
const getValor = () => console.log(a);
```

Para retornar um objeto nesta nova sintaxe, é necessário colocar o valor retornado entre parênteses:

Figura 53 – Funções seta.

```
const marcaCarro = marca => ({marca: marca});
```

Além das simplificações acima, as funções de seta não possuem contexto e nem o objeto **arguments**, a palavra-chave **this** será sempre referente ao contexto externo a elas, conforme imagem abaixo:

Figura 54 – Funções seta.

```
const carro = {  
  marca: "Fiat",  
  getCarro: function() { console.log(this) }  
}  
carro.getCarro(); // {marca: "Fiat", getCarro: f}
```

Se utilizássemos as funções seta, teríamos:

Figura 55 – Funções seta.

```
const carro = {  
  marca: "Fiat",  
  getCarro: () => { console.log(this) }  
}  
carro.getCarro(); // Window
```

Classes

As classes no ES6 são um açúcar sintático sobre o padrão OO baseado em **prototype**. Ter uma única forma declarativa conveniente torna os padrões de classes mais fáceis de usar e estimula a interoperabilidade. As classes suportam herança

baseada em protótipo, chamadas do construtor super, instâncias e métodos estáticos e construtores.

Figura 56 – Classes no ES5.

ES5

```
function Carro(marca) {
    this.marca = marca;
}
Carro.prototype.getMarca = function() {
    return this.marca;
}
```

No ES6, pode ser feito desta forma:

Figura 57 – Classes no ES6.

```
class Carro {
    constructor(marca) {
        this.marca = marca;
    }
    getMarca() {
        return this.marca;
    }
}
```

A herança também foi simplificada:

Figura 58 – Heranças no ES6.

```
class Carro extends Veiculo {
    constructor(marca, modelo, estepe) {
        super(marca, modelo);
        this.estepe = estepe;
    }
    getInfo() {
        return "marca: " + super.getMarca() + ", modelo: " + super.getModelo();
    }
}
```

Templates Strings

As templates strings fornecem açúcar sintático para a construção ou interpolação de uma cadeia de strings sem a necessidade de concatenação. As Templates Strings devem estar entre crases (``) e, para interpolar um valor de uma variável, basta adicioná-la entre `${}`, como fizemos com as variáveis `modelo` e `ano`.

Figura 59 – Templates Strings.

ES5

```
var modelo = 'Onix';
var ano = '2019';
var frase = 'Modelo: ' + modelo + ', Ano: ' + ano;
```

ES6

```
const modelo = 'Onix';
const ano = '2019';
const frase = `Modelo: ${modelo} , Ano: ${ano}`;
```

Os Templates String conseguem interpretar uma quebra de linha sem a necessidade de utilizar o `\n`.

Destructuring

A desestruturação (destructuring) permite descompactar valores de vetores ou propriedades de objetos em variáveis distintas. Com esse conceito, podemos inicializar variáveis de uma vez só, como no exemplo abaixo:

Figura 60 – Destructuring de objetos.

ES5

```
var a = 'Maria';
var b = 'Ana';
```



ES6

```
let [a, b] = ['Maria', 'Ana'];
```

Essa característica também é conhecida como desestruturação de array (*Array Destructing*):

Figura 61 – Destructing de objetos.

```
let nomes = ['Maria', 'Ana', 'Carla'];

let [a, b, c] = nomes;
console.log(a); // "Maria"
console.log(b); // "Ana"
console.log(c); // "Carla"
```

Desestruturação feita com objeto:

Figura 62 – Destructing de objetos.

```
const carro = { modelo: 'Onix', ano: 2019 };
const { modelo, ano } = carro;
```

Desestruturação realizada nos parâmetros de uma função:

Figura 63 – Destructing de objetos.

```
const carro = { modelo: 'Onix', ano: 2019 };
const printCarroInfo = ({ modelo, ano }) => `Modelo: ${modelo}, Ano: ${ano}`;
printCityInfo(carro); // Modelo: Onix, Ano: 2019
```

A desestruturação permite realizar a troca de variáveis (*swap variables*) sem a necessidade de criar uma variável temporária.

Figura 64 – Destructing de objetos.

ES5

```
var temp = a;
b = a;
a = temp;
```



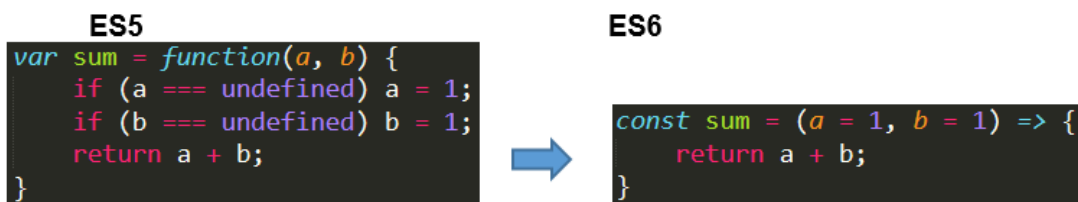
ES6

```
[a, b] = [b, a];
```

Default + Rest + Spread

Na ES6, é possível determinar parâmetros default (Default Parameters) nas funções:

Figura 65 – Parâmetros padrão.



Os parâmetros **rest** nos permite representar um número indefinido de argumentos como um vetor. Sua utilização é no último parâmetro de uma função no qual devemos prefixar com ... o que fará com que todos os argumentos restantes (fornecidos pelo usuário) sejam colocados dentro de um vetor. Importante: Apenas o último parâmetro pode ser um "parâmetro rest".

Figura 66 – Parâmetros rest.

```
function sum(...valores) {
  return valores.reduce((anterior, atual) => {
    return anterior + atual;
  });
}

console.log(sum(1, 2, 3)); //6
console.log(sum(1, 2, 3, 4)); //10
```

A sintaxe de Espalhamento (**Spread syntax**) é utilizada com o operador de espalhamento (...) para passar diversos argumentos para uma função.

Figura 67 – Operador de espalhamento.

```
function sum(x, y, z) {  
  return x + y + z;  
}  
const numeros = [1, 2, 3];  
  
console.log(sum(...numeros)); //6  
console.log(sum.apply(null, numeros)); //6
```

Modules

Os módulos são códigos JavaScript declarados em arquivos separadamente, que permitem a importação de variáveis, funções e classes para um outro código JavaScript. O conceito de módulos foi baseado em outras implementações, como **CommonJS** e **AMD**. Em Node.js, a utilização de módulos é semelhante ao comando *require* (CommonJS), no JavaScript utilizamos o comando **import** e **export**. **Export** é utilizado para definir os valores que serão exportados pelo arquivo. Por meio dele, é possível realizar o **export nomeado** (export named) ou o **export default**. As exportações nomeadas são úteis para exportar várias declarações e o nome da declaração deve ser mantido na importação. As exportações default podem utilizar apenas uma vez no módulo e seu nome não é preservado na importação. **Import** é utilizado na importação de “exports” de outro arquivo JavaScript. Podemos realizar a importação dinâmica, que é útil em situações nas quais o import do módulo deve ser realizado condicionalmente ou sob demanda. A forma estática é preferível para carregar dependências iniciais da aplicação, e pode se beneficiar mais prontamente de ferramentas de análise estática. A Figura 68 exemplifica o uso.

Figura 68 – Export modules JavaScript.

```
//Veiculos.js
export const Fabricantes = ['Fiat','Volkswagen','Chevrolet','Suzuki'];
export const Cor = ['Preto','Branco','Prata'];

export default class Carro extends Veiculo {
  constructor(marca, modelo, cor) {
    super(marca, modelo);
    this.cor = cor;
  }
  getInfo() {
    return "marca: "+super.getMarca() + ", modelo: " + super.getModelo();
  }
}

//Main.js
import Carro, { Fabricantes, Cor } from './Veiculos.js';

let c = new Carro(Fabricantes.Fiat,'Palio',Cor.Preto);
console.log(c.getInfo()); // marca: Fiat, modelo: Palio
console.log(Fabricantes.Volkswagen); // Volkswagen
```

A seguir, nas Figuras 69 e 70, há um descritivo das diversas sintaxes possíveis para utilizar o **import** e o **export**, conforme extraído do MDN.

Figura 69 – Import modules JavaScript.

```
//importar todo o conteudo exportado de um modulo
import * as myModule from './modules/my-module.js';

//importar apenas um item exportado de um modulo
import {myExport} from './modules/my-module.js';

//importar multiplos itens exportados de um modulo
import {foo, bar} from './modules/my-module.js';

//importar um item com um nome mais conveniente
import {reallyReallyLongModuleExportName as shortName}
  from './modules/my-module.js';

//importar multiplos itens renomeando para uma forma mais simples
import {
  reallyReallyLongModuleExportName as shortName,
  anotherLongModuleName as short
} from './modules/my-module.js';

//importar o modulo apenas para o escopo global sem importar nenhuma variavel
import './modules/my-module.js';

//importar o modulo default que foi exportado
import myDefault from './modules/my-module.js';

//importar dinamicamente por uma chamada de funcao --retorna uma promise
import('./modules/my-module.js')
  .then((module) => {
    // Do something with the module.
  });

//alternativa para importar dinamicamente
let module = await import('./modules/my-module.js');
```

Fonte: Adaptado do MDN.

Figura 70 – Export modules JavaScript.

```
// exportar declaracoes realizadas anteriormente
export { myFunction, myVariable };

// exportar um unico item (var, let, const, function, class)
export let myVariable = Math.sqrt(2);
export myFunction() { ... };

// exportar declaracoes realizadas anteriormente como default
export { myFunction as default };

// exportar um unico item como default
export default myFunction() { ... }
export default class { .. }

//renomear declaracoes a serem exportadas
export { myFunction as function1,
        myVariable as variable };

//exportar submodulos agrupando-os
export { myFunction, myVariable } from 'Module1.js';
export { myClass } from 'Module2.js';

// o agrupamento pode ser importado em outro arquivo
import { myFunction, myVariable, myClass } from 'Module.js'
```

Fonte: Adaptado do MDN.

iterators + for..of

O laço de iteração **for...of** é utilizado para iterar objetos iteráveis, ou seja, objetos que definem, em sua estrutura, como eles devem ser percorridos por meio de uma função geradora, na propriedade [Symbol.iterator]. Alguns objetos no JavaScript já são iteráveis, como é o caso do Array, Map, Set e o próprio objeto String. A Figura abaixo exemplifica a utilização do laço de iteração for...of para um vetor de numbers.

Figura 71 – For...of iterador.

```
const numeros = [1,2,3,4,5];
for(let numero of numeros) {
  console.log(numero);
}

// resultado: 1, 2, 3, 4, 5
```

math + number + string + array

Diversas modificações foram realizadas nas bibliotecas de Math, Number, conforme exemplos da Figura abaixo:

Figura 72 – Mudanças na Math.

```
Number.EPSILON
Number.isInteger(Infinity) // false
Number.isNaN("NaN") // false

Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

"abcde".includes("cd") // true
"abc".repeat(3) // "abcabcabc"

Array.of(1, 2, 3) // Semelhante a new Array(...)
[0, 0, 0].fill(7, 1) // [0,7,7]
[1, 2, 3].find(x => x == 3) // 3
[1, 2, 3].findIndex(x => x == 2) // 1
[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
["a", "b", "c"].keys() // iterator 0, 1, 2
["a", "b", "c"].values() // iterator "a", "b", "c"
```

5.3. ES7

A partir de 2015, o comitê decidiu lançar uma versão da ECMAScript a cada ano, na versão de 2016, foi feita a continuação na reforma da linguagem com as principais funcionalidades:

Array.prototype.includes

O método *arr.includes(valueToFind[, fromIndex])*, no Array, determina se um valor está num Array, com retorno do tipo booleano, conforme exemplo da Figura abaixo:

Figura 73 – Método includes em Array.

```
[1, 2, 3].includes(2);    // true
[1, 2, 3].includes(4);    // false
[1, 2, 3].includes(3, 3); // false
[1, 2, 3].includes(3, -1); // true
[1, 2, NaN].includes(NaN); // true
```

Operador de exponenciação

Esse operador é conveniente nas aplicações que envolvem cálculos e análise de dados. Vejamos, no exemplo abaixo, a forma como é utilizada a operação de exponenciação antes da ES7:

Figura 74 – Operador de exponenciação.

```
//Utilizando o operador diretamente
const area = 3.14 * r * r;

//Alternativa utilizando o Math
const area = 3.14 * Math.pow(r, 2);
```

Na ES7, foi introduzido o operador de exponenciação **, conforme exemplo abaixo:

Figura 75 – Operador de exponenciação.

```
//ES7 Novo operador de exponenciacao  
const area = 3.14 * (r ** 2);
```

5.4. ES8

A versão ECMAScript 2017 foi lançada em julho de 2017 pela TC39 e, nesta nova atualização, mudanças significativas na linguagem foram apresentadas, sendo uma delas as Async Functions.

Sintaxe curta de objetos

A definição de propriedades nos objetos, geralmente, é feita com o mesmo nome que seu valor; nesta versão, podemos simplificar a criação dos objetos conforme Figura 76.

Figura 76 – Operador de exponenciação.

```
const nome = 'Paula';  
  
const aluno = {  
  nome // Mesma coisa que nome: nome  
};  
  
console.log(aluno.nome); // Paula
```

async + await

No ES7, foi disponibilizado um novo recurso que são as funções async. Essas funções utilizam a palavra-chave **await** para simplificar a lógica assíncrona e a

legibilidade do código. Se uma função é declarada como **async**, o **await** irá bloquear a execução do código até que a operação assíncrona seja concluída ou falhe. Portanto, adicionado o modificador **async** antes de declarar uma função, a transforma em uma função assíncrona, fazendo com que qualquer processo interno dessa função seja assíncrono. O modificador **await** é utilizado em funções que ficam dentro do escopo de funções assíncronas (somente), fazendo que o fluxo da função assíncrona seja interrompido, esperando pela promise da função interna.

Vejamos, no exemplo da Figura 77, uma função que retorna uma promise, a qual é realizada após 2 segundos e retorna o cálculo da base 2 de um valor recebido como parâmetro.

Figura 77 – Exemplo de promises.

```
function base2s(x) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(2 ** x);  
    }, 2000);  
  });  
}
```

A chamada desta função assíncrona pode ser feita utilizando os conceitos de promises:

Figura 78 – Exemplo de chamada da promise.

```
base2s(1).then((r) => {  
  console.log(r);  
});
```

Entretanto, e se quiséssemos executar a soma de várias chamadas das promises, conforme a Figura 79.

Figura 79 – Exemplo de chamadas múltiplas de promises.

```
let sum = base2s(1)
          + base2s(2)
          + base2s(3);
console.log(sum);
```

Nesta situação, o compilador retornaria *undefined* ou 3 promessas de valores que ainda não foram realizados. Para possibilitar a soma dos valores retornados das promises, poderíamos recorrer à cadeia de promises, conforme Figura 80.

Figura 80 – Cadeia de promises.

```
function somaPromise(){
  return new Promise(resolve => {
    base2s(1).then((a) => {
      base2s(2).then((b) => {
        base2s(3).then((c) => {
          resolve(a + b + c);
        })
      })
    })
  });
}

somaPromise().then((y) => {
  console.log(y);
});
```

As funções `async+await` vieram para facilitar situações como esta e evitar o “Promises Hell”. Podemos reescrever o código da Figura 81, conforme abaixo:

Figura 81 – Função `async+await`.

```
async function somaAssincrona() {
  const a = await base2s(1);
  const b = await base2s(2);
  const c = await base2s(3);
  return a + b + c;
}

somaAssincrona().then((s) => {
  console.log(s);
});
```

String Padding

Novos métodos que manipulam string: **padStart()** e **padEnd()**. Os métodos recebem um número inteiro como argumento e verificam se a string tem aquele tamanho. A sintaxe é definida conforme a Figura 82.

Figura 82 – Sintaxe das funções Padding.

```
str.padStart(tamanhoMinimo [, stringSubstituta]);  
str.padEnd(tamanhoMinimo [, stringSubstituta]);
```

Caso o tamanho da string **str** seja maior que o valor passado como parâmetro (**tamanhoMinimo**), nada é feito. Caso seja menor, é adicionado um espaço no início ou no fim. A Figura 83 exemplifica a utilização da nova funcionalidade:

Figura 83 – Chamada das funções Padding.

```
'IGTI'.padStart(1); // 'IGTI'  
'IGTI'.padStart(6); // ' IGTI'  
  
'IGTI'.padEnd(4); // 'IGTI'  
'IGTI'.padEnd(7); // 'IGTI  '
```

Caso o segundo parâmetro seja informado, seu valor é utilizado para substituir o tamanho excedido, conforme Figura 84.

Figura 84 – Chamada das funções Padding com parâmetro opcional.

```
'IGTI'.padStart(6, 'D'); // 'DDIGTI'  
'IGTI'.padStart(6, 'DAJ'); // 'DAIGTI'  
  
'IGTI'.padEnd(7, 'D'); // 'IGTIDDD'  
'IGTI'.padEnd(7, 'DAJ'); // 'IGTIDAJ'  
'IGTI'.padEnd(13, 'DAJ'); // 'IGTIDAJDAJDAJ'
```

Trailing Commas

Uma funcionalidade útil que não possui muita complexidade, agora não há *SyntaxError* quando adicionamos vírgulas excedentes na separação de argumentos em funções, conforme exemplo da Figura 85:

Figura 85 – Exemplo do Trailing Commas.

```
function Carro(arg1, arg2, arg3,) {  
  // ...  
}  
  
Carro('Fiat', 'Palio', 2019,);
```

Além das mudanças apresentadas acima, também houve as seguintes mudanças: Transferência de dados binários de cópia zero, Mudanças na Math, Fluxos Observáveis, Tipos de SIMD, Meta programação, Propriedades de classe e instância, Sobrecarga de operador, Tipos de valor, Registros, Tuplas e Traits.

5.5. ES9

Além das melhorias descritas abaixo, o ES9 trouxe novas funcionalidades para expressões regulares (RegExp) e uma revisão nos template literals.

Promises.prototype.finally()

A ideia do finally para as promises é seguir a mesma estrutura do conceito sobre try-catch-finally. O finally das promises permite fazer a chamada de um método logo após a promise ser realizada (aceita ou rejeitada). Essa nova funcionalidade elimina a necessidade de repetir o mesmo código no .then() e no catch(). A Figura 86 exemplifica a sintaxe para a utilização desta funcionalidade:

Figura 86 – Promise.finally exemplo.

```
promise
  .then(data => data)
  .catch(err => err)
  .finally(() => {
    // chamada de qualquer método
  })
```

Iteração assíncrona

No ES6, foi apresentada a melhoria do iterator (iterador), um objeto que oferece a funcionalidade de navegar entre os itens de uma lista ou estrutura em sequência. Neste objeto contém o método `next`, o qual retornará o próximo item da lista com as propriedades **value** e **done**, conforme exemplo abaixo:

Figura 87 – Iteração assíncrona.

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]; let it = arr[Symbol.iterator]();
console.log(it.next().value); // 1
console.log(it.next().value); // 2
console.log(it.next().value); // 3
console.log(it.next().value); // 4
```

No ES8, essas chamadas que eram síncronas, agora podem ser feitas de forma assíncrona, basta utilizar a sintaxe **for await of**.

Propriedades Rest + Spread

No ES6, foi disponibilizada a funcionalidade de desestruturação de objetos, uma forma de extrair de um objeto somente as características desejadas. Agora, é possível não só fazer isso, mas como resgatar todo o resto e armazená-lo em uma variável, conforme exemplo da Figura 88:

Figura 88 – Propriedades de rest.

```
const valores = {  
  x:1, y:2, a:3, b:4  
};  
  
const {x,y, ...z} = valores;  
  
console.log(x); // 1  
console.log(y); // 2  
console.log(z); // {a:3, b:4}
```

As propriedades de espalhamento (spread) copiam as próprias propriedades de um objeto fornecido para o objeto recém-criado, conforme a Figura abaixo:

Figura 89 – Propriedades de spread.

```
let n = { x, y, ...z };  
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }
```

Referências

BROWN, Ethan. *Learning JavaScript: JavaScript Essentials for Modern Application Development*. USA: O'REILLY, 2016.

CROCKFORD, Douglas. *JavaScript: The Good Parts*. USA: O'REILLY, 2014.

FLANAGAN, David. *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'REILLY, USA. 2011.

LEANPUB. *Understanding ECMAScript 6*. Disponível em: <<https://leanpub.com/understandings6/read/>>. Acesso em: 29 dez. 2021.

MDN. *Mozilla Developer Network*. Disponível em: <<https://developer.mozilla.org/en-US/docs/Web/JavaScript>>. Acesso em: 29 dez. 2021.

T. PIXLEY, P. *Le Hégarret. DOM Level 3 Events: Document Object Model Level 3 Events Specification*, Editors. World Wide Web Consortium, November 2003. Disponível em: <<http://www.w3.org/TR/DOM-Level-3-Events>>. Acesso em: 29 dez. 2021.