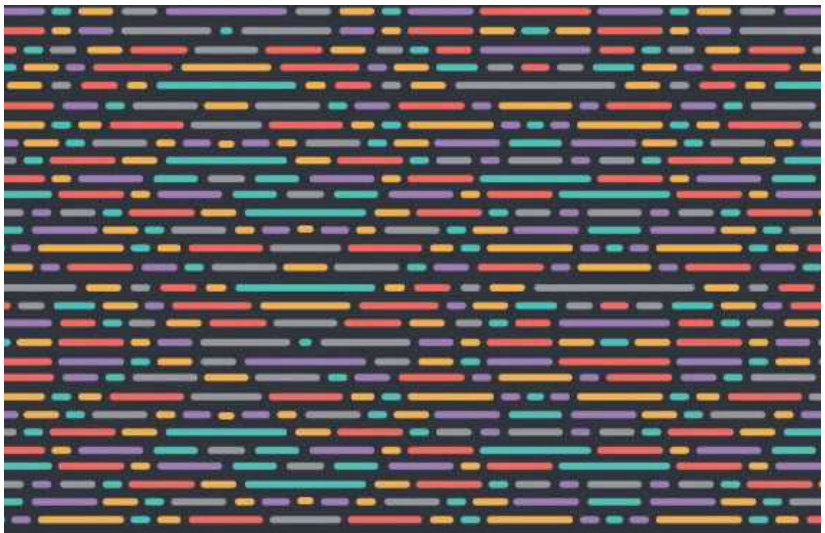
Javascript

Básico ao Avançado

Kelvin Baumhardt Biffi



Javascript

Básico
ao
Avançado

Kelvin Baumhardt Biffi

JavaScript
Básico ao Avançado

Kelvin Baumhardt Biffi

Introdução

Este é um livro que explicará a linguagem de programação JavaScript desde os comandos mais básicos até os conceitos mais complexos.

Durante o percurso da leitura deste livro você verá os princípios da lógica de programação, passando por conceitos básicos necessários para que se possa ter um total entendimento do que se trata a programação em seu todo. Após saber como funciona a linguagem e tendo um conhecimento básico sobre a mesma você começará a entrar no mundo do JavaScript e verá como ele realmente trabalha por baixo dos panos.

Chegando neste ponto você já estará apto a criar a lógica de seus próprios site e aplicativos, porém não terá o conhecimento completo da linguagem, pois é aqui, que as coisas esquentam de verdade e entramos no coração do JavaScript para nos aprofundamos nos conceitos mais fortes da linguagem, que são os objetos e funções, passando por Closures, IIFE, Prototype e assim ter uma total compreensão de o porque que o JavaScript é a linguagem de programação que está dominando o mercado de TI.

O público alvo

Este livro é direcionado para toda e qualquer pessoa que almeja aprender a programar utilizando a linguagem JavaScript ou para profissionais da área que já desenvolvem utilizando a linguagem JavaScript e desejam aprimorar seus conhecimentos desta linguagem de programação que está cada dia mais abrangente entre diversos setores do mercado de TI.

A intenção

A intenção deste livro é apresentar os conceitos mais básicos até os aspectos mais avançados da linguagem JavaScript, capacitando assim todo e qualquer pessoa a se tornar um(a) desenvolvedor(a) JavaScript.

Onde obter os fontes e mais informações

Todos os fontes utilizados e demonstrados neste livro poderão ser obtidos atravez do repositório publico: http://bit.ly/ArquivosJavaScriptPT

Você poderá entrar em contato comigo através do e-mail *kelvinbiffi.developer@gmail.com* ou acessando meu portfolio http://kelvins.cc

Espero que você leia este livro com o mesmo prazer que eu tive enquanto o estava escrevendo.

Ferramentas de desenvolvimento

Durante o decorrer deste livro iremos precisar de algumas ferramentas para poder ter uma melhor compreensão do conteúdo e um aprendizado eficaz.

Neste tópico irei apresentar as três ferramentas que são necessárias para que possamos programar em JavaScript e testar nossos códigos.

Editor de código

Para quem não tem nenhum conhecimento sobre programação, desenvolvimento ou algo parecido, o editor de código é utilizado para ajudar e facilitar a vida dos desenvolvedores, pois escrever centenas de linhas de códigos em um bloco de notas com letras pretas e sem a formatação adequada é algo muito exaustivo.

Desta forma neste livro os exemplos e códigos apresentados serão gerados através do editor de texto denominado Atom, você pode baixar o mesmo através do link https://atom.io/ .

Este editor de texto pode ser instalado nos sistemas operacionais Windows assim como nos sistemas operacionais Linux e MAC.

Uma nota para você que já possui algum conhecimento sobre desenvolvimento e tenha por preferência qualquer outro editor de texto como Brackets, VSCode, Notepad++ ou SublimeText. Sinta-se a vontade de utilizar os mesmos.

Navegador

Durante todo o livro as imagens e exemplos serão apresentados utilizando o navegador Google Chrome, que pode ser baixado através do link https://www.google.com/chrome .

O motivo principal para a escolha deste navegador é a extensão Web Server que irá nos possibilitar criar um servidor local em nosso computador e assim nos possibilitar criar nossos sites e exemplos localmente utilizando um protocolo de comunicação.

Protocolo de comunicação

O protocolo de comunicação é a base para a comunicação de dados na internet.

Nós iremos utilizar este protocolo, pois quando você for criar seus sites ou aplicativos web mais para a frente, quando estiver familiarizado com a linguagem JavaScript e precisar utilizar algum serviço externo, seu servidor local já estará pronto para se comunicar com este sem problemas, sem um protocolo de comunicação não seria possível fazer as requisições para o serviço externo.

Configurando o Web Server

Antes de realizarmos a configuração do Web Server devemos primeiramente instalá-lo no navegador, através do link http://bit.ly/Web_Server .

Agora após ter instalado a extensão no navegador, você deverá criar uma nova pasta, chamada **Server** , no seu computador de preferência na pasta **Documentos** ou no disco local **C** .

Após ter criado a pasta devemos abrir a extensão no navegador, Google Chrome, clicando em **Apps** localizado no início da barra superior de favoritos ou, caso não esteja vendo o ícone de **Apps** em sua tela, acessando o link chrome://apps/ no navegador e selecionar a extensão Web Server.

Ao abrir a extensão você verá um botão **CHOOSE FOLDER** , clique neste botão. Uma tela irá abrir pedindo para que você selecione uma pasta de seu computador, que no nosso caso é a pasta **Server** que criamos recentemente.

Abaixo do botão que você clicou para escolher a pasta do seu servidor local, você verá um botão em formato de interruptor horizontal de cor cinza com um círculo branco na esquerda, clique nele para que ele fique com a cor azul com um círculo azul forte na direita, que significa que seu Web Server está ativo.

Agora para verificar se seu servidor está corretamente configurado, digite o seguinte endereço na barra de navegação do seu navegador **127.0.0.1:8887** .

Você deverá ver esta mensagem na página **Index of current directory…** , simples assim, agora você tem um servidor local para seguirmos adiante com o livro.

JavaScript

Neste capítulo iremos ver sobre as bases do JavaScript, como variáveis, tipos de dados, funções, objetos e muitos outros pontos da linguagem.

Também iremos neste capítulo ver os princípios, fundamentais, sobre programação. O que irá possibilitar a você compreender com mais facilidade qualquer outra linguagem que deseje aprender posteriormente.

Este capítulo irá falar especificamente sobre o básico, então para os desenvolvedores mais experientes talvez seja um pouco monótono, porém aconselho a todos que leiam o livro todo, pois poderá haver pequenos detalhes que vocês não saibam ou talvez tenham esquecido ao longo do tempo.

O que é o JavaScript

JavaScript é uma linguagem de programação que foi criada por meados de 1995 e é uma das bases do tripé para o desenvolvimento web, que são os sites em que estamos acostumados a navegar durante o nosso dia-a-dia.

Sendo ela uma das bases do desenvolvimento web, você pode imaginar que esta linguagem está presente na grande maioria dos sites em que navegamos, porém esta linguagem magnífica não está restrita apenas a sites, há a possibilidade de utilizá-la para muitos outros focos, porém não irei me aprofundar neste assunto.

Agora, por que esta linguagem é uma das bases que nos possibilita criarmos páginas e sites web?

Bem existem três características principais que são:

Alta Performance

Além de ter uma forma consideravelmente simples de escrever, esta linguagem não realiza um consumo excessivo da memória do computador o que nos possibilita criar diversos scripts sem exigir demais da máquina.

Multi-plataforma

Significa que esta linguagem não está restrita a apenas um tipo de sistema ou plataforma, mas podendo ser utilizada em diversas vertentes.

Multiparadigma

Como uma linguagem de multiparadigmas, o JavaScript tende a suportar diversos estilos, paradigmas, de programação, como programação imperativa, programação funcional, orientada a eventos, orientação a objetos e baseado em protótipos.

Existem diversos estilos de programação, e cada um foi criado com um objet ivo, porém as linguagens que aceitam multiparadigmas, assumem, e entendem, que não existe um estilo apenas que irá solucionar todos os problemas de forma rápida e eficaz. Utilizando diferentes estilos em conjunto é de comum acordo que é possível chegar sempre a uma melhor solução do que restringir-se a apenas um paradigma de programação.

Nosso conteúdo é focado em programação Orientada a Objetos, também conhecida pela sigla *PO O* . isso significa que toda a variável ou função que iremos aprender e utilizar durante o livro terá um conceito de objeto. Possivelmente esta última frase não explica muita coisa, então vamos dar uma pequena aprofundada sobre o assunto para entendermos melhor o que é orientação a objetos:

Orientação a objetos é um paradigma de programação que foi criado a partir dos conceitos de cognição, o que significa que é uma forma de programar onde o fluxo de aprendizagem se torna muito rápido e simples, tendo como conceito principal tentar estabelecer que um objeto pode ser requisitado, enviado e muitas vezes moldado, assim como é feito no mundo real.

Desenvolvimento Web

No tópico anterior aprendemos que o JavaScript é um dos pés do tripé que chamamos de desenvolvimento web, então neste tópico iremos explicar como é a estrutura deste tripé e qual a função de cada perna

Conteúdo

A perna central do desenvolvimento web é o HTML que tem como função ser o conteúdo da página, ou seja, toda e qualquer informação que você deseja que o usuário veja, deverá estar no HTML da sua página web.

Apresentação

A forma que você irá apresentar a informação para o seu usuário estará sempre no seu CSS, ou seja, caso você queira que a letra da sua informação seja maior, a cor do seu texto seja rosa, a distância entre as palavras do seu parágrafo seja curta ou longa, tudo isto será definido no seu CSS, também chamado de documento de estilo.

Lógica

Toda a lógica da sua página estará no seu JavaScript, caso você queria que na primeira vez em que alguém entrar na sua página web, seja apresentada para esta pessoa uma mensagem, ou realizar uma verificação de um campo de e-mail em um formulário de cadastro e exibir um alerta de erro quando o campo estiver

preenchido de forma errônea. Qualquer ação de evento e lógica estará no seu JavaScript.

Para algumas pessoas que já trabalham com o desenvolvimento web e devem saber, ou ter ouvido falar, que existem muitas bibliotecas baseadas em JavaScript como AngularJS, React, KnockoutJS entre outras.

Para todas estas pessoas e para as que ainda nem ouviram sequer falar sobre estas bibliotecas, eu aconselho que leiam e estudem bastante sobre a linguagem de programação JavaScript antes de usar qualquer tipo de biblioteca, pois tendo conhecimento suficiente sobre a linguagem JavaScript, vocês poderão trabalhar com estas bibliotecas tendo total entendimento das mesmas e ter controle total do que irão desenvolver. Este livro irá ajudá-los muito neste ponto.

Estrutura básica

Agora que temos uma noção básica sobre o que é a linguagem JavaScript e como se constitui uma página web, vamos partir para um pouco mais de prática e criar a nossa primeira estrutura web.

Criando os arquivos básicos

Tendo como pré-requisito todo o nosso ambiente de desenvolvimento configurado, vamos abrir o editor de código Atom e clicar em *File* > *Open Folder* . Ao clicar nesta opção irá aparecer uma tela para você selecionar uma pasta, então você irá até a pasta **Documentos** , ou disco local **C** , do seu computador e selecione a pasta **Server** que criamos anteriormente.

Após abrir a pasta com o editor de código, clique em cima da pasta **Server** , dentro do Atom , localizado no lado esquerdo da tela, com o botão direito do mouse e clique em ***New File*** , irá abrir uma pequena caixa de texto que você irá digitar **index.html** e pressionar a tecla ***Enter*** .

Irá abrir um arquivo vazio, que você irá escrever a estrutura básica do seu primeiro arquivo HTML:

Index.html

```
01 <!DOCTYPE html>
02 <html lang="pt">
03   <head>
04     <meta charset="utf-8">
05     <title>JavaScript Básico: Estrutura Básica</title>
06  </head>
07   <body>
08     <h1>Primeira estrutura básica</h1>
09  </body>
10 </html>
```

Para as pessoas que estão iniciando agora na área de desenvolvimento, todo e qualquer informação que você deseje

colocar em seu HTML será através de uma tag que são as tags html, head, title, body, h1 e muitas outras.

Como não é a intenção deste livro nos aprofundarmos muito sobre HTML, você pode obter mais informações sobre tags HTML visitando a página http://www.w3schools.com/tags/.

JavaScript

Agora que temos a estrutura básica do nosso HTML, vamos aprender como colocar um JavaScript em nossa página. Há duas formas de fazermos isto.

A primeira forma é colocarmos uma tag script e dentro desta escrevermos nosso código JavaScript:

Index.html

```
01 <!DOCTYPE html>
02 <html lang="pt">
03   <head>
04     <meta charset="utf-8">
05     <title>JavaScript Básico: Estrutura Básica</title>
06     < script >
07         console.log("That's All Folks");
08     </ script >
09  </head>
10   <body>
11     <h1>Primeira estrutura básica</h1>
12   </body>
13 </html>
```

A segunda forma é criarmos um arquivo do tipo JavaScript chamado **script.js** clicando com o botão direito do mouse em cima da pasta **Server** e clicando em ***New File*** . Após criar o arquivo você irá referenciar o mesmo em seu HTML utilizando uma tag script e utilizar o atributo **src** desta:

Index.html

```
01 <!DOCTYPE html>
02 <html lang="pt">
03   <head>
04     <meta charset="utf-8">
05     <title>JavaScript Básico: Estrutura Básica</title>
06     < script   src ="script.js"></ script >
```

```
07   </head>
08
09   <body>
10     <h1>Primeira estrutura básica</h1>
11   </body>
12 </html>
13
```

script.js

```
01 console.log("That's All Folks");
```

Você viu que existem duas formas de utilizarmos o JavaScript em nossa página, porém a melhor prática, e a que devemos sempre utilizar, é utilizarmos os arquivos desta linguagem referenciados em nosso HTML através de uma tag script, pois assim garantirmos uma melhor organização, manutenção e controle do nosso projeto.

Agora depois de vermos as formas de utilizar o JavaScript em nossa página web, vamos ver nossa página no navegador. Para isto abra o Google Chrome e digite o seguinte endereço na barra de navegação http://127.0.0.1:8887/index.html . Ao carregar a página web você irá visualizar um texto bem grande *Primeira estrutura básica* , porém para visualizarmos o resultado de nosso JavaScript precisamos abrir o **console** do navegador, há algumas formas de abrirmos esta ferramenta em nosso navegador.

A primeira é clicar sobre qualquer parte da página web com o botão direito do mouse e selecionar a opção Inspecionar, em seguida clicar na aba **Console** . A segunda forma é pressionar uma vez a tecla **F12** e selecionar a opção **Console** . Após fazer um destes passos você terá o **Console** do seu navegador aberto e deverá estar aparecendo:

Console

```
That's All Folks
```

O Console é um dos componentes que constituem as ferramentas de desenvolvimento do Google Chrome, assim como todo o navegador tem um Console e suas próprias ferramentas de desenvolvimento. Como não é o foco deste livro nos aprofundarmos sobre as funcionalidades do Google Chrome é possível obter informações mais detalhadas sobre as ferramentas de

desenvolvimento do mesmo através do link http://bit.ly/Chrome_DevTools .

Agora que temos uma compreensão de como funciona a estrutura básica de uma página web, temos o conhecimento de como utilizarmos o JavaScript e agora também sabemos como visualizar o Console do navegador, vamos começar a estudar o JavaScript.

Variáveis e Tipos de dados

Neste tópico iremos estudar um conceito fundamental para qualquer linguagem de programação que são as variáveis e tipos de dados e para termos um entendimento de o que são esses conceitos utilizemos um exemplo físico.

Imagine que as variáveis são recipientes e dentro destes podemos colocar diferenciados tipos de produtos, porém um de cada vez. Você pode ter um recipiente preenchido com café e outro recipiente preenchido com nozes. Seguindo esta lógica você pode ter muitos recipientes preenchidos com diversos tipos de produtos e você irá utilizar os produtos contidos nestes recipientes ao longo do tempo e depois voltará a utilizá-los e denovo e denovo.

Utilizando o exemplo prático imagine que os recipientes são as variáveis e os tipos de produtos são os tipos de dados e entenda que é desta forma que são utilizados as variáveis. Você irá preenchê-las com informações e irá utilizar estas variáveis muitas vezes em seu código. Agora tendo uma ideia de como funciona a lógica das variáveis, vamos ver como utilizá-las na linguagem JavaScript.

Na linguagem JavaScript, toda a vez que desejarmos declarar novas variáveis devemos utilizar a seguinte sintaxe (Estrutura de palavras):

var *variavel = valor;*

Sabendo agora como declarar uma variável vamos declarar algumas variáveis em nosso arquivo JavaScript e apresentá-las no Console do navegador:

script.js

```
01 var nomeDaVariavel = "valor da variável";
02 var outraVariavel = 'valor da outra variável';
03 var variavelIdade = 22;
04 var variavelTenhoCabeloBranco = false;
```

```
05 var variavelIndefinida ;
06
07 console.log( nomeDaVariavel );
08 console.log( outraVariavel );
09 console.log( variavelIdade );
10 console.log( variavelTenhoCabeloBranco );
11 console.log( variavelIndefinida );
12 console.log( variavelNula );
```

Agora após digitar o código do exemplo e salvar o arquivo, abra o navegador e pressione F5 para atualizar a sua página web e ver as informações que estão no Console.

Variável

No momento que você declara uma variável você pode identificá-la de qualquer forma como por exemplo **nomezinhoPrimeiro** , **realValor** , **unicornioVoador** , etc. As únicas restrições são que você não pode declarar uma variável chamada **var** , pois esta é uma das diversas palavras reservadas da linguagem JavaScript.

Palavras reservadas são palavras que não podem ser utilizadas pelo desenvolvedor com a finalidade de nomear algo, pois estas palavras compõem a gramática da linguagem de programação como por exemplo na linguagem JavaScript as palavras **for** , **var** , **function** , entre outras .

Você deve ter notado que ao declararmos as variáveis no último exemplo algumas letras estavam como maiúsculo. Isto é uma boa prática para declararmos variáveis, pois toda letra maiúscula indica o início de uma nova palavra, assim quando formos declarar uma variável com um nome muito extenso é possível distinguirmos melhor a finalidade desta, como por exemplo **nomeDoMeuPrimeiroFilho** , esta variável terá como finalidade armazenar o nome do meu primeiro filho. É importante também destacar que no momento em que for declarado uma variável, onde exista m letra s maiúscula s e minúsculas, para utilizar estas variáveis novamente é necessário escrever estas exatamente como

foram declaradas, caso contrário ocorrerá erro na lógica do script, pois a linguagem JavaScript faz diferenciação entre caracteres maiusculos e minusculos, este conceito de análise tipográfica é chamado em inglês de *Case Sensitive* .

Tipos de dados

Ao declararmos as variáveis em nosso exemplo é possível notar que os valores atribuídos as variáveis são diferentes, alguns com aspas, alguns sem aspas, números, etc. Isto ocorre, pois os diferentes tipos de dados necessitam que a sintaxe de atribuição seja diferente. Para melhor entender o assunto vamos dar uma olhada nos tipos de dados existentes na linguagem JavaScript.

Primitivos

Estes dados são considerados primitivos pois não são objetos, são tipos de dados simples e na linguagem de programação JavaScript há cinco deles, que são:

Numérico (Number)

O tipo de dado numérico na linguagem de programação JavaScript será sempre decimal, mesmo que você o declare como 2 a linguagem de programação sempre o entenderá como 2.0, mesmo que você não veja isto.

Em algumas linguagens de programação, denominadas linguagens tipadas, há como definir se uma variável do tipo numérico será inteira ou decimal, mas isto não é necessário no JavaScript.

Para atribuir este tipo de dado a uma variável não é necessário utilizar aspas.

Texto (String)

Tipo de dado texto são basicamente sequências de caracteres como por exemplo o nome de alguém ou uma mensagem de bom dia.

Para atribuir este tipo de dado a uma variável é necessário colocar o texto entre aspas ou aspas duplas, conforme o último exemplo.

Lógico (Boolean)

Tipo de dado lógico pode conter apenas dois possíveis valores que são **true** ou **false** ou seja verdadeiro ou falso.

Para definir este tipo de dado a uma variável é necessário usar **true** ou **false** .

Indefinido (Undefined)

Tipo de dado indefinido é o valor que uma variável recebe automaticamente, pela linguagem JavaScript, ao ser declarada sem que seja definido um valor para a mesma, ou seja uma variável não populada é preenchida com o tipo de dado indefinido, entenderemos melhor isto no decorrer do livro.

Em nosso exemplo foi declarado uma variável chamada **variavelIndefinida** que ao apresentarmos ela no console o resultado será **undefined** .

Nulo (Null)

O tipo de dado nulo é similar ao indefinido, porém este realmente não existe.

Em nosso exemplo estamos tentando apresentar no console uma variável chamada **variavelNula** , porém esta variável não foi

declarada previamente, então ela não existe e consequentemente é preenchida com o valor **null** .

Tipagem dinâmica

Diferente de algumas linguagens de programação, onde ao definir uma variável é necessário especificar o tipo de dado desta, a linguagem JavaScript possui algo chamado tipagem dinâmica, significa que ao declarar uma variável não é necessário definir ou restringir o tipo de dado desta, pois ela irá se adaptar a cada novo tipo de dado atribuído a ela.

Por exemplo, você pode utilizar nosso último exemplo e atribuir um valor numérico para a variável **nomeDaVariavel** , mesmo ela sendo uma variável preenchida inicialmente com um valor do tipo texto, não ocorrerá erro ao ser preenchida com algum outro tipo de dado.

Esta funcionalidade da linguagem JavaScript pode ser muito útil em alguns casos, porém se utilizada de forma desconexa em um projeto sem muita organização pode se tornar um grande risco e causar transtorno, ao tentar corrigir problemas no código.

Coerção

Na linguagem JavaScript há uma propriedade que está dentro da tipagem dinâmica que se chama coerção. Em certos casos, como a soma de duas variáveis, ou alguma outra operação, que veremos mais para frente, contendo tipos de dados distintos. Esta propriedade tenta descobrir o tipo de dado que as variáveis devem ter para que alguma operação seja executada corretamente, quando necessário, e possível, esta propriedade tentará converter o valor das variáveis.

Isto não ocorrerá sempre, pois ao somar duas variáveis do tipo numérico o resultado será a soma aritmética dos valores. Entretanto

ao tentar somar uma variável do tipo texto e uma variável do tipo numérico o resultado será a concatenação dos textos, neste caso a linguagem converteu a variável do tipo numérico para o tipo texto, pois não faz sentido realizar uma operação aritmética entre um texto e um número, veja o exemplo:

script.js

```
01 var variavelTexto = "texto";
02 var variavelNumero1 = 23;
03 var variavelNumero2 = 11;
04
05 console.log(variavelTexto  + variavelNumero2);
06 console.log(variavelNumero1 + variavelNumero2);
```

Mutação de variável

Em todos os exemplos até agora declaramos muitas variáveis, algumas já populadas, outras não, porém em nenhum momento realizamos a alteração do valor destas.

Como já declaramos as variáveis, não é mais necessário a utilização da palavra reservada **var** para continuarmos a utilizá-las.

O primeiro ponto que devemos enfatizar, quando falamos sobre mudar o valor de variáveis, também chamado de mutação de variável, é que a lógica do script é percorrida linha por linha, da esquerda para a direita. Tendo isto em mente é possível alterar o valor de uma variável, apenas se esta linha de alteração estiver abaixo da linha onde foi realizado a declaração da mesma, igualmente o novo valor atribuído para uma variável poderá ser apresentado ao usuário ou utilizado, apenas abaixo da linha onde foi realizado a alteração desta, caso contrário a alteração não irá refletir para a sua página web. Veja o exemplo:

script.js

```
01 var variavel = "valor inicial";
02
03 console.log(variavel);
04
05 variavel = "novo valor";
06
07 console.log(variavel);
```

Alert

Um outra forma de apresentar informações para o usuário em sua página web é através da função **alert** . Esta função tem como objetivo, assim como a função **console** , apresentar informações para o usuário, porém está irá abrir uma pequena janela na sua página web apresentando o conteúdo a ser exibido ao usuário.

Não é uma prática muito aconselhável utilizar **alert** em sites já publicados, normalmente a função nativa **alert** é utilizada durante a etapa de desenvolvimento de um site.

Prompt

Durante o processo de desenvolvimento muitas vezes precisamos alterar repetidamente o valor de uma variável ou queremos por muitas vezes simplesmente utilizar uma informação diferente para testar algo em nosso código, então ao invés de ficar alterando nosso código, utilizamos a função nativa **prompt** .

Ao contrário da função **console** que tem como finalidade apresentar informações no console do navegador, a função **prompt** tem como finalidade solicitar informações do console do navegador.

Semelhante com a função **alert** , a função **prompt** irá apresentar uma pequena tela na sua página web, porém esta irá conter uma caixa de texto onde você poderá informar um valor a ser enviado para o seu código, veja o exemplo:

script.js

```
01 var variavel;
02
03 variavel = prompt ("Informe um valor para esta variável");
04
05 alert (variavel);
```

Veja no exemplo, na linha 1 declaramos uma variável chamada **variavel** , na linha 3 atribuímos a **variavel** o valor que irá retornar

ao utilizamos a função nativa **prompt** , na linha 5 apresentamos na tela o valor da **variavel** utilizando a função nativa **alert** .

Experimente executar este código em seu **script.js** , você verá que ao abrir a página inicial de nosso exemplo aparecerá uma caixa branca no meio da tela com o valor que passamos por parâmetro para a função **prompt** e um local onde você pode digitar algo, logo abaixo verá dois botões **OK** e **Cancelar** , talvez dependendo do idioma de seu computador, o texto nos botões poderá ser diferente, mas a ordem sempre será a mesma, o primeiro botão será para confirmar e enviar o valor que você digitou para seu código e o segundo botão para cancelar a operação.

Operadores

Na linguagem JavaScript existem diversos tipos de operadores, estes operadores nos ajudam a economizar linhas de código e nos auxiliam durante o processo de desenvolvimento do nosso projeto.

Operadores são utilizados na linguagem JavaScript para realização de diversas operações como por exemplo operações aritméticas, subtração e multiplicação, também são utilizados para realizar atribuições de valores, validações, agrupamentos e chamadas de funções, que veremos mais para frente.

Vamos ver aqui alguns exemplos de operadores lógicos:

script.js

```
01 var somaESubtracao = 5 + 5 -7;
02 var subtracaoEMultiplicacao =  30 - 15 * 2;
03 var multiplicacaoEExponenciacao = 2 * 3 ** 2;
04 var agrupamentoEDivisao =  (22 - 2) / 4;
05
06 console.log(somaESubtracao);
07 console.log(subtracaoEMultiplicacao);
08 console.log(multiplicacaoEExponenciacao);
09 console.log(agrupamentoEDivisao);
```

console

```
3
0
18
5
```

Você viu no exemplo alguns operadores e seus resultados no console. Para ter total entendimento de como os operadores funcionam na linguagem JavaScript veja a tabela a seguir, onde você vai encontrar a ordem de precedência dos operadores e a associatividade deles. (Acesse este link http://bit.ly/Operadores_JavaScript para visualizar a tabela completa):

Precedência	Tipo do Operador	Associatividade	Operadores individuais
15	Exponenciação	direita para esquerda	... ** ...
14	Multiplicação	esquerda para direita	... * ...
	Divisão	esquerda para direita	... / ...
	Resto	esquerda para direita	... % ...
13	Adição	esquerda para direita	... + ...
	Subtração	esquerda para direita	... - ...

Precedência

A precedência de operadores determina a ordem em que os operadores são processados. Operadores com maior precedência são processados primeiro, para saber a ordem total de procedência consulte a tabela contida no link http://bit.ly/Operadores_JavaScript .

Associatividade

Associatividade determina a ordem em que operadores da mesma precedência são processados, para consultar a ordem de associatividade completa, consulte a tabela contida no link http://bit.ly/Operadores_JavaScript .

Agora depois de você ter entendido o que é a precedência e a associatividade, você vai entender o porque que o operador de multiplicação é executado antes de uma soma ou subtração e colocar uma operação entre parênteses significa que está agrupando esta operação e consequentemente ela será executada primeiramente de acordo com a ordem de precedência os operadores.

Agora volte para o exemplo e utilizando a explicação e a tabela apresentada, veja o por que dos resultados apresentados no console do navegador.

Decisões lógicas

Lógica Booleana

Antes de começarmos a ver tudo o que abrange as decisões lógicas é importante entendermos como a lógica booleana funciona.

A lógica booleana é uma ramificação da ciência da computação que lida com os valores de verdadeiro e falso (**true** e **false**). Dentro da ciência da computação a lógica booleana é muito importante e se queremos criar decisões lógicas, independente de qual linguagem de programação, realmente eficazes é necessário que tenhamos um ótimo conhecimento da lógica booleana.

Tabela Verdade

Quando falamos sobre lógica booleana é muito importante mencionar a tabela verdade, que utilizamos para entender corretamente os operadores **&&** (E lógico) e **||** (OU Lógico).

&&	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

||	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Como mostra a tabela verdade do operador **&&** (E lógico), a decisão lógica apenas terá um retorno verdadeiro caso todas as decisões em conjunto retornarem verdadeiro (**true**).

Já a tabela verdade do operador **||** (OU Lógico), a decisão lógica terá um retorno verdadeiro se pelo menos uma das decisões lógicas retornar verdadeiro (**true**). Fazendo uma analogia no mundo real para as duas tabelas verdades, podemos pensar o seguinte:

Tabela Verdade && (E Lógico)

Imaginemos que a decisão lógica que criamos, seja se temos mais de uma maçã E se as maçãs são verdes. Se tivermos duas maçãs, mas uma delas for vermelha, a decisão retornará falso. Assim como se tivermos três maçãs e uma for verde e duas forem vermelhas a decisão também retornará falso. A decisão apenas retornará verdadeiro se tivermos mais de uma maçã e todas forem verdes.

script.js

```
01 var numeroMacas = 4;
02 var todasAsMacasSaoVerdes = false;
03
04 if (numeroMacas > 1 && todasAsMacasSaoVerdes === true) {
05   console.log("Temos mais de uma maçã e todas as maçãs são verdes.");
06 } else {
07   console.log("Ou não temos mais de uma maçã ou nem todas são verdes.");
08 }
```

Tabela Verdade || (OU Lógico)

Imaginemos agora que criamos uma decisão lógica onde fazemos um teste para verificar se temos cinco carros OU se todos os carros são da mesma marca. Se tivermos cinco carros de marcas diferente a decisão retornará verdadeiro, pois pelo menos um dos testes retornou verdadeiro, ter cinco carros. Agora se tivermos quatro carros e todos os carros forem de marcas diferentes, a decisão retornará falso, pois nenhuma das premissas retornou verdadeiro.

script.js

```
01 var numeroCarros = 6;
02 var todosCarrosMesmaMarca = false;
03
04 if (numeroCarros === 5 || todosCarrosMesmaMarca === true) {
05   console.log("Ou todos os carros são da mesma marca.");
06   console.log("Ou temos cinco carros.");
07   console.log("Ou as duas coisas juntas.");
08 } else {
09   console.log("Não temos cinco carros e nem todos são da mesma marca.");
10 }
```

Expressões IF e ELSE

Agora que já vimos a tabela verdade, neste tópico vamos tratar sobre uma das questões mais importantes que está presente em todas as linguagens de programação, chamamos de expressões **if** e **else** que são o que nos habilita a tomarmos decisões dentro de nosso código.

Estas expressões também são conhecidas como expressões de controle de estrutura, isso significa que podemos controlar o que será executado por nosso código ou não.

script.js

```
01 var nome = "Ricardo";
02 var idade = 21;
03 var estaSolteiro = true;
04
05 // if (estaSolteiro) { // retorno da decisão true
06 if (estaSolteiro === true) { // retorno da decisão true
07   console.log(nome + " está solteiro!");
08 } else { // se retorno da decisão for false
09   console.log(nome + " está casado!");
10 }
```

Veja o exemplo, declaramos algumas variáveis e criamos um trecho de expressão **if** e **else** onde estamos realizando uma decisão que caso o retorno da decisão lógica seja verdadeiro iremos executar a primeira expressão na linha 7, mas caso o retorno seja negativo iremos executar a segunda expressão na linha 9.

O que se encontra entre parênteses após o **if** nós chamamos de decisão lógica, pois será baseado nesta decisão que iremos

controlar qual expressão será logicamente executada, fazendo assim o controle da estrutura.

Operadores lógicos

Vimos em tópicos anteriores que existem diversos tipos de operadores na linguagem JavaScript, dentre eles estão os operadores lógicos que nos habilitam a criar diversos formatos de decisões lógicas, incluindo decisões compostas e com valores diferentes, vamos ver a tabela de operadores lógicos. (Acesse este link http://bit.ly/Operadores_JavaScript para visualizar a tabela completa):

Precedencia	Tipo do Operador	Associatividade	Operadores individuais
16	NÃO logico	direita para esquerda	! …
11	Menor Que	esquerda para direita	… < …
	Menor ou Igual a	esquerda para direita	… <= …
	Maior Que	esquerda para direita	… > …
	Maior ou Igual a	esquerda para direita	… >= …
10	Igualdade	esquerda para direita	… == …
	Desigualdade	esquerda para direita	… != …
	Igualdade Estrita	esquerda para direita	… === …
	Desigualdade Estrita	esquerda para direita	… !== …
6	E logico	esquerda para direita	… && …
5	OU logico	esquerda para direita	… || …

Utilizando os operadores lógicos descritos na tabela é possível criar uma gama de decisões lógicas, vamos ver exemplos:

script.js
```
01 var variavel1 = 53;
02 var variavel2 = true;
03 var variavelTexto = "zero";
04 var variavelNumero = "24";
```

```
05 var variavelLogica = false;
06 var variavelCurta = "texto curto";
07
08 if (variavel2 && variavelNumero == 24) { // retorno da decisão true
09   console.log("Retorno verdadeira");
10 } else { // se retorno da decisão for false
11   console.log("Retorno falso");
12 }
13 if (variavel1 > 50) { console.log(variavel1 + " é maior que 50"); }
14 if (variavelTexto != "um") { console.log(variavelTexto + " é diferente de um"); }
15 if (!variavelLogica) { console.log("Negação da variável lógica é verdadeira"); }
16 if (variavelTexto != variavel1) { console.log("Valor das variáveis é diferente"); }
```

Utilizando então os operadores lógicos existentes na linguagem JavaScript nós criamos várias decisões lógicas. Inclusive podemos notar que em vários casos não utilizamos a expressão **else** apenas foi necessário a expressão **if** .

Vamos explicar cada uma das decisões lógicas no exemplo.

Decisão Lógica Composta

Lembra da tabela verdade que vimos antes, então, na primeira decisão lógica que está presente nas linhas de 8 a 12 foi criado uma decisão composta onde para que o valor de retorno seja verdadeiro, ambas decisões devem retornar um valor verdadeiro. Isto ocorre pois utilizamos a operação lógica **&&** (E) que só permite que o retorno da operação seja verdadeiro, caso todas as operações simples tiverem um retorno verdadeiro.

Seria possível criar uma decisão lógica composta que retornasse verdadeiro caso uma ou as duas decisões lógicas simples retornassem verdadeiro utilizando o operador lógico **||** (OU) ao invés do operador lógico **&&** (E). Assim bastaria apenas uma das decisões lógicas simples retornar verdadeiro para que fosse executado o primeiro bloco de código que se encontra na linha 9.

Decisão Lógica Simples

As seguintes decisões lógicas que se encontram nas linhas de 13 a 16 são decisões lógicas simples que utilizam diversos operadores lógicos e sem a criação da instrução **else** caso a decisão retorne falso.

Operador Maior Que:

Na linha 13 criamos uma decisão lógica para validar se o valor da variavel **variavel1** é maior que o número 50, assim qualquer valor que a **variavel1** venha a receber for maior que 50 a decisão lógica retornará verdadeiro.

Operador de Desigualdade:

Na linha 14 criamos uma decisão lógica para validar se a variavel **variavelTexto** é diferente do texto **um** , assim qualquer valor que a **variavelTexto** venha a ter for diferente do texto **um** a decisão lógica retornará verdadeiro.

Operador Não Lógico:

Na linha 15 criamos uma decisão lógica para testar se a negação (**!**) da variavel **variavelLogica** retornaria verdadeiro, sendo assim enquanto a variável tiver o valor **false** a negação seria **true** retornando assim verdadeiro e executando o bloco de código da linha 15.

Operador Desigualdade entre Variáveis

Na linha 16 criamos uma decisão lógica para validar se o valor da variavel **variavelTexto** é diferente do valor da variavel **variavel1**

, assim enquanto o valor das variáveis forem diferentes o retorno da decisão lógica será verdadeiro, entrando dentro do bloco de código destinado a expressão **if** .

Coerção

Vimos alguns exemplos de operadores lógicos e como se comportam, porém um detalhe muito importante em relação aos operadores de comparação são os operadores **==** (Igualdade) e **===** (Igualdade Estrita).

A grande diferença entre eles é que na linguagem JavaScript o operador de igualdade executa as comparações com coerção de dados, enquanto o operador de igualdade estrita não utiliza coerção.

script.js

```
01 var variavelCoercao = 37 ;
02 var variavelSemCoercao = 37 ;
03
04 if (variavelCoercao == "37")  {
05   console.log("Comparação com coerção.");
06 }
07
08 if (variavelSemCoercao === "37" ) {
09   console.log("Este trecho não será esecutado, sem coerção.");
10 }
```

Vendo o exemplo, a primeira decisão lógica retornará verdadeiro, pois ao realizar a operação de **==** (Igualdade) a linguagem automaticamente tentou converter o valor de ambas as variáveis para o mesmo tipo de dado para efetuar a comparação, retornando assim verdadeiro pois ambos os valores são 37.

Entretanto ao vermos a segunda decisão lógica, vemos o operador **===** (Igualdade Restrita) tentando comparar o valor da variável com o texto 37, o que retornará falso, pois sem a coerção de dados não é possível um retorno verdadeiro ao comparar dois tipos de dados diferentes.

Normalmente tenha como orientação utilizar o operador **===** (Igualdade Restrita) ao compararmos variáveis em nosso código. A importância de mencionar isto é porque normalmente realizar comparação apenas com **==** pode acabar causando dificuldade em encontrar problemas em nossos códigos no futuro, sendo assim utilizar **===** se torna mais seguro, porém mais restrito.

Else If

Além de decisões lógicas simples e compostas podemos criar também decisões lógicas caso um decisão lógica retorne negativo, parece estranho, porém veja o exemplo:

script.js

```
01 var numeroPassageiros = 23;
02
03 if (numeroPassageiros === 0) {
04   console.log("O transporte está vazio.");
05 } else if (numeroPassageiros > 0 && numeroPassageiros < 50) {
06   console.log("O transporte tem passageiros.");
07 } else {
08   console.log("O transporte está lotado.");
09 }
```

Veja que criamos uma decisão lógica para verificar se a variável **numeroPassageiros** é igual a zero. Caso o retorno desta decisão seja negativo irá para o **else** logo abaixo, porém neste **else** não é um simples **else** é um **else if** que está verificando se a variável é maior que zero e menor que 50. Caso esta decisão lógica retorne verdadeiro, em nosso código irá retornar verdadeiro, irá executar o bloco de código destinado a decisão **else if** na linha 6. Caso está segunda decisão lógica também retorne falso, então sim irá ser executado o bloco de código **else** que se encontra na linha 8.

Neste exemplo que criamos, existe apenas um **else if** , mas poderíamos ter criado diversas decisões lógicas do gênero como por exemplo verificar se a variável está entre zero e vinte e cinco para saber se o transporte está meio vazio e assim por diante.

Switch

Ainda falando sobre decisões lógicas, existe uma forma mais elegante de criar diversas decisões lógicas para uma variável, ao invés de criar vários comandos **if else if else** e assim por diante. Esta forma se chama **switch** .

Imagine que você precisa criar um código para verificar qual é a profissão de alguém, no que esta pessoa trabalha. Existem diversos tipos de trabalhos e poderíamos criar esta verificação utilizando comandos **if else** , mas para deixar nosso código mais apresentável, vamos usar o comando **switch** , vejamos o exemplo:

script.js

```
01 var nome = "Michele";
02 var emprego = "desenvolvedor";
03
04 switch ( emprego ) {
05   case "policial":
06     console.log(nome + " é uma policial.");
07     break ;
08   case "cozinheiro":
09     console.log(nome + " é uma cozinheira.");
10     break ;
11   case "desenvolvedor":
12     console.log(nome + " é uma desenvolvedora");
13     break ;
14   case "lutador":
15     console.log(nome + " é uma lutadora.");
16     break ;
17   default :
18     console.log(nome + " trabalha em outra coisa.");
19 }
```

No exemplo temos a variável **nome** com o nome de uma pessoa e uma outra variável **emprego** com a informação no que esta pessoa trabalha.

Veja a declaração **switch** há entre parênteses a variável **emprego** , isto significa que será está a variável verificada. O **case** seria a decisão lógica, assim como o **if** .

O **case** verifica se caso o valor da variável **emprego** seja uma das opções ali declaradas como **policial** , então executará o código

logo abaixo.

O **break** é obrigatório ser utilizado dentro do **case** , pois serve para parar o código, caso o valor correto seja encontrado, caso contrário o **switch** , mesmo tendo achado o valor correto, sem o **break** vai continuar tentando achar o valor correto.

O **default** pode ser comparado com a declaração **else** , caso nenhum dos **case** sejam compatível com o valor da variável **emprego** o bloco abaixo do **default** será executado.

Em nosso e xemplo a linha que será executado será a linha 12.

Veja neste link http://bit.ly/Exemplo_if_else co mo ficaria esse exemplo utilizando **if else**

Funções

Função, este é um tópico de extrema importância na linguagem JavaScript assim como para todas as outras linguagens de programação. Podemos pensar que uma função é uma máquina, onde você envia uma informação para esta máquina e ela te devolve algum outro valor ou alguma informação importante.

DRY

Nós criamos funções normalmente quando precisamos executar um pedaço de código várias vezes em nosso código, então ao invés de reescrever o mesmo código dezenas de vezes, colocamos este código dentro de uma função para que possamos reutilizar este código várias vezes. Existe um princípio em programação chamado DRY (Don't Repeat Yourself).

Este princípio tem como finalidade prevenir uma gama exagerada de códigos repetidos em nossos scripts e utilizando funções conseguimos aplicar este princípio com facilidade.

Função

Como mencionado antes, uma função é como uma máquina, você pode passar informações para ela, chamamos essas informações em programação de parâmetros. A função vai manipular e tratar estes parâmetros e depois vai devolver um novo valor.

Imagine que precisamos calcular a idade de uma pessoa e depois verificar se esta pessoa é maior de idade, veja como ficaria este trecho de código no exemplo:

script.js

```
01 var nomePessoa = "Joana";
```

```
02 var anoDeNascimento = 2002;
03
04 var idade = 2018 - anoDeNascimento;
05
06 if (idade >= 18) {
07   console.log(nomePessoa + " é maior de idade.");
08 } else {
09   console.log(nomePessoa + " tem menos de 18 anos.");
10 }
11
12 nomePessoa = "Cassiano";
13 anoDeNascimento = 1988;
14
15 var idade = 2018 - anoDeNascimento;
16
17 if (idade >= 18) {
18   console.log(nomePessoa + " é maior de idade.");
19 } else {
20   console.log(nomePessoa + " tem menos de 18 anos.");
21 }
```

Podemos ver que na linha 6 e na linha 17 fazemos a mesma decisão lógica e nas linhas 4 e 15 fazemos o mesmo cálculo.

Este trecho de código funciona, porém imagine que tenhamos cinquenta pessoas, teríamos então que repetir estes trechos várias e várias vezes. Olhe como ficaria se utilizássemos uma função no exemplo:

script.js

```
01 function verificarIdade(nomePessoa, anoDeNascimento) {
02   var idade = 2018 - anoDeNascimento;
03   if (idade >= 18) {
04     return nomePessoa + " é maior de idade.";
05   } else {
06     return nomePessoa + " tem menos de 18 anos.";
07   }
08 }
09
10 var nome = "Joana";
11 var nascimento = 2002;
12 console.log(verificarIdade(nome, nascimento));
13
14 console.log(verificarIdade("Cassiano", 1988));
15
16 nome = "John";
17 nascimento = 1978;
18 var texto = verificarIdade(nome, nascimento);
19 console.log(texto);
```

Veja no novo código estamos fazendo os mesmos cálculos e verificações, porém colocamos toda a lógica dentro de uma função, evitando assim reescrever nosso código e o deixando mais limpo e organizando.

Estrutura de uma função

Primeiro de tudo, vamos entender como declaramos uma função. Para declarar uma função devemos escrever a palavra reservada **function** em nosso código e logo em seguida escrever o nome da nossa função, assim como as variáveis, uma função pode ter qualquer nome que desejemos, como **meMostraOResultado** ou **vaiECalculaIsso** . Após o nome da função, dentro dos *parênteses* , nós declaramos nossos parâmetros, separados por vírgulas, que funcionam como variáveis a serem utilizadas dentro da função, também podem ter qualquer nome que desejemos. Este é um ponto importante, os parâmetros e variáveis declarados para uma função podem ser acessados apenas dentro desta mesma função, se tentar acessar essa variável ou parâmetro fora da função irá gerar um erro em seu código informado que o parâmetro ou variável não foi definida, entenderemos melhor este ponto no decorrer do livro.

Como uma função funciona

Quando o JavaScript lê as linhas onde tem chamadas de funções ele simplesmente armazena essas chamadas em memória e espera para que as usemos o que ocorre nas linhas 12,14 e 18. Nessas linhas estamos executando a função e passando parâmetros para ela de fato. Os parâmetros são passados para a função na ordem em que foram declarados, então se você tentar passar o nome da pessoa como segundo parâmetro ao invés do primeiro, esse valor será armazenado no parâmetro **anoDeNascimento** e acabará gerando um erro quando o código

tentar subtrair 2018 pelo valor da variável **anoDeNascimento** , pois o valor será um texto e não um número como esperado.

Como em nosso exemplo passamos todos os parâmetros na ordem correta, nossa função vai calcular então a idade de cada pessoa e retornar uma mensagem em texto que podemos então armazenar em um variável como na linha 18 ou utilizar o retorno diretamente como nas linhas 12 e 14.

Uma função também pode chamar outras funções e uma função não precisa necessariamente retornar um resultado, poderíamos por exemplo ter colocado o log da mensagem de retorno dentro da função como no exemplo:

script.js

```
01 function verificarIdade (nomePessoa, anoDeNascimento) {
02   var idade = 2018 - anoDeNascimento;
03   if (idade >= 18) {
04     console.log(nomePessoa + " é maior de idade.");
05   } else {
06     console.log(nomePessoa + " tem menos de 18 anos.");
07   }
08 }
09
10 var nome = "Joana";
11 var nascimento = 200 2 ;
12 verificarIdade (nome, nascimento);
13
14 verificarIdade ("Cassiano", 1988);
15
16 nome = "John";
17 nascimento = 1978;
18 verificarIdade (nome, nascimento);
```

Sabendo agora como funções e variáveis funcionam, tente modificar a função **verificarIdade** do último exemplo. Recorte o trecho de código que valida se a pessoa é maior de idade ou não, da linha 3 a 7, coloque esta verificação dentro de uma outra função chamada **verificarMaiorIdade** , faça essa nova função retornar a mensagem como no penúltimo exemplo e armazene esse retorno em uma variável chamada **mensagemRetorno** e faça um log desta mensagem. Você pode achar esta modificação neste link http://bit.ly/exempllo_funcao_mensagem .

Declarações e Expressões

Uma questão que ainda não vimos é a diferença entre uma declaração (*statement*) e uma expressão (*expression*). A grande diferença entre uma expressão e uma declaração é que uma expressão produz um valor, uma saída que pode ser armazenada, enquanto uma declaração apenas executa uma ação, não gera um valor imediatamente para poder ser armazenado.

Vamos ver um exemplo:

script.js

```
01 // Declaração (Statement)
02 function minhaFuncao(param1) {
03   console.log("Executa uma ação");
04 }
05 if (1 !== 2) {
06   console.log("Um IF é uma declaração");
07 }
08 console.log(minhaFuncao);
09
10 // Expressão (Expression)
11 var minhaOutraFuncao = function (param1) {
12   console.log("Executa outra ação");
13 }
14 console.log(minhaOutraFuncao);
15
16 // Uma soma é uma expressão
17 var soma = 5 + 7;
18 console.log("Valor armazenado na variável 'soma': " + soma);
```

No exemplo nós criamos declarações e expressões. Na linha 2 nós declaramos uma função chamada **minhaFuncao** e na linha 5 nós declaramos uma decisão lógica, nenhuma dessas linha está retornando um valor.

Agora observe as linhas 11 e 17, na linha 11 não estamos simplesmente declarando uma função, estamos armazenando o valor da função em uma variável chamada **minhaOutraFuncao** e na linha 17 estamos realizando uma operação de adição e armazenando a saída desta operação na variável **soma** .

A grande diferença entre as linhas 2 e 11 é que na linha 2 estamos criando uma função e na linha 11 estamos criando uma expressão de função, que consistente em armazenar o valor de uma função anônima dentro de uma variável, veremos mais sobre este tópico no decorrer do livro. Nas linhas 8 e 14 estamos fazendo um log da estrutura das duas funções para que seja possível comparar elas no console do navegador.

Console
Um IF é uma declaração
f minhaFuncao(param1) {
console.log("Executa uma ação");
f (param1) {
console.log("Executa outra ação");
}
Valor armazenado na variável 'soma': 12

Saber esta informação é importante e vai nos ajudar a compreender alguns aspectos daqui para a frente.

Arrays

Arrays, também conhecidos como Matrizes, podem nos ajudar a criar códigos mais limpos e organizados assim como facilitar muitas questões e a nos ajudar a economizar várias linhas de código.

Imagine que precisamos criar um script para calcular a pontuação de três jogadores os identificando pelo nome e idade. Primeiramente criaremos variáveis para esses jogadores, para ser mais exato precisaríamos criar nove variáveis inicialmente para armazenar o nome, pontos e idade de cada um dos jogadores. Agora imagine por exemplo ser possível agrupar essas informações em uma única variável, na linguagem JavaScript, assim como em outras linguagens de programação, é possível e existem algumas maneiras de fazermos isto, uma forma é utilizando **array** . Veja o exemplo:

script.js

```
01 var arrJogadores = ['Ricardo', 'Monica', 'Leonardo'];
02 var arrAnoNascimento = new Array(1998, 1975, 1991);
03 var arrPontos = [45, 117, 78];
04 arrPontos[1] = 71;
05 console.log(arrJogadores);
06 console.log(arrJogadores[1] + ' tem ' + arrPontos[1] + ' pontos.');
```

Veja o exemplo que criamos. Na linha 1 utilizamos a forma mais comum de criar um **array** , utilizando colchetes e entre os colchetes colocamos os nomes dos jogadores separados por vírgulas. Uma outra forma de criar um **array** é como fizemos na linha 2 onde utilizamos a expressão **new Array()** para criarmos um **array** de anos de nascimento separados por vírgulas dentro dos parênteses.

Para utilizar as informações contidas em um **array** temos que entender que a primeira posição de um **array** , a primeira informação se encontra no index 0, a segunda informação se encontra no index 1 e assim por diante, como mostra no log em exposição no console do navegador, ou seja para retornar o valor de

uma posição de um **array** , temos que buscar pela casa dele menos o número 1 como fizemos na linha 6.

Para modificarmos o valor de uma posição de um **array** é muito semelhante a forma que utilizamos para retornar o valor de um **array** , como fizemos na linha 5, alterando o valor da posição 1 do array **arrPontos** .

Nestes exemplos que criamos, vimos apenas arrays com os mesmos tipos de dados, porém um array pode conter quantos tipos de dados desejarmos, assim como outros arrays dentro de arrays, vejamos um outro exemplo.

script.js

```
01 // new Array(nome jogador, ano nascimento, pontos)
02 var arrRicardo = new Array('Ricardo', 1998, 45);
03 var arrMonica = new Array('Monica', 1975, 71);
04 var arrLeonardo = new Array('Leonardo', 1991, 78);4
05 var arrJogadores = new Array(arrRicardo, arrMonica, arrLeonardo);
06 console.log(arrJogadores);
```

Neste exemplo criamos um **array** para cada jogador com suas respectivas informações como pode ser visto nas linhas 2,3 e 4. Após a criação destes arrays criamos uma variável **arrJogadores** do tipo **array** e colocamos dentro deste **array** as variáveis dos jogadores separados por vírgulas, e apresentamos o **arrJogadores** no console do navegador na linha 6, assim vemos que realmente podemos utilizar qualquer valor dentro de um **array** .

Métodos

Existem vários métodos e funções específicos do tipo de dado **array** , vamos ver alguns deles aqui (Acesse este l ink http://bit.ly/Array_JS p a ra ver a lista completa de métodos do tipo de dado **array**).

Push

A função **push** consiste em adicionar um valor, variável ou objeto depois da última posição de um array, veja:

s cript.js
```
01 var arrPrato = new Array('Arroz', 'Tomate', 'Alface');
02 arrPrato.push('Feijão');
03 console.log(arrPrato);
```

Veja que neste exemplo criamos um **array** com inicialmente três alimentos, sendo o último o Alface, porém na linha 2 utilizamos o método **push** para adicionar o alimento Feijão após o Alface e na linha 3 fazemos um log deste **array** no console do navegador.

Unshift

A função **unshift** consiste em adicionar um valor, variável ou objeto na primeira posição de um array, fazendo assim todos os outros valores avançarem uma casa a frente, veja o exemplo:

s cript.js
```
01 var arrEmenta = new Array('Matemática', 'Artes', 'Francês');
02 arrEmenta.unshift('Português');
03 console.log(arrEmenta);
```

Veja que neste exemplo criamos um **array** com inicialmente três disciplinas, sendo a primeira Matemática, entretanto na linha 2 utilizamos o métodos **unshift** para adicionar a disciplina Português antes de Matemática e na linha 3 fazemos um log deste **array** no console do navegador, podemos ver que agora a disciplina Português é a primeira informação do **arrEmenta** e a disciplina Matemática passou a ser a segunda.

Pop

A função **pop** consiste em remover a última posição de um **array** , utilizemos o último exemplo como base, veja:

script.js
```
01 var arrEmenta = new Array('Matemática', 'Artes', 'Francês');
```

```
02 arrEmenta.unshift('Português');
03 arrEmenta.pop();
04 console.log(arrEmenta);
```

Veja que neste exemplo, como no anterior, nós adicionamos a disciplina Português como a primeira informação do **array** , porém agora que a disciplina Francês vai sair desta ementa, utilizamos a função **pop** na linha 3 para remover a última posição do **arrEmenta** e logo após na linha 4 realizamos um log no console do navegador.

Shift

A função **shift** consiste em remover a primeira posição de um array, veja:

script.js
```
01 var arrMedicos = new Array('Dr. Barcelos', 'Dra. Fernanda');
02 arrMedicos.shift();
03 console.log(arrMedicos);
```

Veja que no exemplo criamos um **array** com inicialmente dois nomes de doutores, porém hoje é o dia de pesquisa apenas da doutora Fernanda, assim utilizamos o método **shift** para remover a primeira posição do **array** e deixando assim apenas a Dra. Fernanda.

IndexOf

O método **indexOf** é muito útil, ele consiste em retornar a posição de um certo valor, caso ele exista, caso contrário ele retornará **-1** , utilizemos o exemplo de prato de comida como base, veja:

script.js
```
01 var arrPrato = new Array('Arroz', 'Tomate', 'Alface');
02 arrPrato.push('Feijão');
03 if (arrPrato.indexOf('Arroz') !== -1) {
04   arrPrato.shift();
05 }
```

06 console.log(arrPrato);

Veja o exemplo, recebemos uma informação do garçom que o cliente que pediu o prato é alérgico a Arroz, então na linha 3 criamos uma decisão lógica para verificar se o alimento Arroz existe dentro da variável **arrPrato** , como neste caso a decisão lógica retornará verdadeiro, pois retornará **0** que é dife rente de **-1** , porque existe realmente Arroz no prato, na linha 4 removemos o alimento Arroz do **array** e na linha 6 realizamos um log da variável **arrPrato** no console do navegador.

Objetos e Propriedades

Agora veremos um dos tópicos mais importantes na linguagem JavaScript que são os Objetos e Propriedades.

No último tópico vimos arrays e algumas de suas funções, também como pegar e alterar o valor da posição de um array por seu identificador numérico, porém imagine agora que você quer alterar um valor, mas não informando um indicador e sim um nome para uma posição em específico.

Quando utilizamos objetos é possível fazer isto, pois existe um aspecto em objetos chamado de par de valores-chave, conhecido também em inglês como *key-value pair (KVP)* .

Key-value pair (KVP)

Um conjunto de dois itens de dados vinculados, uma chave exclusiva nomeada para identificar um dado ou conjunto de dados, cada valor tem um identificador, este identificador é um ponteiro para a localização dos dados.

Quando utilizamos objetos não precisamos nos preocupar com a ordem dos valores ou informações. Objetos não exigem uma ordenação específica o que é diferente de quando utilizamos arrays, pois quando precisamos retornar um valor de um array, precisamos saber exatamente em qual posição está cada informação. Quando utilizamos objetos, basta saber o nome da chave que identifica o valor que desejamos retornar, veja o exemplo:

script.js

```
01 var objetoCarro = {
02   modelo: '147',
03   numeroVidros: 4,
04   marca: 'Fiat',
05   numeroPortas: 2
06 }
07
```

```
08 var objetoCasa = new Object();
09 objetoCasa.numeroComodos = 5;
10 objetoCasa.valorImovel = 523000.320;
11 objetoCasa['aluguel'] = false;
12 objetoCasa['venda'] = true;
13
14 objetoCarro.numeroPortas = 4;
15 var umIdentificadorDaCasa = 'valorImovel';
16
17 console.log(objetoCarro);
18 console.log(objetoCarro.modelo);
19 console.log(objetoCasa['numeroComodos']);
20 console.log(objetoCasa[umIdentificadorDaCasa]);
```

Vamos entender o exemplo, na linha 1 utilizamos chaves, uma das formas que existem para declarar um objeto, criamos um objeto nomeado **objetoCarro** , entre as linhas 2 e 5 definimos 4 propriedades para este objeto **objetoCarro** .

Na linha 8 criamos um outro objeto chamado **objetoCasa** utilizando a declaração **new** a segunda forma existente para a criação de um objeto, entre as linhas 9 e 12 definimos algumas propriedades para este objeto **objetoCasa** , veja que nas linhas 9 e 10 utilizamos notação de ponto para declarar o valor para duas propriedades do objeto e nas linhas 11 e 12 utilizamos a notação de colchetes para declarar outros dois valores para o objeto.

Na linha 14 utilizamos a notação de ponto para mutar o valor da propriedade **numeroPortas** do objeto **objetoCarro** . Na linha 15 declaramos uma variável com um valor de texto que é exatamente o nome da chave **valorImovel** e na linha 20 utilizamos esta variável para retornar o valor da propriedade **valorImovel** do objeto **objetoCasa** utilizando notação de colchetes. Nas linhas de 17 a 20 utilizamos a mesma sintaxe que utilizamos para mutar a propriedade de um objeto, agora para exibir o valor destas propriedades no console do navegador.

Notação

Chamamos de notação a forma utilizada para retornar um valor de um objeto.

As duas formas possíveis de notação para retornar os valores de um objeto são a notação de ponto e a notação de colchetes, como visto no exemplo.

Sabendo agora como criar um objeto, podemos utilizar objetos para agrupar dados semelhantes, como de uma casa ou de um carro, de uma forma mais estruturada, algo que existe uma certa dificuldade quando utilizamos arrays para agrupar dados.

Podemos ver também no exemplo que criamos que as propriedades de um objeto aceitam diferentes tipos de dados como valores booleanos, numéricos e textos, porém quando trabalhamos com objetos, nós podemos ir além, como por exemplo utilizar objetos, arrays ou mesmo funções dentro de objetos, veja o exemplo:

script.js

```
01 var carro = {
02   extras: ['ar-condicionado', 'freios abs', 'som'],
03   velocidade: 0,
04   faroisLigados: false,
05   ligarFarios: function () {
06     this.faroisLigados = true;
07   },
08   acelerar: function (velocidade) {
09     this.velocidade += velocidade;
10   },
11   freiar: function () {
12     this.velocidade = 0;
13   }
14 }
15 carro.ligarFarios();
16 console.log('Farois ligados: ' + carro.faroisLigados);
17 carro.acelerar(50);
18 console.log('Velocidade: ' + carro.velocidade);
19 carro.freiar();
20 console.log('Velocidade: ' + carro.velocidade);
```

Diferente do último exemplo, declaramos neste novo exemplo, arrays e funções, que podemos chamar de métodos, para propriedades do objeto nomeado **carro** . Na linha 5 declaramos uma função anônima para a propriedade **ligarFarios** , esta função altera o valor da propriedade **faroisLigados** do objeto **carro** para **true** , como se fosse a função de um carro real, executamos esta função na linha 15 do exemplo.

Na linha 8 declaramos uma função anônima com um parâmetro chamado **velocidade** para a propriedade **acelerar** do objeto **carro** , está definido nesta função que para cada vez que ela for chamada ela irá somar na propriedade **velocidade** do objeto **carro** a velocidade passada por parâmetro na função anônima, executamos este método na linha 17 do exemplo passando o valor 50 por parâmetro.

Na linha 11 declaramos uma função anônima para a propriedade **freiar** , esta função define o valor da propriedade **valocidade** do objeto **carro** para o valor 0, executamos esta função na linha 19 do exemplo.

Veja como é o resultado dos logs gerados:

Console
Farois ligados: true
Velocidade: 50
Velocidade: 0

Experimente alterar os valores do exemplo, criar novos logs no exemplo e ver quais serão os resultados obtidos.

this

Antes de irmos para o próximo tópico vamos ver um questão de extrema importância quando trabalhamos com objetos e que precisamos ver para termos um total entendimento do exemplo em questão. No exemplo, mais especificamente nas linhas 6, 9 e 12, utilizamos a declaração **this** com a notação de ponto para poder modificar o valor das propriedades do objeto **carro** , isto foi possível, pois **this** é um palavra reservada que se refere ao objeto atual em questão. Então para podermos acessar o valor da propriedade de um objeto, devemos utilizar a declaração **this** , seja para alterar o valor de uma propriedade ou para retornar o valor da mesma, veremos mais sobre **this** no decorrer do livro.

Laços e Interações

Vamos ver agora uma das expressões de controle de estrutura mais importantes, presente em todas as linguagens de programação, assim como na linguagem JavaScript. Chamamos essas expressões de controle de estrutura de Laços de repetição, conhecido em inglês como *loops* .

Imagine que você está escrevendo um trecho de código e está reescrevendo este trecho continuamente em sequência, como por um exemplo simples, exibir no console do navegador os números de 0 até 9, veja o exemplo abaixo:

script.js

```
01 console.log(0);
02 console.log(1);
03 console.log(2);
04 console.log(3);
05 console.log(4);
06 console.log(5);
07 console.log(6);
08 console.log(7);
09 console.log(8);
10 console.log(9);
```

Veja no exemplo, que utilizamos dez linhas de código para mostrar os números de 0 até 9, uma tarefa simples e fácil de fazer, mas imagine que você precise criar uma lógica mais complexa que será utilizada repetidamente, isso pode se tornar massante e chato, porém com a utilização de Laços de repetição é possível diminuir a quantidade de trabalho e de linhas consideravelmente, vejamos exemplos utilizando laços de repetição.

FOR

O laço de repetição **for** é provavelmente o mais utilizado dentro da linguagem JavaScript e dentro de muitas linguagens de programação. A estrutura deste laço de repetição é

```
for (contador; condição; incremento) {
    ...lógica...
}
```

Contador

O **contador** do laço de repetição **for** é uma variável que normalmente é iniciada com um valor a escolha do programador, dependendo do escopo do que é necessário que este laço de repetição realize.

Condição

A **condição** do laço de repetição **for** utiliza o contador previamente definido para criar uma condição booleana simples ou composta, podemos criar as condições para o **for** nos baseando no tópico de Decisões lógicas visto anteriormente.

Incremento

O **incremento** do laço de repetição **for** modifica o valor do **contador** predefinido até que o **contador** atinja um valor onde o retorno da **condição** seja falso.

Chamamos este último argumento de **incremento** , porém é possível incrementar ou decrementar o valor do **contador** , você lembra das operação que vimos no tópico de Operadores, então podemos utilizar de qualquer uma delas neste último argumento do **for** .

Lógica

O código será executado enquanto a **condição** do laço de repetição retornar verdadeiro, caso contrário o **loop** encerrará.

Vejamos o exemplo:

script.js

```
01 // for (contador; condição; incremento) {
02 for (var i = 0; i < 10; i++) {
03   console.log(i);
04 }
```

Na linha 2 do exemplo estamos declarando a variável **i = 0** que será o nosso **contador** , populado com o valor zero, ou seja o **contador** será iniciado com **0** .

Utilizamos o **contador** **i** em nossa **condição** **i < 10** , enquanto esta **condição** retornar verdadeiro, ou seja enquanto o valor do **contador** **i** for menor que **10** , tudo que estiver dentro do bloco de código, entre as chaves, entre as linhas 2 e 4, será executado.

Após o código, presente no bloco de código do laço de repetição **for,** ser executado, é executado o **incremento** **i++** , após realizado o incremento é realizado novamente a verificação sobre a **condição** **i < 10** , caso retorne verdadeiro, irá executar o código presente dentro do bloco de código do laço de repetição, esse processo se repetirá até que a **condição** retorne falso, até que o valor do **contador i** seja maior ou igual a **10** .

Com três linhas de códigos, utilizando o laço de repetição **for** , teremos o mesmo resultado que escrever dez linhas realizando o log dos número de 0 a 9.

Vejamos mais um exemplo ante de partir para o próximo laço de repetição:

script.js

```
01 var nomes = ['João', 'Maria', 'Aurora', 'Branca', 'Melevola'];
02
03 for (var i = nomes.length - 1; i >= 0; i--) {
04   console.log(nomes[i]);
05 }
```

Na linha 1 criamos um array chamado **nomes** com o nome de alguns personagens.

Na linha 3 criamos um laço de repetição, iniciando o **contador** com o valor da propriedade **length** do array **nomes** , criamos a **condição** que enquanto **i** for maior ou igual a 0 **,** será executado o bloco de código do laço de repetição, após executar o bloco de código do **for** , ao invés de **incrementar** , decrementamos o valor do **contador** com a operação **i--** , após realizar o decremento será novamente checado a **condição** definida, enquanto a **condição** não retornar falso, seguirá executando o bloco de código do laço de repetição **for** .

Seguindo a lógica do laço de repetição, neste exemplo iremos realizar o log do nome dos personagens, iniciando no último até chegar no primeiro nome.

Length

Assim como já vimos que um array ou objeto possuem métodos predefinidos, eles também possuem propriedades, como a propriedade **length** de um **array** que retorna o número de elementos dentro de um array, no caso a propriedade **length** do array **nomes** retornará 5, ou seja, o bloco de repetição do laço de repetição irá ser executado cinco vezes.

WHILE

Diferente do laço de repetição **for** , o laço de repetição **while** tem uma estrutura diferente onde não há um local definido para um **contador** ou **incremento** , vejamos:

```
while (condição) {
    ...lógica...
}
```

Condição

A **condição** do laço de repetição **while** não se limita a utilizar um **contador** , pois pode ser utilizado qualquer tipo de Decisão lógica, vista anteriormente, para a definição de sua **condição** , como a comparação de textos, comparação de valores booleanos ou qualquer outra.

Lógica

O código que será executado enquanto a **condição** do laço de repetição retornar verdadeiro, caso contrário o **loop** encerrará.

Vejamos o exemplo:

script.js

```
01 var nomes = ['Soluço', 'Astrid', 'Melequento', 'Perna de peixe', 'Bocão'];
02 var nomePesquisado = null;
03 var i = 0;
04 while (nomePesquisado == null && i < nomes.length) {
05   if (nomes[i] == 'Melequento') {
06     nomePesquisado = nomes[i];
07   } else {
08     console.log(nomes[i] + ' não é o nome pesquisado');
09   }
10   i++;
11 }
12 console.log('Nome encontrado ' + nomePesquisado);
```

Na linha 1 criamos um array com o nome de alguns personagens chamado **nomes** .

Na linha 2 declaramos uma variável chamada **nomePesquisado** com o valor **null** .

Para poder verificar todos os nomes do array, na linha 3 declaramos uma variável que iremos utilizar como contador para nosso laço de repetição.

Na linha 4 criamos um laço de repetição **while** com a **condição** composta de que enquanto o valor da variável **nomePesquisado** for igual a **null** e o valor do contador **i** for menor que a propriedade length do array **nomes** retornará verdadeiro, executando o trecho de código do laço de repetição.

Na linha 5 dentro da lógica do laço de repetição **while,** criamos uma decisão lógica para verificar se a posição atual do array **nomes** possuir um valor igual ao texto **Melequento** , será, na linha 6 atribuído a variável **nomePesquisado** o valor da posição atual do array **nomes** , então quando a for verificado a **condição** do laço de repetição, retornará falso e o **loop** encerrará, pois agora a variável **nomePesquisado** estará preenchida com o valor texto **Melequendo** .

Caso contrário, se a posição atual do array **nomes** não for igual ao texto **Melequento** , então será executado o log presente na linha 8, sendo assim, quando for verificado a **condição** do laço de repetição, será executado novamente a lógica do laço de repetição **while** , enquanto a **condição** retornar verdadeiro.

Quando o laço de repetição for encerrado, será executado a próxima linha, a linha 12, exibindo o nome encontrado no laço de repetição no console do navegador.

Break

A declaração **break** é uma palavra reservada, uma ferramenta por assim dizer, que nos ajuda a manipular o fluxo do laço de repetição o encerrando quando declaramos esta palavra reservada.

Imagine que a guarda costeira está a procura de um tubarão na costa de uma praia turística, vamos usar este cenário para criar um laço de repetição.

script.js

```
01 var animaisMarinhos = ['peixe', 'baleia', 'golfinho', 'tubarão', 'lula'];
02 for (var i = 0; i < animaisMarinhos.length; i++) {
03   if (animaisMarinhos[i] == 'tubarão') {
04    console.log(animaisMarinhos[i] + ' encontrado.');
05     break;
06   } else {
07     console.log('Procurar animal marinho.');
08   }
09 }
```

Veja o exemplo, na linha 1 criamos um array chamado **animaisMarinhos** com os animais encontrados pela guarda

costeira.

Na linha 2 criamos um laço de repetição **for** para verificar os animais encontrados, na linha 3 realizamos uma decisão lógica para verificar se o valor da posição atual do array possui o nome do animal a ser encontrado, um **tubarão** , caso seja um **tubarão** , na linha 4 está definido para exibir no console do navegador que o animal foi encontrado e na linha 5 encerra o laço de repetição com a declaração da palavra reservada **break** , caso contrário, exibirá no navegador uma mensagem no console do navegador para informar que a busca pelo animal ainda está em processo.

A declaração **break** pode ser utilizada em qualquer laço de repetição.

Continue

Assim como a declaração **break** a declaração **continue** é uma palavra reservada que nos ajuda a manipular o fluxo do laço de repetição, encerrando a verificação atual e passando para a próxima verificação quando declaramos esta palavra reservada.

Imagine que um botânico esteja contando o número de flores, porém ele está ignorando as flores de cor amarela.

script.js

```
01 var flores = ['rosa', 'azul', 'amarelo', 'rosa', 'amarelo', 'vermelho'];
02 var i = 0;
03 var numeroDeFlores = 0;
04 for (var i = 0; i < flores.length; i++) {
05   if (flores[i] == 'amarelo') {
06     continue;
07   }
08   numeroDeFlores++;
09 }
10 console.log('Numero de flores: ' + numeroDeFlores);
```

Veja o exemplo, na linha 1 estamos criando um array chamado **flores** com as flores do botânico.

Na linha 2 declaramos um contador para o número de posição do array **flores** .

Na linha 3 declaramos um contador para calcular o número de flores.

Na linha 4 criamos um laço de repetição **for** para verificar as flores.

Na linha 5 criamos uma decisão lógica para verificar se a flor atual é uma flor amarela, caso seja uma flor amarela, na linha 6 utilizamos a declaração **continue** para seguir para a próxima posição do array, ignorando o código abaixo a linha 6, dentro do bloco de código do laço de repetição, caso não seja uma flor amarela, irá seguir para a linha 8 e incrementar o contador de flores e então sim, passar para a próxima posição do array.

Assim como a declaração **break** a declaração **continue** pode ser utilizada em qualquer laço de repetição.

Comentários

Vimos vários exemplos no decorrer dos tópicos, em vários haviam certas linhas, ou trechos, de código que não afetaram o funcionamento nem a lógica do nosso código JavaScript, chamamos estes trechos de comentários.

Utilizamos comentários para explicar e documentar nosso código, tentar esclarecer o que a lógica que criamos deveria fazer e o que esperar de resultado no final da execução de nossa lógica. Pode parecer um tanto monótono e exaustivo, mas utilizar comentários é uma das formas mais simples de prevenir grandes desastres. Quando você mesmo ou algum colega de trabalho pegar seu código para realizar alguma manutenção ou ajuste no futuro, pense nos comentário, realmente, como parte da documentação do projeto, quanto mais, melhor será o entendimento.

Existem três tipos de comentários na linguagem JavaScript:

Comentário de linha

O comentário de linha é utilizado normalmente para descrever um pequeno trecho de código ou definir um certo fluxo no código e é iniciado com duas barras (**//**) como mostra no exemplo:

script.js

```
01 // Aqui será declarado uma variável
02 var a = 'valor';
03
```

Comentário de múltiplas linhas

O comentário de múltiplas linhas é normalmente utilizado para a documentação e descrição de um trecho mais complexo do código

e é iniciado com barra e asterisco (***/****) e fechado com asterisco e barra (****/***) como mostra no exemplo:

script.js

```
04 /*
05  Aqui estou descrevendo com
06  varios detalhes o que este
07  trecho de código deve executar
08 */
09 function codigoComplexo() {
10   console.log('Não existe nada mais complexo que isto!');
11 }
```

Comentário com compilação condicional

Este tipo de comentário é utilizado pelo navegador Internet Explorer para uso de novos recursos da linguagem JavaScript sem sacrificar a compatibilidade com versões antigas, como as versões 4 a 8 do internet explorer, que não suportam certos recursos.

A compilação condicional é utilizada dentro de comentários, pois caso um site seja aberto em um navegador antigo sem suporte de compilação condicional, esses comentários serão simplesmente desconsiderados.

A compilação condicional é ativada usando a declaração **@cc_on** ou usando **@if** ou **@set** , veja mais no site da Microsoft:

script.js

```
13 /*@cc_on @*/
14 /*@if (@_jscript_version >= 4)
15     console.log("Versão JavaScript 4 ou superior");
16     @else @*/
17     console.log("compilação condicional não suportada por este mecanismo de script.");
18 /*@end @*/
```

Para mais informações sobre este assunto acesse o link http://bit.ly/Compilacao_Condicional

ECMAScript

Vamos ver agora, brevemente, uma das coisas que todo o desenvolvedor JavaScript deveria saber, as versões do JavaScript e um pouco sobre elas.

LiveScript

Em 1996 a linguagem LiveScript teve seu nome alterado para JavaScript, esta alteração nada mais foi que uma jogada de Marketing para atrair os desenvolvedor Java, umas das linguagens em alta na época. Sendo assim podemos concluir que a linguagem Java nada tem haver com a linguagem JavaScript, tirando o nome.

ECMAScript 1

Em 1997 o ECMAScript 1 tornou-se a primeira versão da linguagem JavaScript.

Chamamos de ECMAScript a padronização da linguagem JavaScript e chamamos de JavaScript a linguagem em prática.

ECMAScript 5 (ES5)

Em 2009 a padronização ECMAScript 5, a versão que utilizamos neste livro, também chamada de ES5, foi liberada trazendo incontáveis novas atualizações e novas características para a linguagem JavaScript.

Demorou anos para que os navegadores conseguissem se adaptar e assimilar todas as mudanças liberadas na versão ES5,

felizmente, hoje em dia podemos utilizar todas essas ferramentas desta versão sem nenhuma restrição.

ECMAScript 2015 (ES2015 / ES6)

Em 2015 a padronização ECMAScript 2015, a versão mais esperada e a versão com mais atualizações de todos os tempos, foi liberada, trazendo centenas de novas funcionalidades para a linguagem JavaScript.

Houve a pretensão de nomear está versão como ECMAScript 6, porém como de 2015 em diante houve novas atualizações com pequenas melhorias, definiu-se o nome como ECMAScript 2015 ou ES2015, entretanto você pode encontrar o termo ES6 em alguns lugares.

ECMAScript 2016 (ES2016 / ES7)

Em 2016 a padronização ECMAScript 2016 foi liberada com pequenas melhorias, como o operador de exponenciação.

ECMAScript 2017 (ES2017 / ES8)

Em 2017 a padronização ECMAScript 2017 foi liberada com algumas melhorias, como a inclusão de **async** e **await** .

ES.Next

Você pode ver em alguns lugar o termo ES.Next, este é o termo que estão dando para a próxima versão do ECMAScript que ainda não foi finalizada.

A tendência é que todo ano, novas atualizações sejam feitas com pequenas melhorias, para deixar a linguagem cada vez mais

robusta e bem estruturada.

Fechamento

Tudo o que nós vimos nesta primeira parte do livro, chamamos de ECMAScript 5, umas das versões da linguagem JavaScript, como visto no tópico anterior.

Agora que aprendemos os fundamentos da linguagem JavaScript, como laços de repetição, operadores, funções, decisões lógicas, todas estas funcionalidades, se utilizadas juntas, com clareza e bom entendimento acabam nos poupam muito trabalho, e código, como podemos ver em nossos exemplos, além de poder nos ajudar a criar códigos mais claros e complexos ao mesmo tempo que o deixamos entendíveis e organizados.

Eu aconselho a você, utilizar as informações obtidas até agora para praticar e criar seus próprios códigos, treinar as ferramentas e os conhecimentos que aprendeu até agora.

Ainda não temos um total entendimento de como a linguagem JavaScript funciona, então vamos seguir com nosso conteúdo e ver como o JavaScript realmente trabalha por debaixo dos panos.

Como o JavaScript funciona

Estamos agora entrando no tópico que iremos entender de fato como a linguagem JavaScript funciona, onde vamos aprimorar nossas habilidades como desenvolvedores JavaScript ao entender como as linhas desta linguagem são executadas, algo que você realmente vai querer aprender, pois você sabendo como a linguagem realmente funciona, salvará muito tempo de pesquisa sobre como resolver muitos problemas do seu código.

Bugs (Problemas)

A tradução literal de *bugs* do inglês para o português é *insetos* , porém esta terminologia é utilizada para referenciar erros e problemas em códigos, ou seja, caso você ouvir alguém falando sobre *bugs* em um código, esta pessoa estará falando sobre problemas ou erros de um código, independente da linguagem de programação que seja utilizada.

Interpretador, Motor e Analisador

A linguagem JavaScript sempre é hospedada em algum ambiente, normalmente um navegador, como Internet Explorer, Google Chrome, Safari, entre outros, é onde o código JavaScript será executado.

Nosso conteúdo é focado em JavaScript para navegadores, então não iremos nos aprofundar sobre outros ambientes onde é possível executar códigos JavaScript, como Node.JS ou Aplicativos Híbridos, que também utilizam a linguagem JavaScript.

Quando escrevemos nosso código JavaScript e queremos o executar no navegador, acontecem várias coisas por debaixo do panos até que tenhamos nosso resultado de fato.

Quando rodamos nosso código dentro de um navegador, existe um programa, em termos simples, que realiza a leitura de nosso código JavaScript e o executa. Este programa chamada-se Interpretador JavaScript ou Motor JavaScript, em inglês *JavaScript Engine* . Existem diversos motores JavaScript, como Google's V8 Engine, SpiderMonkey, JavaScript Core, entre outros, confira a lista completa atraves do link http://bit.ly/2sn40yK .

A primeira tarefa que é executada dentro de um Motor JavaScript é analisar nosso código, essa análise é feita por um analisador, em inglês *Parser* , que lê linha por linha do nosso código e verifica se todas as linhas estão corretas, ou seja, este analisador conhece muito bem a linguagem JavaScript e se houver algum erro em nosso código ele irá retornar uma mensagem de erro no console do navegador.

Caso não haja nenhum erro em nosso código, o Analisador produzirá uma estrutura de dados, conhecida como Árvore de Sintaxe Abstrata, em inglês *Abstract Syntax Tree* , e então transformar esta estrutura de dados em um conjunto de instruções,

passos, que o processador do computador possa interpretar, apenas então nosso código será executado.

Nós não precisamos memorizar todos estes passos, porém é importante entender como um interpretador/motor JavaScript funciona.

Árvore de Sintaxe Abstrata (Abstract Syntax Tree)

Quando se lê árvore de sintaxe, imagine realmente um árvore, onde cada ramificação de cada galho se torna dois ou três galhos novos, cada uma dessas ramificações pode ser a declaração de uma variável ou uma decisão lógica, tenha isso em mente quando ler sobre árvore de sintaxe abstrata.

Contexto de Execução

Neste tópico vamos ver a ordem em que cada código é executado dentro da linguagem JavaScript.

Todo código JavaScript precisa ser executado dentro de um ambiente, chamamos estes ambientes de contextos de execução, vamos colocar em termos mais palpáveis.

Imagine que um contexto de execução é semelhante a uma caixa que armazena nossas variáveis , dentro dessa caixa nossas variáveis e códigos são validados, tratados e executados.

Agora você já tem em sua mente que um contexto de execução é como uma caixa, agora imagine que esta caixa é um objeto da linguagem JavaScript, ou seja tudo o que for declarado dentro deste objeto será atribuído ao objeto caixa como um propriedade.

Também é importante mencionar que toda vez que uma função é executada, é criado um novo contexto de execução para a mesma.

Quando falamos sobre contexto de execução, temos que entender como ele funciona de fato, assim vamos ver a estrutura e as fases de um contexto de execução.

Estrutura

Como mencionado, podemos associar um contexto de execução a um objeto e este objeto tem três propriedades, que são **Objeto de Variável (OV)** , em inglês *Variable Object (VO)* , **Cadeia de Escopo** , em inglês *Scope Chain* e a variável **this** , que já tivemos a oportunidade de ver um pouco sobre ela no tópico de objetos.

Objeto de Variável (OV)

Esta propriedade tem os argumentos da função, as declarações e definições de variáveis e funções.

Cadeia de Escopo

Esta propriedade tem todas as variáveis locais, assim como todas as variáveis de todos os seus pais.

O pai é o contexto que chama certa função, está certa função chamada, tem como pai o contexto que a chamou.

this

Esta propriedade é a variável **this** , o objeto relacionado com o contexto de execução atual.

1 - Fase de Criação

Agora que sabemos quais as propriedades que um contexto de execução possui, vamos ver como essas propriedades são criadas e definidas de fato.

Quando uma função é chamada, é aberto um novo processo de criação para um novo contexto de execução, vamos ver como as propriedades são criadas.

Após criar o Objeto de Variável é criado o Objeto de Argumento, em inglês *Argument Objeto* , que armazena todos os argumentos passados para a função na hora que ela é chamada.

Após criado o Objeto de Argumento, o código da função é escaneado, procurando por declarações de funções, para cada declaração de função encontrada é criado uma propriedade no Objeto de Variável com um ponteiro referenciando a função encontrada, isso significa que todas as funções deste contexto são salvas dentro do Objeto Variável, mesmo antes da fase de execução ser iniciada.

Após criado as propriedades para as declarações de funções, o código da função é escaneado, procurando por declarações de variáveis, para cada declaração de variável encontrada é criado uma propriedade no Objeto Variável para esta variável, com o valor **undefined** .

Após estes processos é finalmente criado a Cadeia de Escopo e a variável **this** .

Içamento (Hoisting)

Um conceito da fase de criação, geralmente chamado de Içamento, em inglês *Hoisting* , é um termo referente a criação das propriedades do Objeto Variável relacionado às declarações de funções e variáveis, este é um termo que confunde muito novos programadores JavaScript, porém vamos entendê-los agora.

Declarações de Funções

Para funções o Içamento escaneia as declarações de funções e cria um propriedade com um valor definido, o valor é um ponteiro, apontando para a função a ser chamada.

Declarações de Variáveis

Para variáveis o Içamento escaneia as declarações de variáveis e cria um propriedade com um valor não definido, **undefined** , o valor das variáveis será definido apenas na fase de execução.

2 - Fase de Execução

Na fase de execução o contexto de execução já está criado e as propriedades já estão definidas no Objeto de variável, porém

apenas agora na fase de execução que as propriedades são preenchidas com seus respectivos valores, após a execução do código linha por linha. Vamos entender com exemplos como funciona o içamento de funções e variáveis.

Funções

script.js

```
01
02 exibirTexto();
03
04 function exibirTexto() {
05   console.log("Fase de execução, declaração de função");
06 }
07
08 exibirTexto();
09
10 // exibirOutroTexto();
11
12 var exibirOutroTexto = function () {
13   console.log("Fase de execução, expressão de função");
14 };
15
16 exibirOutroTexto();
```

No exemplo temos uma declaração de função na linha 4 e uma expressão de função na linha 12, nas linhas 2 e 8 chamamos a função **exibirTexto** e nas linhas 10 e 16 chamamos a função **exibirOutroTexto** , vejamos o resultado.

Podemos ver que a mensagem da função **exibirTexto** apareceu no console do navegador, mas ocorreu um erro na linha 10 de nosso script informando que **exibirOutroTexto** não é uma função, isso ocorreu porque **exibirOutroTexto** na verdade é uma variável que está recebendo o valor de uma função anônima na linha 12. Como mencionado anteriormente, variáveis são pré atribuídas com o valor do tipo de dado **undefined** na fase de criação e apenas na fase de execução é que, após nosso código ser lido linha por linha, cada variável terá seu valor atribuído, consequentemente uma expressão de função apenas pode ser

consumida em linhas após a definição do valor da variável, como nosso código tentou chamar o **exibirOutroTexto** antes de o mesmo ter recebido o valor da função anônima, acabou gerando um erro, pois o tipo de dado **undefined** não é uma função.

Experimente remover ou comentar a linha 10 de seu código.

Variáveis

script.js

```
01
02 var minhaIdade;
03
04 console.log(minhaIdade);
05
06 minhaIdade = 42;
07
08 console.log(minhaIdade);
```

No exemplo temos uma definição de variável **minhaIdade** na linha 2, exibimos o valor da variável no console do navegador na linha 4, atribuímos o valor numérico 42 a variável **minhaIdade** na linha 6 e na linha 8 exibimos novamente o valor da variável no console do navegador, vejamos o resultado.

Podemos ver que no console do navegador é exibido primeiramente **undefined** e em seguida 42, isso ocorre, pois na fase de criação do contexto de execução é criado uma propriedade para cada variável com o valor **undefined** e na fase de execução é que ocorre a atribuição de um valor para as variáveis, após nosso código ser lido linha por linha, como a primeira vez que tentamos exibir o valor da variável no console do navegador, em nosso exemplo, ainda não havíamos atribuído um valor para a variável, foi exibido no console do navegador o valor **undefined** , como esperado.

Contexto de Execução Global

Vamos falar sobre o contexto de execução global, este contexto está acessível a todos os demais contextos, ou seja tudo o que for declarado, sejam variáveis ou funções, em seu código fora de qualquer função é atribuído e executado no contexto global.

Nos navegadores o contexto global é um objeto global denominado **window** , então toda variável ou função declarada no contexto global será automaticamente atribuída ao objeto **window** .

script.js

```
01 var primeiraVariavel = "Sou um valor inicial";
02
03 console.log(window.primeiraVariavel);
04
```

Veja o exemplo, na linha 1 definimos uma variável chamada **primeiraVariavel** e atribuímos a ela um valor do tipo texto, em seguida na linha 3 realizamos um log no console do navegador da propriedade **primeiraVariavel** do objeto **window** .

Este exemplo mostra na prática como funciona a declaração de uma variável no contexto global, sempre que algo for declarado fora de qualquer função, ou seja no contexto global de execução, será automaticamente atribuído ao objeto **window** .

Contexto de Execução Local

Toda a vez que uma função é executada, um novo contexto de execução é criado para esta função, as variáveis definidas neste novo contexto não serão atribuídas ao objeto global **window** , pois serão declaradas em um novo contexto de execução, que não é o global, consequentemente um contexto local.

Bind, Call e Apply

Agora que sabemos como um contexto de execução funciona e como se compõe sua estrutura é importante sabermos que funções, assim como arrays, possuem métodos próprios. Em específico, funções possuem três métodos muito interessantes que são **Bind** , **Call** e **Apply** que nos possibilitam escolher, definir, qual será o **this** do contexto das funções que chamarmos, parece um pouco complexo, mas vejamos alguns exemplos para entender melhor, veja:

script.js

```
01 var  carro = {
02     nome: 'Fusca',
03     marca: 'VW',
04     mostrarDados : function (cor, ano) {
05         console.log(this.nome,ano, this.marca, cor);
06     }
07 }
08
09 carro. mostrarDados ('Azul', 1970);
```

Veja que na linha 1 declaramos uma variável, chamada **carro** , recebendo um objeto como valor, este objeto possui um método chamado **mostrarDados** esperando dois parâmetros **cor** e **ano** e a lógica deste método será apresentar no console de nosso navegador as propriedades **nome** , **marca** do objeto a que pertence juntamente com os parâmetros **cor** e **ano** .

Na linha 9 realizamos a chamada do método **mostrarDados** do objeto **carro** , você verá no console do navegador isto:

Console

```
Fusca 1970 VW Azul
```

Agora que temos um exemplo de objeto, vejamos como os métodos **call** , **bind** e **apply** se comportam.

Call

O método **call** consiste em realizarmos uma chamada para uma função, porém passando como primeiro parâmetro o que deverá ser o **this** para esta função ou método e em seguida os demais parâmetros esperados pelo método, função, chamada.

script.js
```
11 // Call
12 carro .mostrarDados. call ({
13     nome: 'Opala',
14     marca: 'Chevrolet'
15 }, 'Bordo', 1972);
```

Veja que na linha 12 realizamos a chamada do método **mostrarDados** utilizando o método **call** e como esperado passando um objeto como primeiro parâmetro para ser o **this** do contexto, em seguida os parâmetros esperados pelo método **mostrarDados** , você verá no console do navegador isto:

Console
```
Opala 1972 Chevrolet Bordo
```

Bind

O método **bind** é muito similar ao método **call** , porém o método **bind** não executa a função de imediato, quando o método **bind** é chamado ele cria uma cópia de um método ou função com o **this** deste alterado, nós pré definimos o **this** de forma explícita para esta cópia.

Imagine que você precise mostrar na tela vários carros da mesma cor, porém de anos diferentes, para cada um deles você poderia utilizar o método **call** , porém seria um trabalho desnecessário, pois utilizando o método **bind** podemos criar uma cópia do método **mostrarDados** , pré setando o **this** e o valor do primeiro argumento **cor** , vejamos:

script.js

```
18 // Bind
19 var brasiliaAmarela = carro.mostrarDados.bind({
20     nome: 'Brasilia',
21     marca: 'VW'
22 }, 'Amarelo');
23
24 brasiliaAmarela(1974);
25 brasiliaAmarela(1975);
26 brasiliaAmarela(1982);
```

No exemplo, na linha 19 declaramos uma variável chamada **brasiliaAmarela** e atribuimos a ela um cópia do método **mostrarDados** do objeto **carro** , modificando o **this** da função e pré setando o argumento **cor** , em seguida nas linhas 24, 25 e 26 realizamos a chamada da nova função, veja o resultado:

Console

```
Brasilia 1974 VW Amarelo
Brasilia 1975 VW Amarelo
Brasilia 1982 VW Amarelo
```

Adoção (Carrying)

Existe um conceito chamado adoção, em inglês *carrying* **,** que consiste em criar uma função, baseada em uma outra função e o método **bind** é extremamente eficaz na aplicação deste conceito.

Apply

O método **apply** consiste em realizarmos uma chamada para uma função ou método, porém, assim como o método **call** , passando como primeiro parâmetro o que deverá ser o **this** para esta função ou método, em seguida os demais parâmetros, em formato de **array** , esperados pelo método, função, chamado.

script.js

```
28  // Apply
29 carro .mostrarDados. apply ({
```

```
30      nome: 'Onix',
31      marca: 'Chevrolet'
32 }, ['Cinza', 2016]};
```

Veja que na linha 29 realizamos a chamada do método **mostrarDados** utilizando o método **apply** e como esperado passando um objeto como primeiro parâmetro para ser o **this** do contexto, em seguida os parâmetros esperados pelo método **mostrarDados** em formato de **array** , você verá no console do navegador isto:

Console
Onix 2016 Chevrolet Cinza

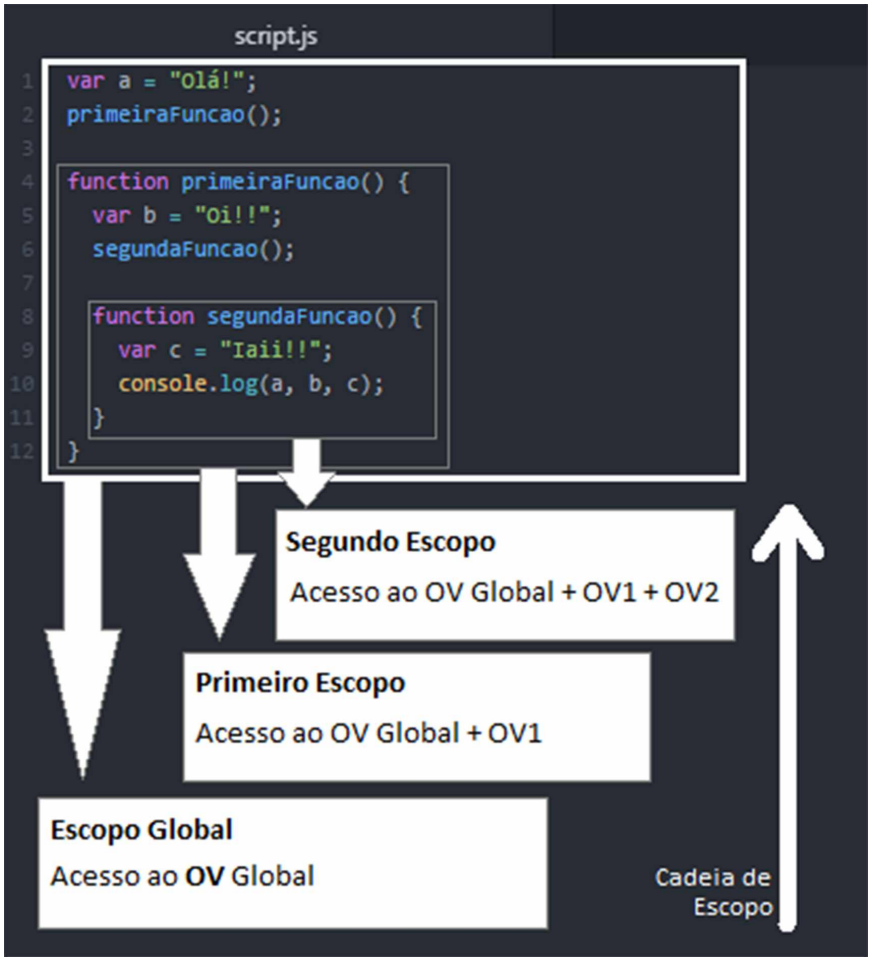
Cadeia de Escopo

Comentamos no tópico sobre Contexto de Execução que a cadeia de escopo é a propriedade que possui todas as variáveis locais, assim como todas as variáveis de todos os seus pais, porém não foi explicado como funciona a cadeia de escopo de fato, então vamos a entender.

Existe uma pergunta que podemos usar para nos ajudar a explicar o conceito de Escopo: *Onde podemos acessar uma determinada variável?*

Cada nova função cria um novo escopo, espaço, ambiente, no qual as variáveis definidas são acessíveis.

Há um conceito chamado escopo léxico, em inglês *Lexical Scoping* , que define: Se uma função é léxica dentro de outra função, esta função obtém acesso ao escopo da função externa que irá a executar, em outras palavras uma função filha tem acesso ao escopo do pai, mas uma função pai não tem acesso ao escopo dos filhos, isto é chamado de cadeia de escopo.



Vemos na imagem que há três pontos de definições de variáveis nas linhas 1, 5 e 9, cada variável sendo declarada em um escopo diferente.

Nas caixas em branco podemos ver a descrição de cada escopo e a quais OV (objeto de variável), os mesmos possuem acesso. O escopo global possui acesso apenas acesso ao escopo global, já o primeiro escopo, o da **primeiraFuncao** possui acesso aos OV Global e OV1 e o segundo escopo, o da **segundaFuncao** , possui acesso aos OV Global, 1 e 2.

Na linha 10 dentro da função **segundaFuncao** mostramos no console do navegador o valor das variáveis a, b e c, porém como o código faz para descobrir em que escopo cada variável está.

A lógica JavaScript tenta achar a variável dentro do objeto de variável do escopo atual, caso ela não encontre, irá procurar esta variável no objeto de variável do escopo pai, caso também não encontre no escopo pai, seguirá este processo até achar a variável

ou caso não seja encontrado a variável dentro de nenhum dos escopos visíveis, léxicos, um erro será mostrado no navegador, informando que a variável não foi definida.

Assim como falamos antes e podemos ver no exemplo, um pai não tem acesso ao escopo dos filhos, mas um filho tem acesso ao escopo do pai.

Pilha de Execução

A linguagem JavaScript é uma linguagem de encadeamento único, em inglês *single threaded* , o que significa que apenas uma tarefa será executada por vez. Quando o interpretador JavaScript executa inicialmente o código, ele primeiro entra no contexto de execução global por padrão, como já tivemos a oportunidade de ver.

Agora cada invocação de uma função a partir desse ponto resultará na criação de um novo contexto de execução e este novo contexto de execução é colocado no topo da pilha de execução, após terminar de executar o bloco de código desta função o contexto de execução desta função é removido do topo da pilha de execução.

Algo importante que devemos entender é que pilha de execução é referente a ordem em que cada função é executada e cadeia de escopo é relacionado a forma que o código é lido pelo interpretador, ou seja se alterarmos nosso código, mas fizermos as mesmas chamadas para nossas funções a pilha de execução não irá ser alterada, mas a cadeia de escopo pode mudar, caso realizemos alguma alteração significativa em nosso código.

Cadeia de Escopo != Contexto de Execução

Algo importante a esclarecer é que contexto e escopo não é a mesma coisa. Muitos desenvolvedores ao longo dos anos, muitas vezes confundem os dois termos, eu mesmo já cometi este deslize, descrevendo incorretamente os termos. Para ser justo, a terminologia ao longo dos anos se tornou bastante confusa, mas vamos entender.

Cada chamada de função tem um escopo e um contexto associados a ela, como já pudemos ver nos outros tópicos, mas não são o mesmo, o escopo é baseado em função, enquanto o contexto é baseado em objeto.

Em outras palavras, o escopo pertence ao acesso variável de uma função quando ela é chamada e é exclusiva para cada chamada.

Contexto é sempre o valor da palavra-chave **this** , que é uma referência ao objeto que armazena o código atualmente em execução.

this

A palavra-chave **this** é a variável que cada e todo contexto de execução tem e como já podemos ver, por ser uma variável, tem seu valor preenchido apenas na fase de execução.

Lembra-se que no tópico sobre objetos onde vimos o conceito da palavra **this** , então em funções a palavra reservada **this** têm um comportamento similar.

Chamada Regular de função

Em uma chamada normal de função a variável **this** aponto para o objeto global **window** , que já tivemos a oportunidade de conhecer.

script.js

```
01 function funcao() {
02   return this;
03 }
04
05 // Chamada de função regular
06 console.log(funcao());
```

Veja que criamos na linha 1 uma função e retornamos na linha 2 o valor da variável **this** desta função. Na linha 6 exibimos no console do navegador o valor da variável **this** retornada.

Como esperado, podemos ver no console do navegador que o valor da variável **this** é o mesmo valor da variável global **window** .

Chamada de método e Nova Instância

Em uma chamada de método, que consiste em uma função ter seu valor atribuido a uma propriedade de um objeto, o **this** irá apontar para o objeto em que esta função pertence. Um princípio

muito similar se aplica ao executar uma função com o operador **new** para criar uma nova instância desta função. Quando chamado dessa maneira, o valor **this** dentro do escopo da função será definido para a instância recém-criada.

script.js

```
01 var objeto = {
02   retorno: function () {
03     return this;
04   }
05 };
06
07 // Chamada de método
08 console.log(objeto.retorno());
09
10 // Chamada de função/método em uma nova instância
11 console.log(new objeto.retorno());
```

Veja em nosso exemplo, na linha 1, onde criamos um objeto chamado **objeto** com uma propriedade denominada **retorno** recebendo como valor uma função retornando o **this** da função.

Na linha 8 exibimos o valor da variável **this** retornada da chamada de método e na linha 11 exibimos o valor da variável **this** retornada da nova instância da função retorno.

Veja que no console do navegador os valores são diferente, pois realmente são coisas diferentes, quando utilizamos a palavra **new** estamos assumindo que realmente queremos criar uma nova instância, modificando assim o valor da variável **this** dentro da função.

Quando falamos sobre o valor de **this** , tudo sempre vai depender de como chamamos uma função.

Empréstimo de método

Uma prática muito utilizada na linguagem JavaScript é um conceito chamado Empréstimo de método que consiste em atribuir o valor da sintaxe, estrutura gráfica, de um método a propriedade de outro método.

script.js

```
01 var objetoGato = {
02   alimento: "ração",
03   comer: function () {
04     console.log("Estou comendo " + this.alimento);
05   }
06 }
07
08 var objetoLeao = {
09   alimento: "carne"
10 };
11 objetoLeao.comer = objetoGato.comer;
12
13 objetoLeao.comer();
```

console

```
Estou comendo carne
```

Veja que em nosso exemplo criamos dois objetos, **objetoGato** na linha 1 e **objetoLeao** na linha 8, ambos objeto possuem a propriedade **alimento** mas apenas um possui o método **comer** , porém ambos objetos necessitam desta função, então na linha 11 emprestamos a sintaxe do método comer do **objetoGato** para criarmos o método **comer** para o **objetoLeao** .

Fechamento

Até agora já vimos os conceitos sobre a linguagem JavaScript, assim como podemos melhor organizar nosso código, também como a linguagem funciona por debaixo dos panos e agora sabemos realmente o que acontecerá com nosso código e como será interpretado pelos navegadores antes de que seja executado de fato.

Certo agora sabemos como programar em JavaScript, porém ainda podemos aprofundar nossos conhecimentos sobre a linguagem e deixar nossos códigos ainda mais robustos e complexos, então vamos para a próxima parte do livro e entender a fundo alguns conceitos muito importante s sobre a linguagem JavaScript.

JavaScript Avançado

Neste tópico vamos entrar a fundo e falar sobre funções e objetos, os aspectos mais utilizados da linguagem JavaScript, novos desenvolvedores JavaScript tem dificuldade de entender este tópico e todo o seu potencial, mas não se preocupe, pois agora iremos esclarecer como tudo funciona.

Ob jetos e Funções

Na linguagem JavaScript, na maior parte, tudo é um objeto, mas não tudo, pois como vimos nos tópicos iniciais do livro, também existem dados primitivos. Este é um dos fatores que transforma a linguagem JavaScript única e a distingue de várias outras linguagem de programação.

Orientação a Objetos

No decorrer do conteúdo podemos ver como objetos são importantes para a linguagem JavaScript e no começo do livro tivemos uma pequena explicação sobre o que é uma linguagem orientada a objetos, porém agora que passamos por conceitos muito interessantes até aqui, acredito que seja interessante voltar ao assunto.

Em termos muito simples, Programação Orientada a Objetos faz um uso pesado de Objetos, Propriedades e Métodos, estes objetos interagem um com os outros, para criar aplicações mais complexas. Usamos objetos para armazenar dados, estruturar nosso código e manter nosso código limpo e organizado.

Até agora criamos apenas objetos, um por um, para armazenar dados simples e sem graça, mas há uma maneira melhor de fazermos isto. Imagine algo como um modelo e com esse modelo podemos criar quantos objetos quisermos.

Vamos criar mais para frente uma função denominada **Carro** e vamos utilizar esta função como modelo para criar vários objetos de carros.

Em outras linguagens de programação, muitas vezes este modelo é chamado de classe, entretanto na linguagem JavaScript costumamos chamar isto de construtor, em inglês *Constructor* , e

baseado nesse construtor podemos criar quantos objetos desejarmos.

Quando utilizamos um construtor como base para a criação de um objeto, este objeto criado se transforma em uma instância do construtor utilizado. Todas as instância criadas a partir de um construtor terão acesso a todas as informações do construtor, como por exemplo métodos, vamos entender melhor.

Prototipo (Prototype)

Herança é o termo utilizado que nos possibilita acessar as informações do construtor, como por exemplo métodos e propriedades. A linguagem JavaScript é uma linguagem orientada a objetos, porém é baseada em protótipos, o que significa que a herança nesta linguagem funciona utilizando algo chamado **prototype** . Na prática todo e cada objeto na linguagem JavaScript possui uma propriedade chamada **prototype** que torna possível a utilização de herança na linguagem JavaScript.

Herança (Inheritance)

Em termos simples herança é quando criamos um objeto baseado em outro objeto e o objeto criado tem acesso às propriedades e métodos de outro objeto, como explicado antes.

Imagine que precisemos criar um objeto de jogador, podemos considerar que um jogador é um pessoa certo, então para criar um objeto de jogador precisaríamos criar as mesmas propriedades de uma pessoa o que iria aumentar nosso trabalho, ao invés disto, podemos criar um objeto **jogador** herdando os métodos e propriedades do objeto **pessoa** , desta forma o jogador terá acesso às suas propriedades assim como as propriedades e métodos do objeto **pessoa** .

Todos os métodos e propriedades que desejamos compartilhar, com as instâncias do **construtor** que utilizarmos, devem estar

dentro o objeto **prototype** do **construtor** , assim possibilitando as instâncias herdarem estes métodos e propriedades.

Estamos vendo sobre colocar as funções a serem herdadas no **prototype** do **construtor** , fazemos isto, pois este objeto **prototype** será o mesmo em toda, e cada, instância criada a partir do **construtor** , então por tabela se adicionarmos um método ou propriedade no **prototype** do **construtor** , toda instância deste **contrutor** terá estes métodos e propriedades em seu objeto **prototype** também.

Antes de continuarmos é importante saber que todo e qualquer objeto que criarmos terá como construtor o objeto **Object** , sei que isso parece um tanto confuso, mas realmente é isso que acontece e isso significa que todo e qualquer objeto criado, terá em seu **prototype** as propriedades e métodos herdados do **prototype** do objeto **Object** pois todo e qualquer objeto será uma instância do construtor **Object** .

Cadeia de Prototipo (Prototype Chain)

Falamos sobre herança e protótipo, mas como funciona a cadeia de protótipo, como os métodos e propriedades são encontrados entre as instâncias e seus construtores.

No tópico de cadeia de escopo vimos como é realizado a busca de variáveis entre os diferentes níveis da cadeia de escopo, então, a cadeia de protótipo funciona de forma similar, caso em nosso código tentemos utilizar uma propriedade ou um método, o comportamento padrão será tentar encontrar estes nas propriedades do objeto atual, após no **prototype** do objeto atual, caso não seja encontrado nestes níveis os passos seguintes serão tentar encontrar estes métodos e propriedades no **prototype** do construtor do objeto, caso não encontre novamente, o processo se repetirá até chegar ao **prototype** do objeto **Object** e caso também não seja encontrado neste último nível, então será retornado **undefined** .

Agora que vimos alguns conceitos sobre Objetos, e temos ciência, de como funciona a orientação a objetos, herança e cadeia de escopo na linguagem JavaScript, vamos entrar mais a fundo e entender de forma prática estes conceitos.

Construtor de função (Function Constructor)

Até agora estávamos criando objetos de forma literal

```
var  objeto = {
        propriedade : valor,
        propriedade : valor,
        propriedade: valor
};
```

Porém se quisermos criar centenas de objetos com as mesmas propriedades do objeto que criamos, usar a forma padrão seria trabalhoso, entretanto existem diferentes formas de criar um objeto na linguagem JavaScript e provavelmente a forma mais comum é utilizando algo chamado construtor de função, em inglês *function constructor* .

Por padrão quando criamos um construtor de função utilizamos a primeira letra do nome do construtor em maiúsculo, para identificar que é um **construtor** .

script.js

```
01 // Primeira letra em maiúsculo para identificar que é um construtor
02 var Carro = function (nome, cor, marca, ano) {
03   this.nome = nome;
04   this.cor = cor;
05        this.marca = marca;
06        this.ano = ano;
07 };
08
09 var fusca = new Carro('Fusca', 'Azul', 'VW', 1975);
10
11 console.log(fusca);
```

Veja o exemplo, na linha 2 criamos uma função, com a primeira letra em maiúsculo para identificar que a função é um

construtor, na linha 9 utilizamos o operador **new** para criar uma nova instância do construtor **Carro** e atribuir o valor dessa nova instância a variável **fusca** e na linha 11 mostramos o valor do objeto **fusca** no console do navegador.

Experimente criar novas variáveis, instâncias, utilizando o construtor **Carro** e mostrar no console o valor dessas para ver o que aparece, no link de repositório há um exemplo disto http://bit.ly/2MeCT29

Operador New

Para entendermos melhor o último exemplo, vamos entender o que o operador **new** faz realmente.

Toda a vez que utilizamos o operador **new** um novo objeto vazio é criado, agora temos um objeto vazio criado, após isto o construtor invocado é chamado com os argumentos que especificamos para ele através dos parâmetros. Como um construtor é uma função e estamos realizando uma chamada para esta função, significa, como já podemos ver em tópicos anteriores, que um novo contexto de execução será criado, junto com ele a variável **this** , em uma chamada de função comum o valor da variável **this** seria igual ao valor da variável global **window** , porém como estamos utilizando o operador **new** estamos criando uma nova instância que retornará a princípio um objeto vazio, por seu um objeto o valor da variável **this** refere-se ao próprio objeto chamado, neste caso a nova instância do construtor **Carro** , por este motivo conseguimos utilizar no último exemplo a variável **this** com notação de ponto para criar uma nova propriedade para o nosso objeto vazio, que agora possui as propriedades **nome** , **cor** , **marca** e **ano** .

Herança na Prática

Vimos no tópico de Objetos como funciona Herança na linguagem JavaScript, porém não vimos um exemplo prático ainda.

Então imagine que precisamos calcular se o modelo do carro/ano do carro se enquadra dentro da faixa de isenção de IPVA, para isso precisamos saber se o carro foi fabricado a mais de trinta anos. Para poder realizar esse cálculo deveríamos criar um método para cada objeto carro, porém como estamos utilizando um construtor, podemos simplesmente criar este método dentro do nosso construtor, entretanto se seguirmos ir por este caminho, não estaremos utilizando o conceito de herança, pois estaremos criando para cada nova instância um método que irá executar exatamente a mesma função. Lembra-se o que vimos no tópico de Objetos, sobre os métodos e propriedades a serem herdados deveriam estar dentro o **protótipo** (**prototype**) do **objeto** e é exatamente isso que precisamos fazer.

script.js

```
01 // Primeira letra em maiúsculo para identificar que é um construtor
02 var Carro = function (nome, cor, marca, ano) {
03   this.nome = nome;
04   this.cor = cor;
05         this.marca = marca;
06         this.ano = ano;
07 };
08
09 Carro.prototype. calcularIsencaoIVPA  = function () {
10   var anoAtual = new Date().getFullYear() - this.ano;
11   if (anoAtual >= 30) {
12     console.log('O carro ' + this.nome + ' possui isenção de IPVA.');
13  } else {
14     console.log('O carro ' + this.nome + ' não possui isenção de IPVA.');
15   }
16 };
17
18 var fusca = new Carro('Fusca', 'Azul', 'VW', 1975);
19 var onix = new Carro('Onix', 'Bege', 'Chevrolet', 2014);
20 var opala = new Carro('Opala', 'Bordo', 'Chevrolet', 1973);
21
22 fusca. calcularIsencaoIVPA ();
23 onix. calcularIsencaoIVPA ();
24 opala. calcularIsencaoIVPA ();
```

Utilizemos o nosso último exemplo, veja que temos o mesmo construtor **Carro** definido na linha 2, na linha 9 adicionamos ao

prototype de nosso construtor um método para calcular se o carro está ou não dentro da faixa de isenção de IPVA, criamos algumas instâncias do nosso construtor nas linhas 18,19 e 20 e nas linhas 22, 23 e 24 executamos o método **calcularIsencaoIVPA** presente agora no **prototype** do construtor **Carro** e por herança, presente no **prototype** de todas as suas instâncias.

Em nosso exemplo, nosso método possui poucas linhas, o que não pesaria tanto caso deixássemos o método de cálculo de isenção de IPVA em cada objeto de fato, como uma propriedades, porém imagine que fosse um método com cem linhas de código e possuíssemos vinte instâncias de nosso construtor, desta forma acabaria pesando, então colocando o método no **prototype** do construtor e utilizando de herança deixamos nosso código muito mais leve no momento da execução.

Prototype no Console

Vimos até agora no console do navegador apenas as informações superficiais de nossas variáveis e objetos, porém agora que sabemos como funciona o **prototype** de um objeto, podemos usufruir melhor do console do navegador, uma ferramenta realmente poderosa, que nos permite ver como é a estrutura e o **prototype** de nossos objetos.

script.js

```
01 function Carro(nome, cor, marca, ano) {
02   this.nome = nome;
03   this.cor = cor;
04         this.marca = marca;
05         this.ano = ano;
06 };
06
07 Carro.prototype.velocidade = 0;
08 Carro.prototype.acelerar = function () {
09   this.velocidade += 5;
10 };
11 Carro.prototype.parar = function () {
12   this.velocidade = 0;
13 };
14
15
```

```
16 var fusca = new Carro('Fusca', 'Azul', 'VW', 1975);
17
18 console.log(fusca);
```

Vejamos um exemplo já utilizado, porém um pouco modificado. Na linha 8 atribuímos ao **prototype** do construtor **Carro** uma propriedade chamada **velocidade** e preenchemos a mesma com o valor **0** , na linha 9 criamos um método atribuído à propriedade **acelerar** e na linha 12 um método atribuído à propriedade **parar** .

Como atribuímos estas propriedades ao **prototype** do construtor, significa que por herança todas as instâncias do objeto **Carro** terão também acesso a estas propriedades.

Podemos ver que há uma seta logo atrás de **Carro** no console do navegador, clicando sobre ela podemos expandir nosso objeto e visualizar mais informações.

Com as informações expandidas, podemos ver que há alguns níveis de herança. Nos quadrantes demarcados vemos as propriedades do objeto **fusca** assim como o **__proto__** do objeto **fusca** .

A propriedade **__proto__** do objeto **fusca** nada mais é que o **prototype** dele, como o construtor utilizado para criar o objeto **fusca** foi o objeto **Carro** , ambos terão o mesmo **prototype**

Experimente digitar no console do navegador isto:

fusca.__proto__ == Carro.prototype

Irá aparecer no console **true** , pois isto confirma que o atributo **__proto__** do objeto fusca tem exatamente o mesmo conteúdo que o **prototype** do construtor **Carro** .

Vimos anteriormente que todo e qualquer objeto tem como construtor o objeto **Object** , então veja o nosso objeto **fusca** expandido. No quadrante **Carro** também tem um **__proto__** e este

tem, como já comentados e como mostra no console, como construtor o objeto **Object** .

Podemos ver que no quadrante **Object** existem diversos métodos, então vamos ver o método **hasOwnProperty** presente no **prototype** do object **Object** que tem como função verificar se um objeto tem como propriedade própria alguma propriedade específica.

Execute no console do navegador este comando:

fusca.hasOwnProperty(acelerar);

O comando irá mostrar no console **false** , pois a propriedade **acelerar** não pertence ao objeto **fusca** em si, mas sim no **prototype** dele, assim como no **prototype** de todas as instâncias do objeto **Carro** . Assim confirmando que se utilizarmos o **prototype** de nossos objetos da forma correta iremos otimizar muito nosso código em questão de peso e dinamicidade.

A propriedade **constructor** presente em todos os **__proto__** dos objetos irá sempre informar quem é o construtor de cada objeto.

Object.create

Vimos como criar um objeto a partir de um construtor, que é a forma mais comum utilizada e que você vai achar na internet, porém existe a possibilidade de utilizar um método do objeto **Object** para criar objetos que é o **Object.create** , vejamos um exemplo:

script.js

```
01 // Como não é um construtor o nome do objeto
02 // Inicia em minúsculo
03 var carroProto = {
04   velocidade: 0,
05   acelerar: function () {
06     this.velocidade += 5;
07   },
08   parar: function () {
09     this.velocidade = 0;
```

```
10   }
11 };
12
13 // Utilizando uam função do prototype do Object
14 // Para criar um objeto
15 var fusca = Object.create(carroProto, {
16   nome: { value: 'Fusca' },
17   cor: { value: 'Azul' },
18         marca: { value: 'VW' },
19         ano: { value: 1975 }
20 });
21
22 console.log(fusca);
```

No exemplo, na linha 15 escrevemos nossa lógica para criar o objeto **fusca** , porém desta vez não utilizamos um construtor, mas sim o método **Object.create** .

O método **Object.create** consiste em passar no primeiro parâmetro um objeto simples, que nos criamos na linha 3 chamado **carroProto** , destinado a ser o **prototype** do objeto **fusca** e como segundo parâmetro passamos um objeto contendo as propriedades que o objeto **fusca** terá de fato.

O segundo parâmetro é um objeto meio estranho, mas realmente é esta a estrutura que o segundo parâmetro deve ter para que se possa utilizar o método **Object.create** :

```
var  segundoParametro: {
        propriedade: { value : valor},
        propriedade: { value : valor},
        propriedade: { value : valor}
}
```

Vejamos o objeto criado utilizando o método **Object.create** no console do navegador:

Veja que o objeto **fusca** está praticamente igual ao objeto **fusca** criado quando utilizamos um construtor, porém a única grande diferença, além da forma de escrever o código, é que o objeto **fusca** não tem a propriedade **constructor** em seu **__proto__** , pois de fato não utilizamos um construtor para a sua criação.

Ao utilizar o método **Object.create** o objeto **fusca** se tornou uma instância direta do objeto **Object** , assim como todo e qualquer objeto criado sem a utilização de um construtor predefinido.

Dados Primitivos x Objetos

Falamos sobre este assunto de forma superficial durante o decorrer do livro até aqui, porém não entramos a fundo sobre o assunto, então agora que já obtivemos um bom conhecimento sobre

Objetos e suas características, vamos falar sobre a diferença entre dados primitivos e objetos.

A grande diferença entre dados primitivos e objetos é que uma variável, que recebe como valor um dado primitivo, mantém de fato esse valor armazenado e, com objetos é muito diferente, variáveis associadas com objetos não contém o objeto de fato, mas sim uma referência de onde o objeto está armazenado na memória, ou seja quando criamos um novo objeto na verdade ele é armazenado na memória e é retornado para a variável associada a ele um ponteiro indicando onde ele está armazenado para que se possa o manipular.

Vamos ver na prática como essa diferença acontece.

script.js

```
01 // dados primitivos
02 var nome1 = 'Kayo';
03 var nome2 = nome1;
04 nome1 = 'Forest';
05
06 console.log(nome1, nome2);
```

Veja o exemplo, na linha 2 declaramos uma variável chamada **nome1** e a preenchemos com um texto, na linha 3 criamos uma outra variável chamada **nome2** e a preenchemos com o valor da variável **nome1** , na linha 4 preenchemos a variável **nome1** com outro valor e em seguida mostramos o valor da variável **nome1** e **nome2** no console do navegador.

Veja o resultado:

Console

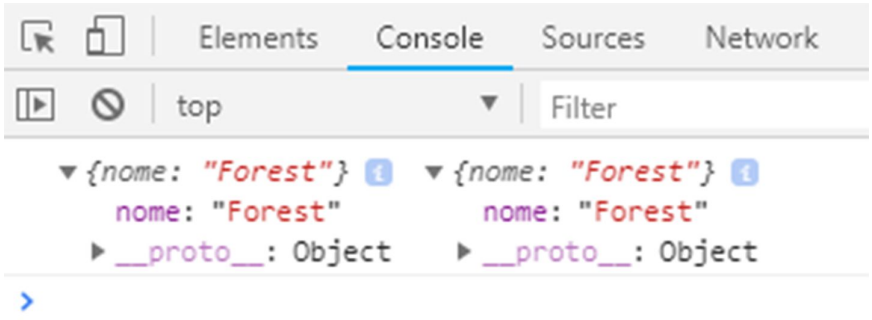
```
Forest Kayo
```

Como falado anteriormente o valor das variáveis é diferente, como esperado, pois quando trabalhamos com tipos de dados primitivos, sempre que atribuirmos um valor a uma variável, esta variável armazenará de fato o valor informado, assim quando atribuímos o valor da variável **nome1** a variável **nome2** na linha 3 de nosso exemplo, realizamos uma cópia do valor da variável **nome1** e armazenamos a cópia deste dado na variável **nome2** , diferente de como acontece com objetos.

script.js

```
01 // dados primitivos
02 var obj1 = {
03   nome: 'Kayo'
04 };
05 var obj2 = obj1;
06 obj2.nome = 'Forest';
07
08 console.log(obj1, obj2);
```

Veja o exemplo, na linha 2 criamos uma variável denominado **obj1** e atribuímos a ela um objeto com uma propriedade chamada **nome** , na linha 5 criamos uma outra variável chamada **obj2** e atribuímos a ela o valor da variável **obj1** , em seguida na linha 6 atribuímos ao valor da propriedade **nome** da variável **obj2** um outro valor diferente do atribuído à propriedade **nome** da variável **obj1** e na linha 8 mostramos o valor de ambos os objetos no console do navegador, veja o resultado:



Podemos ver que o valor das variáveis é igual, como esperado, pois como dito anteriormente a variável **obj1** não possuía o valor do objeto em questão, mas o valor de um ponteiro para objeto criado, então quando na linha 5 atribuímos o valor da variável **obj1** a variável **obj2** , na verdade atribuímos a variável **obj2** o mesmo ponteiro para o objeto criado na linha 2.

Função de primeira classe

Existe um termo em ciência da computação que é Função de primeira classe, em inglês *First-class function* , que consiste em

permitir passar funções como parâmetros, ou retornar funções como resposta de alguma execução.

A linguagem JavaScript é uma função que permite esse tipo de comportamento, pois tem suporte a funções anônimas, e quando utilizamos este padrão de chamada, onde passamos uma função como parâmetro, esta função passada como parâmetro é chamada de **callback** , pois ela será executada dentro da função chamada.

script.js

```
01 var arrAnoCarros = [1965, 1992, 1997, 1975, 1981];
02
03 // Função callback
04 function calcularTempoCarro(anoCarro) {
05   var data = new Date();
06   return data.getFullYear() - anoCarro;
07 }
08
09 function anoDosCarros(arr, fun) {
10   var arrTempoCarros = [];
11   for (var i = 0; i < arr.length; i++) {
12     arrTempoCarros.push(fun(arr[i]));
13   }
14   return arrTempoCarros;
15 }
16
17 // Passando no segundo parametro uma função
18 var arrCarros = anoDosCarros(arrAnoCarros, calcularTempoCarro);
19
20 console.log(arrCarros);
```

Veja o exemplo, na linha 1 criamos uma variável chamada **arrAnoCarros** onde a preenchemos com um array contendo o ano em que vários carros foram fabricados.

Na linha 4 declaramos uma função chamada **calcularTempoCarro** onde irá calcular a diferença em anos do ano atual **2018** e o ano passado por parâmetro **anoCarro** .

Na linha 9 criamos uma função chamada **anoDosCarros** que espera dois parâmetros, o primeiro um array contendo os anos em que os carros foram fabricados e no segundo parâmetro uma função. Na linha 11 a função **anoDosCarros** utiliza um laço de repetição para percorrer o array **arr** , passado por parâmetro, na linha 12 a função usará o método **push** para armazenar o retorno da função, que passamos por parâmetro, no Array **arrTempoCarros**

declarado na linha 10 e na linha 14 retornamos o **arrTempoCarros** contendo agora a idade dos carros. Na linha 18 executamos nossa função **anoDosCarros** passando por parâmetro o **arrAnoCarros** e a sintaxe da função **calcularTempoCarros** .

Sendo assim a função **fun** que chamamos na linha 12 na verdade será a mesma função **calcularTempoCarros** , se você lembrar do tópico de empréstimo de métodos, esta passagem de função como parâmetro é muito semelhante.

Console
[53, 26, 21, 43, 37]

Veja que no console do navegador mostra o valor de um Array contendo o tempo que passou desde que os carros foram fabricados até 2018, isto foi possível, pois na verdade na linha 12 a sintaxe do parâmetro **fun** é a mesma da função **calcularTempoCarros** , pois informamos para a função **anoDosCarros** na linha 18, entretanto poderemos passar diferentes funções por parâmetro para a função **anoDosCarros** e assim obter diversos resultados.

IIFE

Neste tópico vamos ver um padrão de código extremamente comum na linguagem JavaScript que é Expressões de Função Imediatamente Invocadas, em inglês *Immediately Invoked Function Expressions* , também chamada de *IIFE* , como nome já diz este tipo de declaração de função será auto executada.

Em JavaScript o que está entre parênteses não pode ser uma declaração, apenas uma expressão, por isso colocamos nossa declaração de função entre parênteses, para o JavaScript entender que não é uma declaração perdida no meio de nosso código e sim uma expressão, assim não gerando um erro pelo interpretador do navegador.

```
(function () {
    // nosso código aqui
})();
```

Também é possível passarmos parâmetros para nosso IIFE:

```
(function (param1, param2) {
    // nosso código aqui
})(valorParam1, valorParam2);
```

Imagine que você queira criar um jogo simples, um jogo onde será ganhador quem tirar um número gerado aleatoriamente maior ou igual a **10** e irá perder quem tirar um número menor que 10, entretanto queremos que o valor da pontuação fique escondido neste jogo, o valor da pontuação não será visível para os jogadores.

Pelos conhecimentos que agregamos até agora você deve estar pensando em usar uma função e declarar esta pontuação dentro desta função, pois como vimos até agora não é possível acessar esta variável fora do escopo que ela foi criada, logo não é possível acessá-la fora da função onde foi declarada.

Poderíamos utilizar uma declaração de função simples, mas como o objetivo é simplesmente ocultar do escopo global o valor da pontuação iremos utilizar um IIFE para demonstrar sua sintaxe.

script.js

```
01 (function () {
02   var pontuacao = Math.random() * 20;
03   if (pontuacao >= 10) {
04     console.log('Jogador ganhou!');
05   } else {
06     console.log('Jogador perdeu!');
07   }
08 })();
```

Veja o exemplo, na linha 1 declaramos nosso IIFE e dentro dele criamos uma variável chamada **pontuacao** que recebe como valor um número aleatório (randômico) entre **0** e **20** . Na linha 3 criamos uma decisão lógica para verificar se a pontuação é maior ou igual a **10** para informar no console do navegador se o jogador ganhou ou perdeu.

Utilizando IIFE da forma correta conseguimos utilizar uso pesado de privacidade de dados e modularização de código, veremos melhor esses conceitos no próximo tópico.

Closure

Até agora vimos realmente muita teoria e a este ponto você já tem o conhecimento necessário para entender um dos conceitos mais importantes e cruciais da linguagem JavaScript que é traduzido de forma literal como **fechamento** , em inglês *Closure* .

Este é um dos tópicos que novos desenvolvedores JavaScript tem grande dificuldade de compreender, porém juntando todo o conteúdo que vimos até agora, acredito que você terá facilidade de ter um bom entendimento sobre como **Closures** funcionam.

script.js

```
01 function tempoIsencaoIPVA(anoAtual) {
02   var mensagem = "Carro não possui isenção de IPVA";
03   var mensagemIsencao = "Calculo dentro da faixa de isenção de IPVA";
04
05   return function (anoCarro) {
06     var idadeCarro = anoAtual - anoCarro;
07     if (idadeCarro > 30) {
08       console.log(mensagemIsencao);
09     } else {
10       console.log(mensagem);
11     }
12   };
13 }
14
15 var verificadorIPVA = tempoIsencaoIPVA(2018);
16
17 verificadorIPVA(1978);
18 verificadorIPVA(1991);
```

Veja o exemplo, na linha 1 declaramos uma função chamada **tempoIsencaoIPVA** que espera receber um parâmetro chamado **anoAtual** , dentro desta função criamos uma variável na linha 2 chamada **mensagem** contendo um texto e na linha 3 uma outra variável chamada **mensagemIsencao** contendo um outro texto. Na linha 5 retornamos uma função anônima que espera um parâmetro chamado **anoCarro** , dentro desta função anônima criamos uma decisão lógica para verificar se o carro se enquadra na faixa de tempo para ter isenção do IPVA, e dependendo do retorno da

decisão lógica, mostramos no console do navegador ou a variável **mensagem** ou a variável **mensagemIsencao** .

Na linha 15 executamos a função **tempoIsencaoIPVA** passando como parametro **2018** e armazenamos o retorno da função na variável **verificadorIPVA** , como o retorno da função **tempoIsencaoIPVA** é uma função, nas linhas 17 e 18 executamos essa função de retorno passando parâmetros diferentes.

Veja que no console do navegador irá aparecer o valor das duas variáveis de mensagens declaradas dentro da função **tempoIsencaoIPVA** , mas espera um pouco, essas variáveis já não saíram da pilha de execução, depois que a função **tempoIsencaoIPVA** encerrou sua execução elas continuam visíveis.

Realmente as variáveis declaradas na função **tempoIsencaoIPVA** , que já deixou a pilha de execução, continuam dentro da cadeia de escopo da função anônima que retornamos da função **tempoIsencaoIPVA** e isto é o que chamamos de **closure** e é possível utilizarmos isto, pois uma função interna sempre tem acesso a variáveis e parâmetros de sua função externa, mesmo após a função externa ter retornado, ou seja mesmo que um contexto de execução tenha fechado e retirado da pilha de execução a cadeia de escopo continua intacta.

Utilizar closure é o meio mais popular de encapsular, manter protegido do mundo externo, o código de nossos aplicativos e estruturas, normalmente expondo uma única interface, objeto, global na qual é possível interagir com nosso código.

Modulo

Modularizar uma aplicação é um dos padrões mais utilizados na criação de códigos JavaScript.

Em JavaScript, utilizamos os conceitos de **Closure** e **IIFE** para criar módulos para nossos scripts. Conseguimos modularizar nossa aplicação ou site, mantendo as informações correlacionadas, independentes e organizadas. A estrutura de um módulo consiste basicamente em criar uma variável e atribuir a ela como valor um

IIFE e este if retornar apenas o que queremos que o mundo externo veja:

```
var nomeDoModulo = ( function () {
    var variavelPrivada = 10;
    var funcaoPrivada = function () {
        console.log(variavelPrivada);
    };

    return {
        acessoExterno: funcaoPrivada
    };
})();
```

Utilizando a estrutura de código acima conseguimos criar um módulo.

script.js

```
01 var moduloCarro = (function () {
02   // variavel privada
03   var velocidade = 0;
04
05   // função privada
06   var girarEngrenagens = function () {
07     velocidade += 5;
08   };
09
10   // função privada
11   var aumentarVelocidade = function () {
12     girarEngrenagens();
13   };
14
15   // retorno interface de acao
16   return {
17     acelerar: aumentarVelocidade
18   };
19 })();
20
21 moduloCarro.acelerar();
```

Veja no exemplo, na linha 1 declaramos uma variável chamada **moduloCarro** recebendo como valor o nosso **IIFE** , dentro de nosso **IIFE** na linha 3 estamos declarando uma variável chamada **velocidade** recebendo **0** como valor, na linha 6 declaramos a variável **girarEngrenagens** recebendo como valor uma função

anônima que irá somar a variável **velocidade** mais **5** a cada vez que a função for chamada, na linha 11 declaramos a variável **aumentarVelocidade** recebendo como valor uma função que irá executar a função **girarEngrenagens** a cada vez que for chamada e na linha 16 retornamos um objeto com uma propriedade chamada **acelerar** recebendo emprestada a sintaxe da função **aumentarVelocidade** . Na linha 21 executamos o método **acelerar** de nosso módulo, pois foi o que retornamos em nosso **IIFE** criando assim nosso módulo, porém se tentarmos acessar qualquer uma das outras propriedades ou funções que declaramos dentro de nosso **IIFE** não iremos conseguir, pois estes estão ocultos do mundo externo, encapsulados. Conseguimos ter acesso apenas ao que retornamos de nosso IIFE.

Como mostrado no exemplo, não temos acesso as engrenagens de um carro, porém podemos acelerar o carro, pois é o que temos acesso e o que nosso módulo nos permite visualizar e usar, caso estivessemos desenvolvendo algum modulo que fosse destinado a ser utilizado por terceiros, poderíamos manter as informações que desejássemos seguras, enquanto mantemos outras visíveis, isso é possível graças a utilização de módulos e closures.

Conclusão

Agora depois de ver todo o conteúdo de nosso livro, temos total entendimento de como funciona a linguagem JavaScript, assim como suas particularidades, conceitos e pontos fortes.

Agora você tem todas as ferramentas para criar um script muito bem estruturado e organizado, umas das capacidades que muitas empresas prezam.

Sobre o Autor

Brasileiro, Kelvin Baumhardt Biffi nasceu na cidade de Porto Alegre no estado do Rio Grande do Sul. Kelvin trabalha na área de TI desde seus 17 anos de idade como desenvolvedor, já passado por grandes empresas incluindo multinacionais. Kelvin sempre foi apaixonado por desenhar e escrever desde sua infância, atualmente tem a oportunidade de realizar seus sonhos como escritor, juntando suas paixões.

Contato Online
http://kelvins.cc