JavaScript Furtivo

Eric Douglas

JavaScript Furtivo

Iniciando no mundo do desenvolvimento JavaScript!

Eric Douglas

Esse livro está à venda em http://leanpub.com/javascriptfurtivo

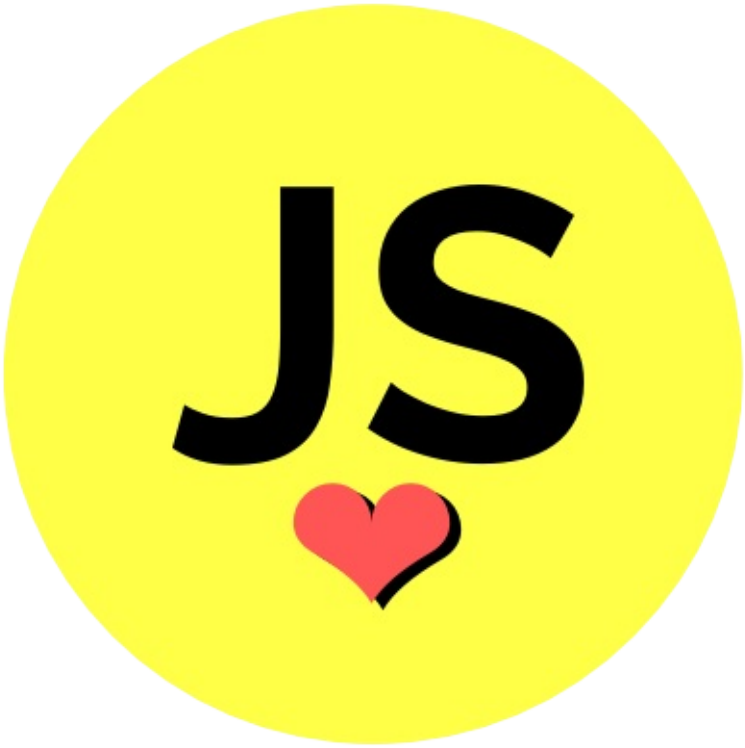
Essa versão foi publicada em 07/05/2014

* * * * *

* * * * *

Índice analítico

Apresentação



Neste livro iremos focar nos tópicos iniciais da linguagem JavaScript, sendo este um guia **ideal** para quem está iniciando no desenvolvimento web, e também para quem já possui algum conhecimento mas pretende estudar com base em algum material mais aprofundado e com vários exercícios práticos, consolidando assim o que já sabe e aprendendo coisas novas.

Após cada etapa teórica, irei apresentar **exemplos** e **exercícios** para fixação de todo o conteúdo, e após uma parte significativa dos estudos, iremos criar **projetos** para total entendimento prático de como aplicar essas informações. Os **exercícios** e **projetos** serão resolvidos e contarão com uma detalhada explicação, para total entendimento por parte do leitor.

Outro ponto importante é que comprometo-me a ser o mais prático possível. Sim, teoria é ótimo e nos dá muita base, mas iremos ficar também bastante tempo no console testando e aplicando nossos códigos.

Para seu aprendizado ser consolidado, e a informação contida neste livro se transformar em **seu** conhecimento, é extremamente necessário que você pratique todos os códigos aqui presentes, e continue a leitura apenas quando tiver um entendimento mínimo do que eles estão fazendo.

Em caso de dúvidas sobre os temas apresentados, você pode compartilhá-las nos grupos JavaScript Brasil e impJS. Apenas não deixe dúvidas acumularem. Irei tentar lhe ajudar da melhor maneira possível!

Além disso, você pode ficar por dentro das novidades do blog, novos livros/projetos e muito mais sobre o universo JavaScript assinando a minha newsletter. Prometo que só terá links e assuntos **úteis** relacionados ao universo do JavaScript! Nada de spam!

Motivação para este trabalho

Tenho uma enorme satisfação em poder compartilhar conhecimento, e faço isso de forma mais empenhada ainda quando se trata de **JavaScript**!

Sendo assim, para que este conhecimento não fique restrito apenas para quem pode pagar, mas esteja disponível para todos que, de fato, queiram aprender, decidi criar este material e disponibilizá-lo de forma gratuita no meu blog.

Realmente é um prazer imenso poder fazer parte desta comunidade de desenvolvedores, e esse livro (e os próximos) serão uma pequena retribuição, um agradecimento a tudo o que já aprendi através dela.

Bom, se tenho o livro de graça, por que pagar?

É fato que o trabalho de se escrever um livro consome **muito** tempo e exige uma **grande dedicação**.

A compilação deste trabalho em formato de livro tem como objetivo *experimentar* uma nova forma de compartilhar conhecimento, onde irei **presenteá-lo** com o **melhor** conteúdo que me for possível criar, e caso você fique satisfeito com este, você também poderá me *presentar* com sua colaboração.

Além dessa forma de *"negociarmos"* através de presentes, estou iniciando uma metodologia própria de divisão do conteúdo a ser ensinado de forma **modular**. Irei te explicar melhor isso no póximo capítulo.

Tendo este formato obtido êxito, já tenho em mente outros temas para novas séries *furtivas*. Se você está curioso em saber quais são estes possíveis temas, irei lhe contar.

Próximos Temas

- **NodeJS**
- **Meteor**
- **Express**
- **AngularJS**
- **MongoDB** (e bancos de dados NoSQL)
- **MEAN**
- e também a combinação **Jade e Stylus**.

Conclusão

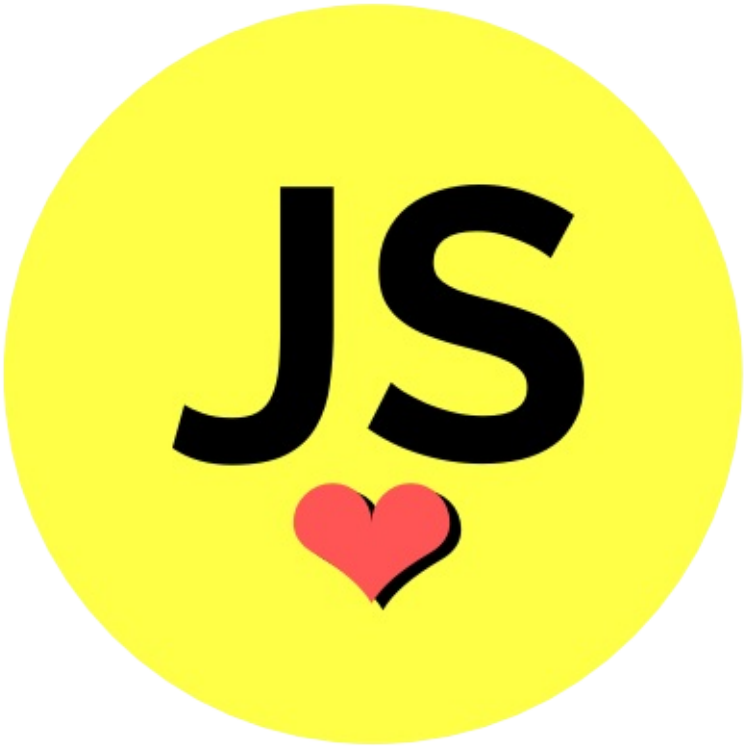
O objetivo desta série é ser uma referência de material de **alta qualidade** sobre JavaScript, levando o **melhor** do mundo teórico e prático para você.

Os grandes diferenciais são o conteúdo sintetizado e claro, levando o leitor direto aos pontos chaves da linguagem, e os diversos exercícios e projetos práticos **resolvidos** e

comentados, que fecham o ciclo do conhecimento e domínio **real** dos assuntos abordados.

> “A fortaleza íntima do homem é a sua sinceridade de propósitos. Enquanto árvores colossais nenhum fruto produzem, outras pequenas árvores cobrem-se de belos pomos.”

Livros Modulares



Continuando as inovações na forma de se apresentar conteúdo técnico, irei implementar um sistema onde o leitor poderá **montar o seu próprio livro**, de acordo com sua vontade/necessidade.

Como assim? Ainda não entendi!

Simples! Os livros serão divididos por áreas e paradigmas da linguagem. Este primeiro livro, **JavaScript Furtivo**, irá abordar a estrutura (sintaxe) básica da linguagem e a parte lógica utilizada em suas expressões/instruções.

Os próximos livros abordarão funções, objetos, arrays e demais assuntos específicos da linguagem, tendo cada um a devida profundidade no tema.

Porém, para que você não fique defasado em relação a estes outros assuntos, no final deste livro estará presente uma lista de recursos para você continuar por conta própria estudando estes temas, a única diferença é que a explicação dos mesmos ficará para futuras publicações (que sairão em breve).

Mas o que eu, leitor, ganho com isso?

Liberdade!!!

Acompanhe comigo a seguite situação:

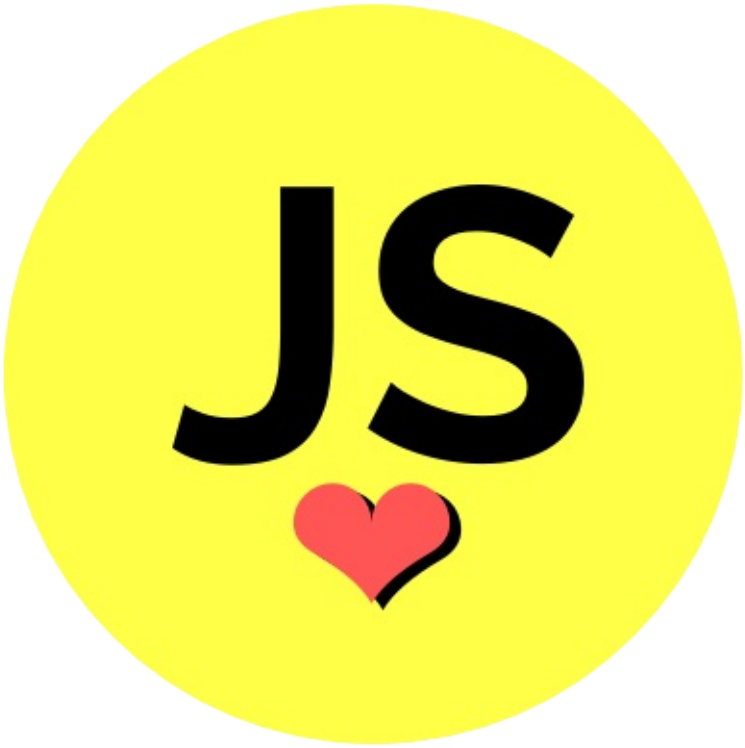
Imagine uma pessoa que já tem experiência com desenvolvimento JavaScript, estudou seus princípios básicos e tem um bom entendimento do paradigma *orientado a objetos*, mas tem muita vontade de aprender mais sobre *programação funcional*, e gostou do conteúdo apresentado nesta série, através do meu blog, e gostaria de adquirir o livro para poder estudar através dos exercícios e projetos.

Imaginou isso? Legal! Agora vamos continuar, se ele já sabe a maior parte e quer apenas aprofundar em *funções*, se o material for disponibilizado de forma única, ele terá que pagar por **todo** o livro, porém, se o mesmo estiver **modularizado** (dividido), ele poderá adquirir **apenas** a parte que lhe interessa. **Justo**!

Então, com essa visão que inicio essa nova maneira de dispor conteúdo, permitindo que o mesmo seja adquirido de forma independente.

Outro ponto importante nessa abordagem é que assim teremos mais liberdade para aprofundar em cada tema em específico, elevando e muito nosso conhecimento e dominío nos temas.

Depoimentos



Aqui você pode encontrar algumas opiniões dos leitores do blog sobre o conteúdo deste livro **JavaScript Furtivo**!

> Show a comunidade precisava de um projeto desses!
>
> - Suissa

> Curti muito a iniciativa! Sucesso, cara! =D
>
> - Leonardo Alberto Souza

> Tá cada dia melhor!!!
>
> - Valdiney França

> Parabéns cara perseverança.
>
> - Osmar Cs

> Ta muito da hora
>
> - Luiz Henrique

Muito Bom! Show!

- Falconiere Rodrigues Barbosa

Perfect !

- Jonathan Brunno

Muito legal!

- Deise Dall’Agnol

Muito bom!

- Robson Bittencourt

Muito bom!

- Márcio Carvalho

Ótimo! Bom saber que existem pessoas escrevendo artigos de JS em português

- Luis Fontes

Muito bom artigo e bem explicado.

- André Luiz

Boa Eric. Tenho acompanhado desde a primeira parte e está sendo bem proveitoso pra mim. Finalmente vou aprender JS, xD.

- Euclides Fernandes

Muito bom!

- Raphael Fabeni

Estava realmente precisando, bem legal os artigos. parabéns!

- Jefferson Daniel

Show de bola hein!

- Flávio

Post muito bom Eric parabéns cara!!

- Rafael Antonio Lucio

Parabéns Eric!

- gustavoSRodrigues

Muito bom! parabéns mesmo Xará rs

- Erick

Isso ai Eric! Episódios de alta qualidade… parabéns pelo ótimo trabalho!!

- Guilherme Decampo

Sou fã desse Eric

- Palmer

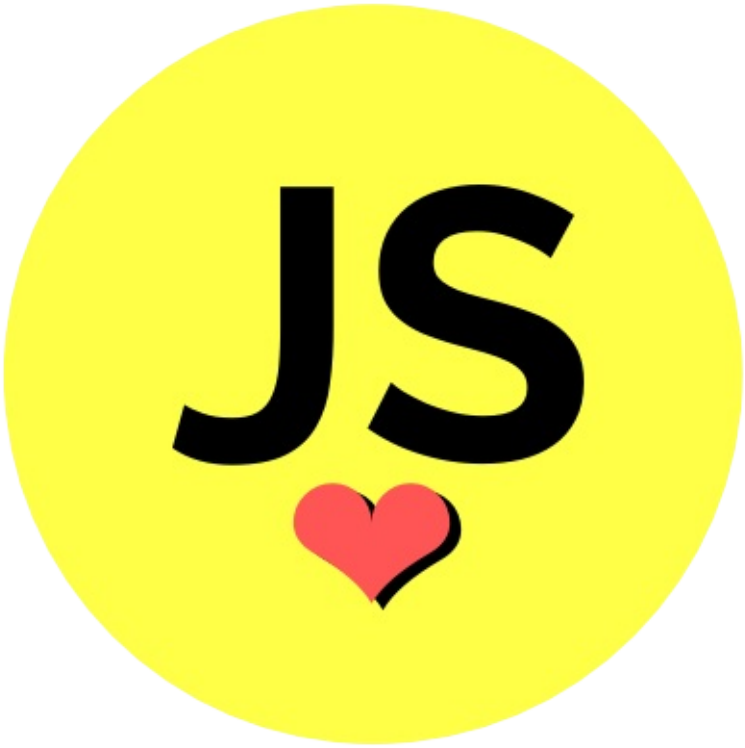
Muito obrigado. Como sempre, excelentes conteúdos. Abraço.

- Denis Toledo

Nice post man! +1

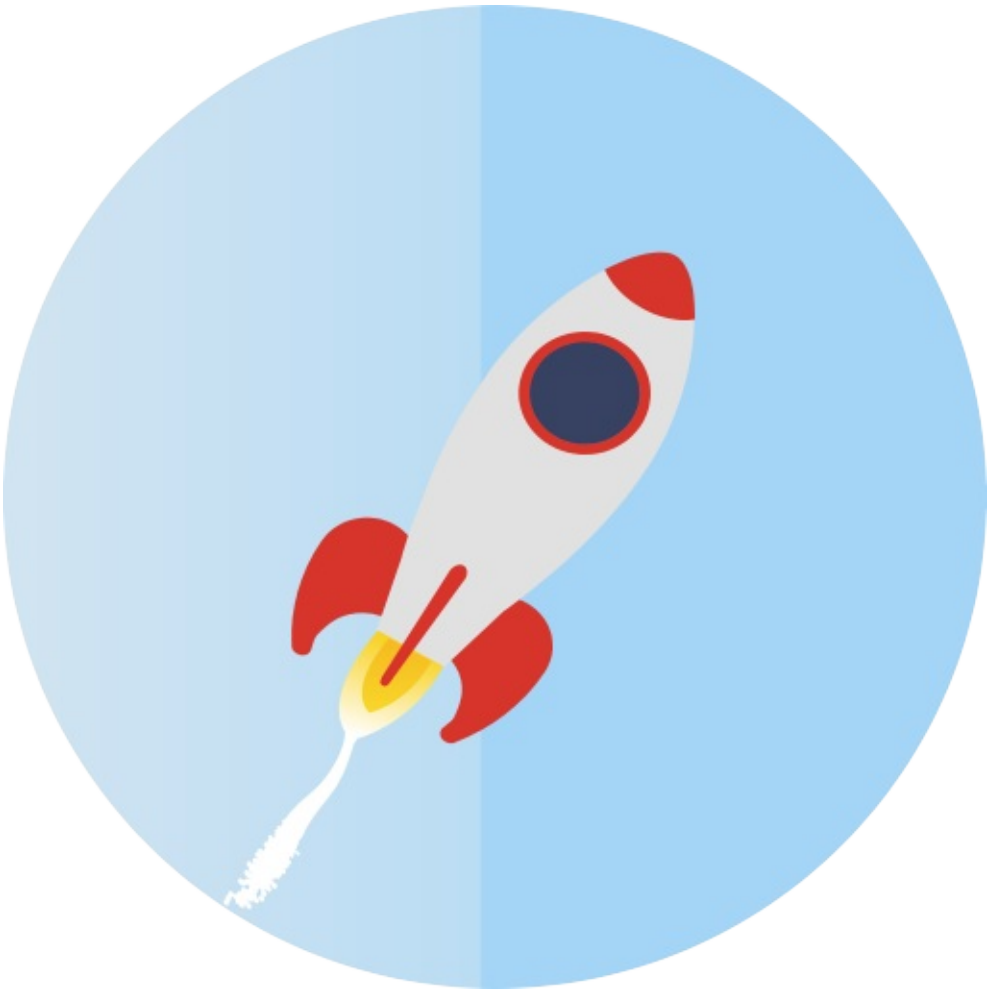
- Caio Ribeiro Pereira

Sobre a Capa



A capa do nosso livro não foi criada apenas com fins *estéticos*, mas tem ali representada a síntese de todo o trabalho e ideologia que será encontrado neste livro e no projeto em si.

Foguete



Foguete

Você sabia que um foguete consome mais de **80%** do seu combustível apenas para ser lançado?

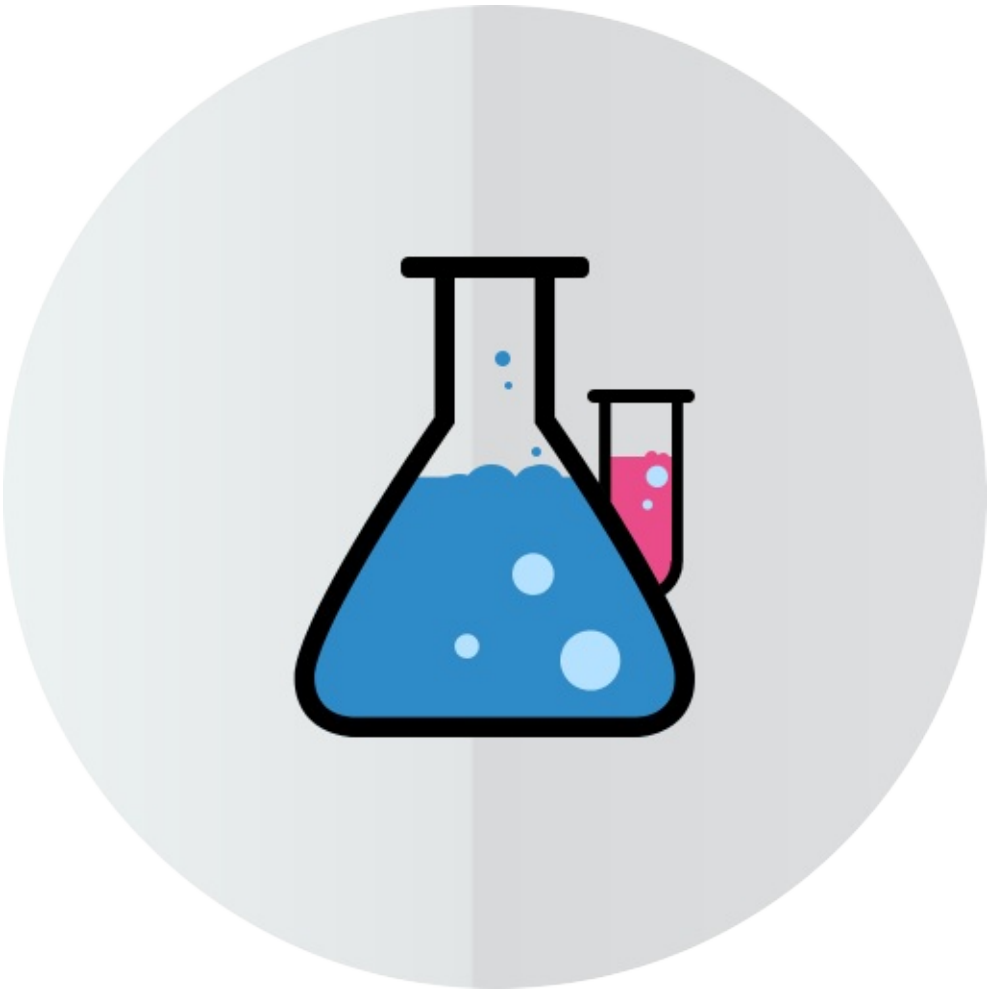
Exatamente, é um gasto absurdo de energia apenas para tirá-lo do chão… Mas não é sobre foguetes que quero falar aqui, mas sim sobre **dedicação** e, principalmente **hábito**.

Criar um (bom) hábito pode ser perfeitamente comparado com o lançamento de um foguete. No início, temos que lutar, nos esforçar **muito** para conseguir apenas *sair do chão*, mas depois que conseguimos superar essa barreira inicial, o vôo se dá de forma fluida.

Antes de tentar aprender sobre qualquer coisa, tenha em mente a certeza que você só conseguirá obter **bons resultados** se você cultivar **bons hábitos** relacionados a tal tarefa.

Cultive bons hábitos!, e elimine os maus…

Experimentos



Experimentos

Nosso livro estará recheado de **exemplos**, **exercícios** e **projetos**.

- Usaremos os **exemplos** para explicar trechos curtos da teoria, através de pequenos códigos que tem o papel de deixar visual e mais “palpável” este primeiro contato com um determinado assunto.
- Os **exercícios** estarão contidos no fim dos capítulos, e tem o objetivo de verificar se você realmente aprendeu os assuntos em questão. Vale ressaltar que além dos exercícios, no fim do livro estarão todas as resoluções dos mesmos, para auxiliá-lo **quando necessário**, e também a explicação detalhada de como o exercício foi resolvido.

ps: Olhe a resolução somente **após** tentar resolver os exercícios por várias vezes, pois isso é **essencial** para seu **aprendizado**. O importante não é você resolver exercício X ou Y, mas sim **entender** o que está acontecendo ali.

- Os **projetos** serão o ponto máximo dos nossos estudos, onde iremos juntar o conteúdo de alguns capítulos para criar aplicações **reais**, tendo com elas uma experiência prática de um verdadeiro **desenvolvedor JavaScript**!

Importante!

Não irei passar **nenhum** exercício/projeto que exija mais do que tenha sido explicado, **pode confiar**! Os exercícios serão exatamente do mesmo nível do que já tenha sido

ensinado. Eventuais dúvidas estarão ligadas mais em relação à *lógica* empregada, e isso é algo muito pessoal pois cada um visualiza o problema de uma forma.

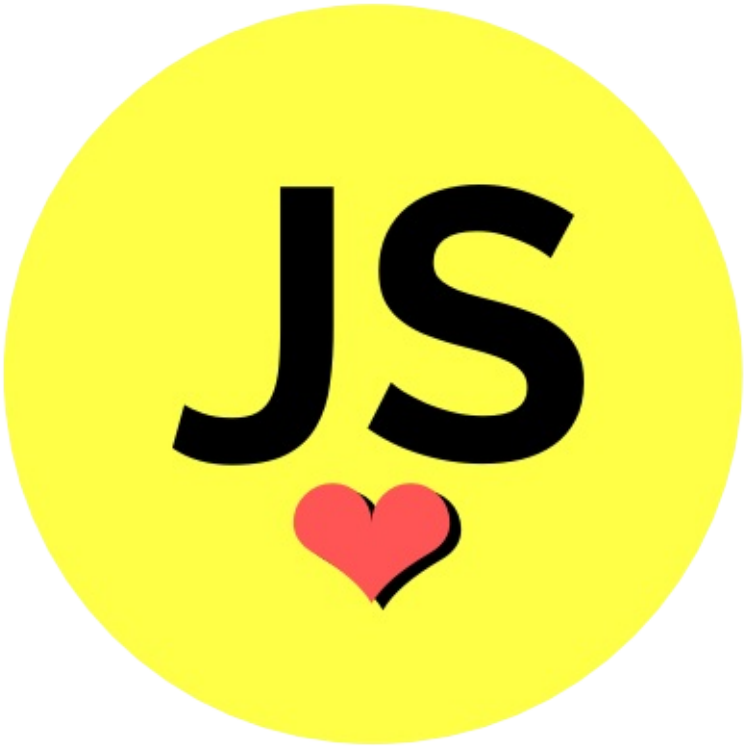
No entanto, não se preocupe com acertar ou não. Se esforce para resolver o problema, pois caso não consiga obter êxito inicial em algum problema, quando você verificar a resposta, essa será muito melhor aproveitada e absorvida do que se você tivesse a visto sem esforço prévio.

Terno e Gravata

Terno e gravata

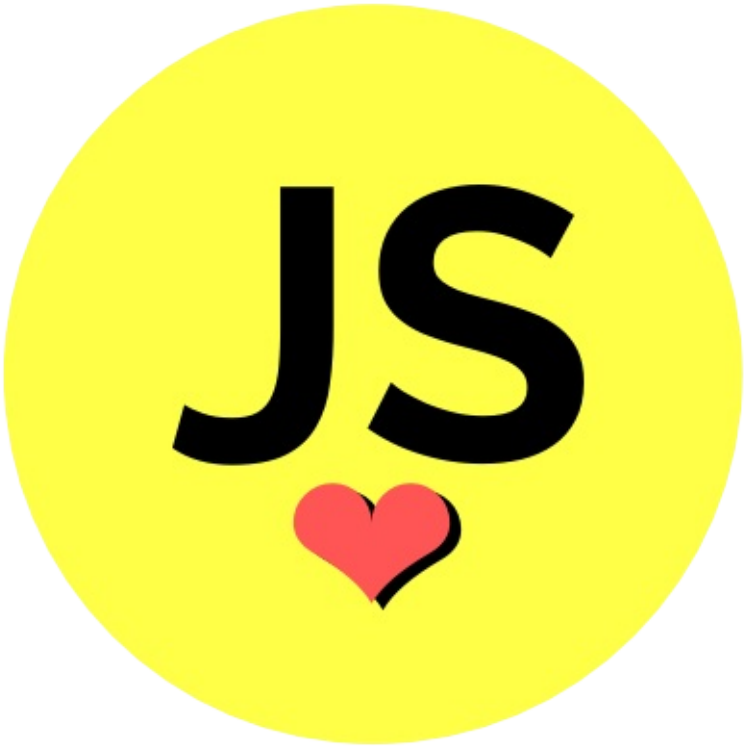
Essa ilustração, por fim, tem o objetivo de nos dizer que, ao término destes estudos, estaremos aptos a ser considerados verdadeiros **desenvolvedores JavaScript** com um sólido conhecimento da linguagem, e prontos para atuar e contribuir não somente com a comunidade de desenvolvedores, mas com o mundo todo, criando aplicações de alto nível, e implementando ferramentas que irão nos ajudar no nosso dia a dia!

Dúvidas e Considerações



Caso você tenha alguma dúvida ou consideração a fazer sobre o material aqui apresentado, além dos locais mensionados no capítulo *Apresentação*, você também poderá utilizar este grupo para se comunicar comigo e com os demais leitores do livro.

Prefácio



Agora que você está ciente de como irá funcionar essa nova proposta de compartilhamento de conteúdo, podemos iniciar nossa jornada rumo à maestria no JavaScript.

Neste livro você irá aprender toda a estrutura (sintaxe) básica da linguagem e sua estrutura lógica. Irei te mostrar como nomear suas variáveis, como criar dados, como usar os operadores que a linguagem nos fornece para criar programas de lógica, e no último capítulo será introduzido um tópico **avançado** com dicas de otimização para estes processos.

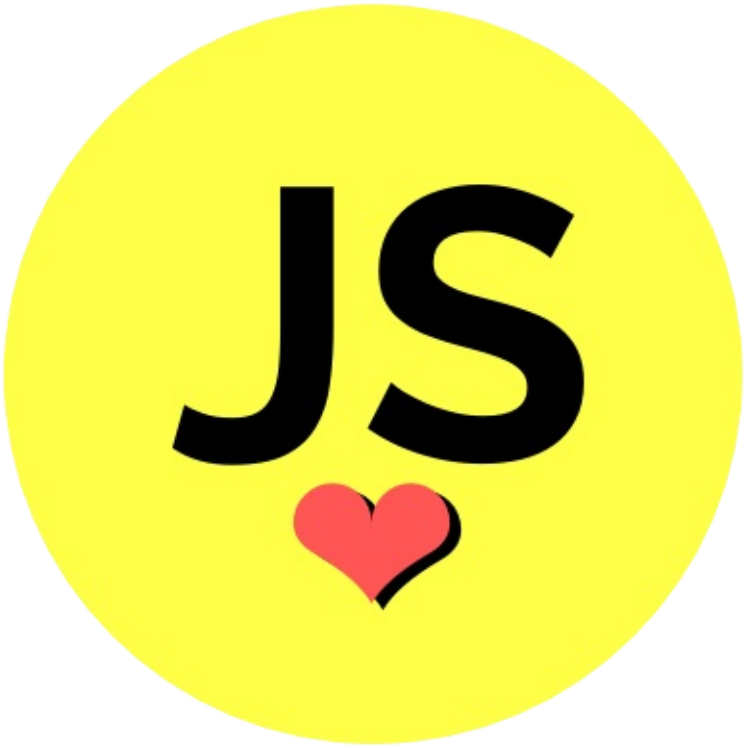
Fazendo uma comparação com a nossa linguagem *falada*, iremos aprender como criar palavras, frases e textos simples, mas de forma eficaz, construindo uma base **sólida** para o aprofundamento nos temas futuros.

Além disso, para cada capítulo do livro você terá uma lista de exercícios correspondente, e também a resolução detalhada de cada exercício, com explicações minunciosas, passo a passo, para que você possa entender **verdadeiramente** TUDO que está sendo feito.

E para fechar nosso ciclo de aprendizagem, teremos um projeto que irá englobar todos os assuntos abordados no livro, para consolidar nosso conhecimento.

Tenha uma ótima leitura! Vamos começar… =)

Olá JavaScript!



Antes de iniciarmos as apresentações, vou lhe contar algo que você deverá tomar cuidado ao compartilhar, pois pode lhe gerar algumas discussões, mas o fato é que o mercado nunca foi tão interessante e animador para um programador JavaScript como agora, e nunca foi tão onipresente para qualquer linguagem como está sendo para o JavaScript!

Sério, hoje podemos fazer maravilhas com o JavaScript, desde programação front-end (no lado do cliente [navegador]), como no back-end com o espetacular **Node.JS**, que além disso permitiu que o JavaScript fosse utilizado em diversas outras áreas além dos navegadores e mundo web.

O trio HTML, CSS e JavaScript está cada vez mais presente nos celulares, seja como aplicações híbridas ou até mesmo como o próprio sistema operacional, tendo como exemplo o Firefox OS.

Para mais motivação para aprender JavaScript, veja este link.

O que é JavaScript?

Vamos fazer uma analogia de uma aplicação web, seja um simples site ou um complexo sistema, com uma casa.

Na casa, toda a parte estrutural (tijolos, telhado) é equivalente ao HTML. Ele é responsável por estruturar e definir, semanticamente, quais elementos estarão presentes no

nosso documento (outro nome para nossa página HTML). É muito importante ter em mente **exatamente** o que cada tecnologia nos oferece, para aproveitarmos o melhor de cada, e não delegar tarefas de uma para a outra.

O CSS é responsável de fazer o acabamento da casa, digo, da aplicação. É a parte que devemos "estilizar" todo o documento, para que o mesmo não fique com cara de apenas um documento para impressão, mas sim com a cara de uma aplicação/programa o mais agradável e utilizável possível.

OBS: Tanto é verdade sobre a questão do HTML ser um documento, que sua origem foi justamente essa. A WWW (World Wide Web) foi criada com fim de auxiliar pesquisadores a mandarem seus trabalhos acadêmicos uns para os outros. O **HTML** (**H**yper**T**ext **M**arkup** **L**anguage, ou linguagem de marcação de hipertexto) foi a linguagem criada para identificação do navegador de onde se teria um título, um parágrafo, uma citação, e por ai vai… O que vemos hoje com o HTML5 é um sonho utópico, analisando o cenário atual com a mentalidade de quando esta tecnologia (HTML) fora criada.

Por último, e não menos importante, temos o JavaScript! Em relação a uma casa, a função do JavaScript seria dar "vida" a casa, ou seja, seria a luz e a água da mesma, por exemplo. De forma menos lúdica, de fato o JavaScript cuida das interações da página com o usuário, e vice-versa.

A função primária para qual o JavaScript foi criada era a de tratar informações do lado do cliente, sem que fosse necessário o envio de informações para processamento no servidor. Vale mais uma vez ressaltar a importância da base histórica para real compreensão e valorização das tecnologias, ferramentas que nos auxiliam a realizar determinada tarefa.

Hoje em dia, o acesso a internet se faz em grande parte utilizando serviços de banda larga. Bom, atualmente é assim! No início da nossa querida web, você poderia levar tranquilamente 1 minuto apenas para receber uma mensagem que seu formulário foi preenchido incorretamente…

Isso é só o começo, teremos uma longa caminhada para desvendar os segredos dessa linguagem fantástica que nos permite criar diversas coisas, tantas quanto nossa imaginação permitir!

Chega de conversa, vamos aprender Java!!!

Meu amigo, de coração, **nunca** diga Java referindo-se à nossa querida linguagem JavaScript. **Nunca**!!! Este nome, infelizmente, foi uma terrível escolha de marketing feita no início da história do JavaScript. Como a linguagem Java estava muito em alta na época em que o JavaScript foi criado, resolveram ~~estragar~~ nomear a linguagem de tal forma. Essa é a maior semelhança que Java e JavaScript tem.

JavaScript™

Nossa linguagem foi criada por Brendan Eich em 1995 quando o mesmo trabalhava na *Netscape* (hoje *Mozilla*). Já teve o nome de Mocha e LiveScript, porém quando foi apresentada a implementação de *Java* nos navegadores da Netscape em conjunto com a *Sun Microsystems* (hoje *Oracle*), teve seu nome alterado para *JavaScript*. Este nome,

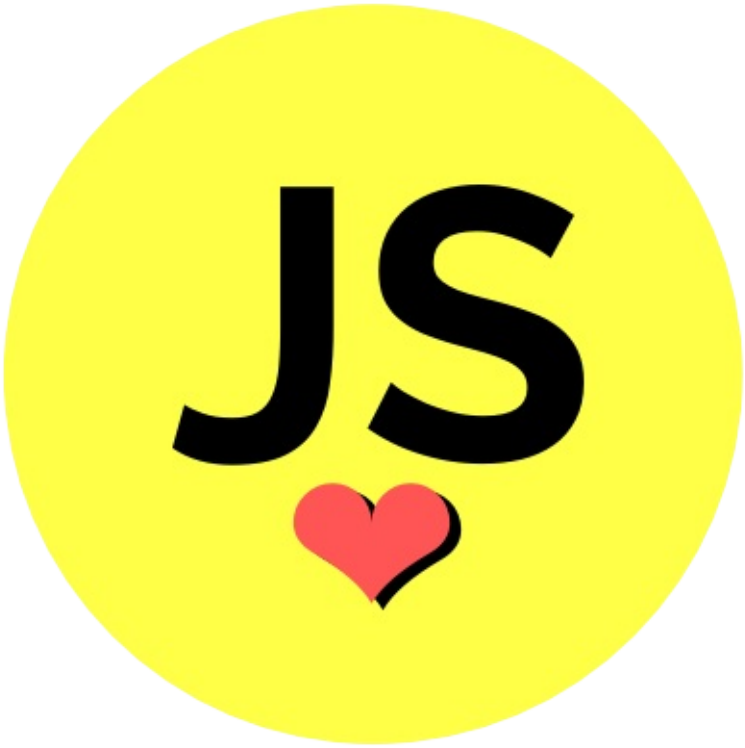
JavaScript, é marca registrada da Oracle atualmente.

Quando o JavaScript foi padronizado pelo grupo *ECMA* (European Computer Manufacter's Association), recebeu o nome de ECMAScript, e temos então essa versão padronizada implementada nos navegadores.

Já na ECMAScript temos que ter atenção atualmente a três de suas versões. A primeira é a versão **3** (ECMAScript 3), que é completamente suportada por todos os navegadores atuais. A segunda versão é a **5** (ECMAScript 5) que está em grande parte implementada, mas iremos analisar quando alguma *feature* (propriedade, recurso) da mesma não for comum entre os navegadores. A versão 5 é a que iremos abordar principalmente nessa série. E a próxima versão é a **6** (ECMAScript 6), que deverá ser lançada em breve. Preocupe-se hoje com a 3 e 5, a 6 é bom, no presente momento, apenas ter conhecimento de que está para chegar. =)

- Mais sobre o início da JavaScript aqui;

Conhecendo o Idioma JavaScript



Para você começar estudar a linguagem de programação JavaScript, é necessário que tenha o entendimento de como se dá esse intercâmbio dos caracteres que inserimos com a forma que os mesmos são interpretados pelo computador. Muito difícil? Vamos simplificar!

É algo bem simples, você se comunica com outras pessoas através de uma linguagem, a qual nada mais é que uma padronização de sequências de letras e fonemas que juntos referem-se a alguma coisa e dão sentido a tal. Na área da programação é exatamente a mesma coisa! Temos que conhecer as estruturas básicas da linguagem para começarmos a montar palavras, frases e textos, que no nosso caso serão os programas.

Essa estrutura de comunicação básica, ou estrutura léxica, nos indica como a linguagem deve ser escrita, sendo essa estrutura o nível mais baixo de abstração, nos indicando como criar e nomear variáveis, como inserir comentários nos códigos e como escrever as *expressões* e *instruções* que farão toda a mágica acontecer.

Agora que já temos em mente o que precisamos aprender, vamos, então, aprender!

Nesse capítulo você irá aprender:

1. Qual o padrão de escrita usado no JavaScript;
2. Detalhes de nomenclatura;
3. Como inserir comentários
4. Como nomear variáveis;
5. O que são identificadores e palavras-reservadas.

Leia sempre com bastante atenção cada tema passado e reflita-o um pouco para iniciar a absorção das informações. Todo o conteúdo deste livro é de suma importância para seu aprendizado e crescimento como **desenvolvedor JavaScript**, porém, não pense que você terá que aprender tudo na primeira vez que ler e/ou praticar.

O aprendizado é algo que leva tempo e muito esforço repetitivo, e muitas coisas não são triviais, e é isso que no final fará toda a diferença e dará a sua satisfação em ter vencido cada etapa do processo. =)

As “Letras” do Nosso Alfabeto

Nosso código JavaScript é escrito com base no **padrão Unicode**, que podemos sucintamente definir como um padrão que permite aos computadores representar e manipular, de forma consistente, texto de qualquer sistema de escrita existente. Este é composto com mais de 107 mil caracteres!

JavaScript não é JAVASCRIPT nem javascript!

A linguagem JavaScript **faz diferenciação** de letras maiúsculas de letras minúsculas, portanto, todo o devido cuidado deve ser tomado para evitar discordâncias na chamada de funções e todas as propriedades próprias do JavaScript e outras implementadas por você.

```
1 var agora = 'teste minúsculo';
2
3 console.log( Agora ); // -> ReferenceError: Agora is not defined
4 console.log( agora ); // -> 'teste minúsculo'
```

Nesse código acima, quando testado no Devtools do Chrome, por exemplo, terá como resultado os valores na frente da seta `->`.

Veremos como comentar nosso código agora.

“Rodinhas”, digo, Comentários no Código

Existe uma grande discussão de até que ponto é interessante um código muito comentado. Uns defendem dizendo que tal prática ajuda a esclarecer o que está ali feito, ajudando no futuro quando você revisitar tal seção, mas por outro lado, o lado *prático*, vemos que muito comentário simplesmente quebra o fluxo de leitura do código, fazendo com que você tenha que ficar saltando entre todos os textos, ou tendo que ler centenas/milhares de linhas a mais do que o necessário. Sem contar que quando o código é atualizado, o comentário também deve ser, e um comentário desatualizado é infinitas vezes pior do que a falta do mesmo.

Pense comigo, o que estamos fazendo já não é escrever em uma linguagem? Tudo bem, eu sei que a mesma é diferente da nossa linguagem usada para comunicação interpessoal, porém, ela por si só deve ser **auto-explicável**. Como conseguir isso? Nomeando suas funções de forma inteligente por exemplo, definindo no nome exatamente o que ela faz. E ai vem outra dica, código modular é **vital** para o sucesso de um programa. E para ter algo modular, esse algo deve fazer uma, e apenas **uma** tarefa. Conseguindo deixar um pedaço isolado de código fazendo apenas **uma** coisa, será mais fácil nomear essa coisa, fazendo com que o nome já se auto-explique.

Outra forma é criar uma documentação para seu código separado do mesmo, para evetuais consultas. Isso é algo bem elegante, diga-se de passagem.

Você vai me dizer: “Cara, você não acha que está pegando pesado demais não?! Nem me mostrou código e já está me mandando modularizar?!”

Ai meu amigo que é a parte boa! Sendo modular, você terá que fazer menos coisas! E assim irá ficar mais fácil desde o início da sua vida de programador =)

É muito importante começar a trabalhar as ideias na sua cabeça antes de serem implementadas. O **conceito** é tão importante quanto a **prática**, para quem quer realmente **dominar** o que está fazendo. E precisamos primeiro *tentar* criar algo, e depois de várias tentativas e erros, iremos conseguir fazer tal coisa da forma mais correta.

Voltando para os comentários, as duas formas que temos de inseri-los nos nossos códigos são:

- Usando duas barras: `//`
- Usando `/* */`

Vamos ver um exemplo:

```
1 /* Essa parte do meu código é comentada
2 e com esse sinal eu posso fazer comentários
3 em várias linhas...
4 */
5 var valorX = 13;
6
7 // Com esse símbolo, o comentário fica restrito a esta linha
```

O termo do título dessa seção, *rodinhas*, se refere às rodinhas quando estamos aprendendo a andar de bicicleta. No início, são super importantes, mas depois que aprendemos a andar, não precisamos mais delas. Usaremos comentários nos nossos códigos para fins **didáticos**, mas os mesmos devem ser usados de forma muito escassa, pois de fato não são necessários.

> “Se você precisa explicar muito detalhadamente alguma parte do seu código, é porque algo ali não está muito correto…”
>
> - Provérbio Chinês

Como Escrever Meu Código

Ao longo do livro, você irá aprender de forma prática as *melhores práticas* de escrita de código, mas inicialmente você apenas precisa saber que seu código JavaScript pode conter qualquer quantidade de espaços em branco entre os sinais delimitadores do programa e, por conseguinte, também ignora quebra de linhas.

```
1 // código válido
2 var       vazio =     'muito espaço vazio aqui'    ;
```

Tome cuidado com as quebras de linhas, pois como veremos no fim deste capítulo, em determinadas situações, o seu código pode ter um ponto e vírgula (`;`) adicionado pela própria linguagem, para prevenção de erros, porém que de fato irá gerar outros inesperados (mas que agora são esperados pois eu te falei que vão acontecer =P).

Para que isso não aconteça, tome cuidado com as quebras de linhas e com a omissão do ponto e vírgula. Por favor, seja *caprichoso* com seu código, não é porque você **pode** fazer algo que você **deve** fazer.

Sempre use Ponto e Vírgula `;`

Ao final de suas instruções, você ~~não~~ tem a possibilidade de omitir o ponto e vírgula (`;`), símbolo esse que mostra que uma *instrução* terminou.

O problema é que, como falado anteriormente, isso pode gerar vários erros (in)**esperados**. Mas fica a seu critério, pra que facilitar se podemos complicar, né?!

Cartório JavaScript

Sim, aqui iremos falar da parte mais “pesada” desse capítulo, que foi bem leve, então de fato essa é a parte *menos leve*. Qual parte é essa? Na verdade são duas: **identificadores** e **palavras reservadas**.

Identificadores

Um identificador nada mais é do que um nome que usamos para chamar nossas variáveis e funções. Sim, iremos registrar o seu nome na certidão, ou no programa como queira, para que futuramente quando esses meninos e meninas crescerem ainda nos lembremos deles e eles de nós.

Temos apenas algumas restrições para criar nossos identificadores, que são elas:

- Um identificador deve começar com alguma letra, $ ou _ (underscore);
- Números só são permitidos após o primeiro caractere estar de acordo com a regra anterior.

Exemplos de nomes válidos: `carro`, `_teste`, `v8`, `$nomeValido`.

Palavras Reservadas

Essas palavras são as que já estão definidas na linguagem, portanto apenas as **usaremos**. Confira abaixo algumas destas palavras:

```
break
case catch continue
debugger default delete do
else
finally for function
if in instanceof
new
return
switch
this throw try typeof
var void
```

```
12 while with
```

Na ECMAScript 5 temos mais palavras reservadas para uso futuro em novas implementações do padrão, que são:

```
1 class
2 enum export extends
3 import
4 super
```

No `strict mode` (modo estrito da linguagem, veremos futuramente do que se trata), temos mais algumas palavras reservadas para futuras implementações

```
1 implements interface
2 let
3 package private protected public
4 static
5 yield
```

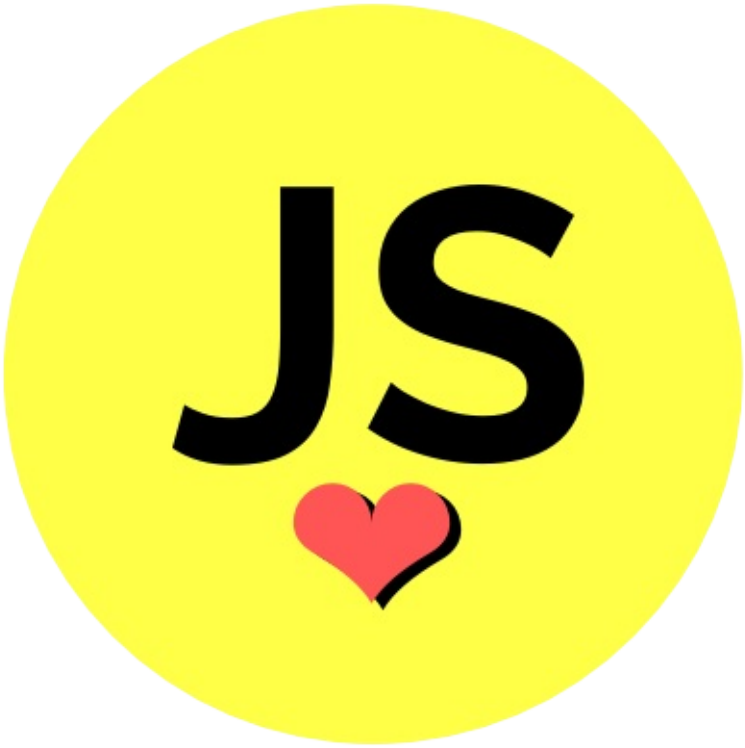
Veja mais sobre palavras reservadas aqui.

Conclusão

Neste capítulo nós aprendemos:

- que JavaScript usa o padrão Unicode para seus caracteres;
- que JavaScript diferencia letras maiúsculas de minúsculas;
- maneiras de comentar, e por que *evitar* comentar;
- que devemos **sempre** usar ponto e vírgula (`;`);
- como criar identificadores e como usar os existentes (palavras reservadas).

O Valor dos Tipos de Operadores - Parte 1



O título deste capítulo é uma representação dos três assuntos que iremos aprender: **Valores**, **Tipos** e **Operadores**.

Na primeira parte, iremos ver sobre:

1. Tipos de Dados
2. Tipo Number
3. Operadores Matemáticos
4. Números Especiais
5. Strings

Tipos de Dados

O que fazemos programando computadores é basicamente *manipular* dados. No JavaScript, chamamos esses dados de **valores**. Estes valores são divididos em dois **tipos**:

- **Tipos Primitivos**
    - `number` (números)
    - `string` (sequência de caracteres/texto)
    - `boolean` (booleanos ou valores de verdade)
    - **Tipos Especiais**: `null` (nulo) e `undefined` (indefinido)

Todos os valores em JavaScript diferentes dos citados acima, são pertencentes ao tipo

`object` (objeto). Vejamos quais são eles:

- **Tipos de Objetos**
    - arrays (vetores)
    - funções
    - Date (data)
    - RegExp (expressões regulares)
    - Error

Bom, agora que vimos teoricamente os tipos de dados em JavaScript, vamos perceber tudo isso de forma **real**. Para isso, você deverá abrir o console do seu navegador (no Google Chrome você pode usar o atalho `ctrl shift j`.

Dentro da seção *console* existente nas *ferramentas do desenvolvedor*, você deverá digitar os comandos abaixo, para perceber que tudo isso que te falei ali em cima é verdade. O resultado esperado está indicado na frente da seta `->`. Caso o resultado não seja o que você conseguiu, certifique-se de ter digitado corretamente o código.

Ah, já ia me esquecendo… **Digite TODO o código, não copie apenas!!! Isso será muito importante para a memorização e internalização dos conceitos.**

ps: Não acredite em meus resultados, faça todos os testes! =)

Quando estamos **dentro** do console, a utilização do comando `console.log()` é desnecessária para visualizar o resultado de uma instrução, pois o resultado da operação sempre é retornado.

Vamos utilizar o console diretamente apenas para códigos curtos, pois para códigos maiores, iremos rodá-los através de nossa página html, e ai o comando `console.log()` será de grande utilidade.

```
/* verificação de tipos primitivos */
typeof 13 // tipo number
// -> "number"
typeof "javascript furtivo" // tipo string
// -> "string"
typeof true // tipo boolean
// -> "boolean"

/* tipos primitivos especiais */
var semValor = null; // tipo null
typeof semValor
// -> "object"

var semAtribuir; // tipo undefined
typeof semAtribuir
// -> "undefined"
```

Olhando o código acima, mais precisamente na parte de *tipos primitivos especiais*, você pode perceber que o *tipo* `null` na verdade é um `object`. Bom, mas por que isso?

Essa peculiaridade do JavaScript na verdade é um *"erro"* dos primóridios da linguagem que foi herdado pela ECMAScript. Teremos um tópico específico mais a frente para explicar essa questão.

Vamos ver agora no nosso console os tipos existentes de objetos.

```
/* Tipos de Objeto */

// objeto do tipo array (vetor)
var guitarras = [ 'ibanez', 'music man', 'suhr' ];
typeof guitarras; // -> "object"

// objeto do tipo function
var soma = function( valor1, valor2 ) { return valor1 + valor2; }
typeof soma; // -> "function"

// objeto do tipo Date
var agora = new Date();
typeof agora; // -> "object"

// objeto do tipo RegExp
var minhaRegExp = /padrao/;
typeof minhaRegExp; // -> "object"

// objeto do tipo Error
var perigo = new Error( 'Alguma coisa deu errado!' );
typeof perigo; // -> "object"
```

Muito bom! Agora já conhecemos **todos** os tipos de dados do JavaScript! Você pode parar um pouco para respirar e beber uma água, pois a partir de agora iremos aprofundar um pouco em cada *tipo primitivo de dados* que acabamos de ver. Os *tipos de objetos* são mais complexos, e por isso cada um terá um livro próprio (no mínimo)…

Number

Representamos números no JavaScript da mesma forma que escrevemos no nosso dia a dia. `var numero = 13` é a forma que dizemos no JavaScript que a variável `numero` receceu o valor `13`.

Números fracionários são representados com um ponto, ficando assim a representação: `var numero = 1.3`.

Quando tivermos trabalhando com um número muito grande ou muito pequeno, usamos notação científica através da letra `e`, ficando nosso código, no caso, assim: `1.313e13` que é igual a **1.313 x 10^{13}**.

Cálculos com números inteiros menores que *9 quadrilhões* são garantidos de sempre serem precisos, porém não se pode dizer o mesmo para números fracionários, que devem ser tratados como **aproximações**.

Operadores e Operações

Agora que conhecemos um pouco mais sobre os números no JavaScript, vamos aprender a utilizar alguns operadores para então fazer alguns cálculos com estes números.

Temos os 4 operadores matemáticos básicos representados no JavaScript da seguinte forma:

- `+`: efetua a adição de dois números (ou concatenação, no caso de strings);

- `-`: efetua a subtração de dois números;
- `*`: efetua a multiplicação de dois números;
- `/`: efetua a divisão de dois números;
- `%`: efetua a divisão entre dois números, e retorna o **resto** da divisão.

Agora que já conhecemos os números e operadores, vamos brincar um pouco no console. Digite as operações abaixo e confira se o resultado é o mesmo do indicado no comentário.

```
1.3 + 1.8 // -> 3.1

1.3 - 1.17 // -> 0.13000000000000012

1.3 * 10 // -> 13

13 / 2 // -> 6.5

13 % 2 // -> 1
```

Números Especiais

Temos três tipos de números que não se comportam como números. `Infinity`, `-Infinity` e `NaN`. Os dois primeiros são gerados quando tentamos dividir algum número por zero, e o outro, **N**ot **a** **N**umber (`NaN`), surge quando há uma coerção de tipo de string para number, no caso, mal sucedida, pois a string não pode ser convertida em um número *de fato*.

Vamos ver como surgem estes números! De volta ao console! =)

```
// criando um número através do objeto Number
var test = Number( 'treze' );
test // -> NaN
typeof test // -> "number"

13 / 0 // -> Infinity

-13 / 0 // -> -Infinity
```

ps: Uma particularidade do `NaN` é que ele é o único valor em JavaScript que **não é** igual a ele mesmo. Como verificar isso? A **melhor** forma para saber se uma variável está com valor `NaN` é testando se ela é igual a ela mesmo (ou diferente).

Se você comparar a variável `A` com ela mesmo e o resultado for `false`, logo, `A` é igual a `NaN`. Vamos dar uma olhada mais de perto:

```
var A = NaN;

A == A; // -> false
A != A; // -> true
```

Você poderia também verificar se uma variável está com o valor `NaN` usando o método `isNaN()`, porém o mesmo apresenta alguns comportamentos inesperados por fazer coerção dos valores para o tipo *number*. Veja mais aqui.

Strings

Este é o tipo de dados básico usado para representação de texto. Para o programa identificar uma string, tal deve estar contida entre aspas duplas ou simples, assim:

- `"isso é uma string"` e
- `'isso também é uma string'`

ps: Geralmente, usamos aspas simples no JavaScript e aspas duplas no HTML.

Você pode colocar tudo dentro de uma string, mas alguns caracteres precisam ser colocados com uma certa atenção, como aspas, quebras de linha, tabulação entre outros, e toda a string deve ser escrita como uma simples expressão, apesar de poder ser escrita em várias linhas de código, como será demonstrado logo mais.

Para fazer o *escape* dos caracteres especiais, utilizamos o símbolo da barra invertida `\`. Esse sinal nos diz que temos algum caractere com significado especial na frase.

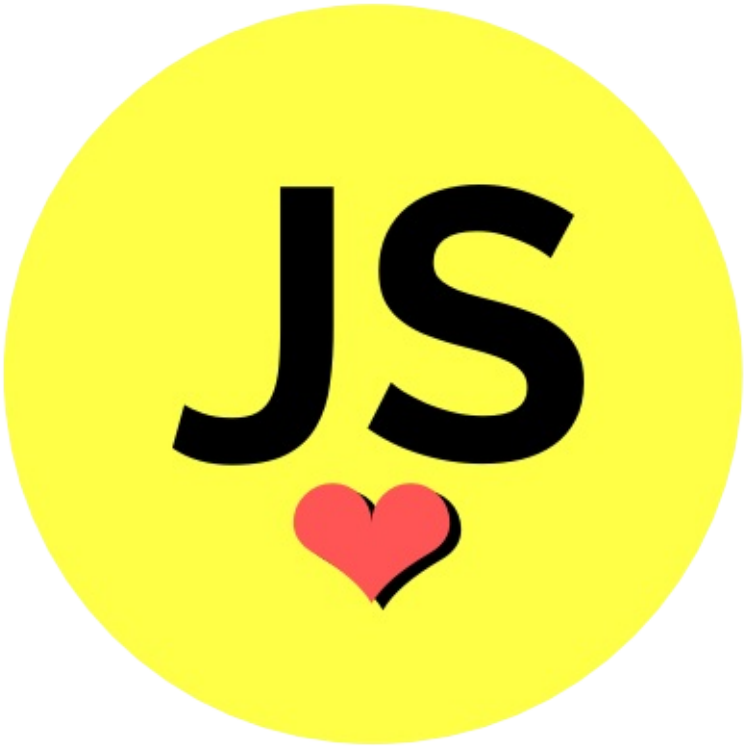
Vamos praticar um pouco essa “teoria das cordas”.

```
var
  frase01 = 'Este é o parágrafo inicial...',
  frase02 = "... complementado por este trecho.",
  frase03 = '\nA partir daqui começa a ficar interessante...',
  frase04 = '\tagora com tabulação, e \'aspas\' também',
  texto = frase01 + frase02 + frase03 + frase04;

texto;
/*
-> "Este é o parágrafo inicial..... complementado por este trecho.
A partir daqui começa a ficar interessante...        agora com tabulação, e 'aspas' tamb\
ém"
*/

// Para escrever o mesmo texto acima como
// uma sentença única, faríamos assim:
var texto2 = 'Este é o parágrafo inicial...\
... complementado por este trecho.\
\nA partir daqui começa a ficar interessante...\
\tagora com tabulação, e \'aspas\' também';

texto2;
```

ps:O caracter `\` quando colocado no fim da linha, permite que você continue a string de outra linha. Isso é útil para formatação do código, melhorando a sua legibilidade. Use a forma que for mais conveniente e agradável para você.

Vamos encerrar esta primeira etapa por aqui, para que não fique muita informação de uma só vez. No próximo capítulo iremos ver mais sobre operadores, booleanos, coerção de tipo e mais algumas coisas.

O Valor dos Tipos de Operadores - Parte 2



Continuando o último capítulo onde falavamos sobre **valores**, **tipos** e **operadores**, veja o que será visto neste capítulo logo abaixo:

1. Valores Booleanos
2. Valores *Truthy* e *Falsy*
3. Truques com o Operador `!`
4. Cuidado com a Falsidade!
5. Valores *undefined*
6. Operadores Unários
7. Operadores de Incremento
8. Operadores de Comparação
9. Coerção (ou conversão) de Tipo
10. Como Evitar a Coerção de Tipo
11. Operadores Lógicos
12. Curiosidades sobre os Operadores Lógicos
13. Operador Ternário
14. Regras de Precedência

Valores Booleanos

Esse **tipo** de dado, `boolean`, é bem específico, tendo apenas dois valores possíveis, `true` ou `false` (verdadeiro ou falso). Conseguimos obter estes valores através de operações de comparação (x é maior que y ? A resposta é sempre sim ou não, verdadeiro ou falso no

caso) e através de um “truque” utilizando o *operador unário* `!` (mais sobre operadores unários e comparações adiante).

Este operador `!` (negação) inverte o valor passado à ele para um valor booleano oposto ao original. Vou explicar melhor, acompanhe comigo:

Quando aplicamos o operador `!` à uma variável que tenha *realmente* um valor **considerável**, obtemos o valor `false` como retorno desta operação. Mas para ficar mais claro o que é um valor considerável, vou te explicar o que é um valor *descartável*.

Valores *Truthy* e *Falsy*

Agora você já precisa ter os conceitos de **valor**, **tipo** e **operador** em JavaScript bem consolidados. Faça um exercício mental de definir para você mesmo o que cada um é.

Todos os tipos de **valores** em JavaScript têm, intrínsicamente (em sua essência), um respectivo *valor booleano* associado. Estes valores são claramente perceptíveis quando fazemos *comparações* entre valores e quando utilizamos o operador unário de negação `!`.

Valores *falsy* (falsos) são os que tem em sua essência o valor booleano `false`. Estes valores são:

- `false`
- `0` (zero)
- `""` (string vazia)
- `null`
- `undefined`
- `NaN`

Todos os outros valores em JavaScript são considerados *truthy* (verdadeiros). Alguns valores *truthy* peculiares são:

- `"0"` (zero como string)
- `"false"` (false como string)
- `function() {}` (funções vazias)
- `[]` (arrays vazios)
- `{}` (objetos vazios)

Truques com o Operador `!`

Agora que sabemos **exatamente** quais valores booleanos cada valor do JavaScript carrega, podemos tranquilamente e **conscientemente** utilizar o operador `!` para obter o valor **contrário** do natural do valor em operação. Vamos aos exemplos, e portanto, **ao console**!:

```
1 !false        // -> true
2 !0            // -> true
3 !""           // -> true
4 !NaN          // -> true
5 !"0"          // -> false
```

```
6 !"false"       // -> false
7 !function() {} // -> false
8 !{}            // -> false
```

Um modo interessante de saber qual é o valor *truthy* ou *falsy* de um valor é negando-o duas vezes com o operador `!`. Sim, é como na Matemática, `-1 x -1 = 1`. Então temos que:

```
1 !!false           // -> false
2 !!NaN              // -> false
3 !!{}               // -> true
4 !!function() {} // -> true
```

Cuidado com a Falsidade!

Iremos ver ainda neste capítulo como comparar valores através dos operadores de comparação, mas é importantíssimo que eu lhe explique mais uma última coisa sobre os valores *falsy*.

As regras de comparação entre estes valores é um pouco *não-intuitiva*, logo ter o conhecimento dos valores esperados em determinados casos é crucial na hora de um eventual *bug*. Vamos ver quais são essas peculiaridades.

`false`, `0` e `""`

Quando comparamos dois destes três valores utilizando o operador ==, o resultado é sempre `true`. Vamos testar no console o código abaixo:

OBS: Quando utilizamos o console, podemos omitir o sinal `;` (ponto e vírgula) no fim das expressões, pois estamos fazendo apenas um **teste rápido**, porém no seu código JavaScript de fato, **sempre utilize o ponto e vírgula**!!!

```
1 false == 0  // -> true
2 false == "" // -> true
3 0      == "" // -> true
```

`null` e `undefined`

Os valores `null` e `undefined` somente são equivalentes a eles mesmos. Vejamos:

```
1 null == false            // -> false
2 null == null             // -> true
3 undefined == false      // -> false
4 undefined == undefined // -> true
5 undefined == null       // -> true
```

O último valor falsy que temos para citar é o `NaN`, porém este já foi abordado no capítulo anterior, e sabemos que é o **único** tipo de dado em JavaScript que não é igual a ele mesmo.

Valores Undefined

Temos definidos no JavaScript dois tipos indefinidos, usados para representar a falta de representatividade de algum dado.

ps: O paradoxo da frase anterior é apenas uma brincadeira, pois a definição é muito simples, e vou lhe explicar agora! =)

Como já foi explicado no capítulo anterior, na realidade apenas `undefined` é de fato um tipo primitivo de dado, `null` é um tipo de objeto, quando fazemos sua inspeção no console podemos verificar isso.

O valor `undefined` aparece quando declaramos uma variável e não atribuímos a ela nenhum valor. Já o valor `null` é um valor que deve ser atribuído a uma variável, e representa a ausência de valor nesta variável.

Confira os exemplos no capítulo anterior na seção *tipos de dados* para fixar esta diferença.

Operadores Unários

Alguns operadores são *palavras* ao invés de símbolos. Utilizamos um destes operadores muito nos exemplos do capítulo anterior, o operador `typeof`, que produz uma string identificando o tipo do valor do elemento que você passou a ele.

ps: Para ver o operador `typeof` em funcionamento, volte no capítulo anterior onde você encontrará vários exemplos da utilização dele.

Outros operadores unários são `delete` e `void`. Para este material não se tornar muito extenso, você pode procurar o que cada um faz aqui(delete) e aqui(void).

Operadores unários utilizam **um** valor para fazer seu trabalho, operadores binários e ternários, **dois** e **três** valores respectivamente.

Operadores de Incremento

Temos dois tipos de operadores de incremento: os utilizados para números e os utilizados para *strings* e números.

Incrementando Números

Os operadores `++` e `--` são utilizados para incrementar variáveis/valores do tipo `number`. São operadores frequentemente utilizados para auxiliar na estrutura lógica do programa, aumentando ou diminuindo em **uma unidade** o valor da variável. A partir do próximo capítulo iremos começar a focar em pedaços de código maiores, e assim iniciar a criação de *mini-programas*, para depois chegarmos no nosso objeto de juntar todas essas peças para criarmos o que quisermos. Por agora, vamos ver no console como funcionam estes operadores.

Uma última informação. Estes operadores podem ser utilizados **antes** ou **depois** da variável, sendo assim denominados operadores de **pré-incremento** ou **pós-incremento**. Qual a diferença entre eles? A diferença é que quando utilizamos o operador **antes** da variável, ela será alterada antes da execução do código, então o valor processado para a operação **atual** será o já modificado, e quando usamos o operador **depois** da variável, essa alteração será percebida apenas **depois** quando esta variável for solicitada.

Vamos aos testes!

```
var total = 0;
total++ // -> 0 (valor foi alterado, mas será percebido na próxima operação)
total   // -> 1
++total // -> 2
--total // -> 1
total-- // -> 1
total   // -> 0
```

Incrementando Números e Strings

Estes operadores serão utilizados **sempre** por você! São realmente MUITO importantes e elegantes, diga-se de passagem. A função deles é adicionar mais conteúdo a antiga variável mas sem precisar declarar isso de forma *verbosa* (escrevendo mais do que se poderia/deveria). De fato, estes operadores **operam** e **atribuem** o valor para a mesma variável. Ele pode ser utilizado para *somar*, *subtrair*, *multiplicar* e *dividir* números, ou para *concatenar* novas strings em variáveis que contém valores deste tipo.

Vamos dar uma olhada como estes operadores trabalham.

```
// somando números
var resultado = 7
resultado += 6 // -> 13

// concatenando strings
var meuNome = 'Eric'
meuNome += ' Douglas' // -> "Eric Douglas"

// subtraindo números
resultado -= 6 // -> 7

// multiplicando números
resultado *= 3 // -> 21

// dividindo números
resultado /= 2 // -> 10.5

// resto da divisão de números
resultado %= 2 // -> 0.5

// um exemplo de como somar na mesma variável
// sem usar estes operadores
resultado = resultado + 0.8 // -> 1.3
```

Operadores de Comparação

Operadores de comparação são um dos tipos de *operadores binários*, que no caso, utilizam dois valores para efetuarem a operação. Os operadores de comparação sempre retornam um *booleano*, dizendo se a comparação é verdadeira ou falsa. São também utilizados na estrutura lógica do programa.

Vamos conhecer estes operadores!

< : **menor que -** verifica se o número da esquerda é menor que o número/string da direita
<= : **menor ou igual que -** verifica se o número da esquerda é menor ou igual ao

número/string da direita > : **maior que** - verifica se o número da esquerda é maior que o número/string da direita >= : **maior ou igual que** - verifica se o número da esquerda é maior ou igual ao número/string da direita

Cuidado ao Comparar Strings!

A maneira de comparar strings pode não ser muito intuitiva, pois qualquer letra *maiúscula* será sempre **menor** que uma letra *minúscula*.

Vamos confirmar toda essa teoria agora. Já sabé né? Console! =)

```
13 < 26 // -> true

'treze' < 'Vinte seis' // -> false *preste atenção aqui*

var treze = 13
treze <= 13 // -> true

26 > 13 // -> true
'vinte seis' > 'Treze' // -> true
```

Comparadores de Igualdade

Além dos comparadores mostrados acima, também temos os *comparadores de igualdade*, que são constantemente usados em nossos códigos, porém tem algumas peculiaridades que se você não souber, **certamente** irá gerar erros nos seus programas. Mas fique tranquilo, é bem simples de entender a diferença entre eles, e irei lhe provar agora! Primeiro, vamos conhecer quais são estes outros operadores.

==: testa igualdade `!=`: testa desigualdade ===: testa igualdade de forma **restrita** `!==`: testa desigualdade de forma **restrita**

A grande diferença entre um operador e outro é que a primeira dupla (== e `!=`) ao comparar os valores, caso eles não sejam do mesmo **tipo**, procedem com uma particularidade do JavaScript chamada de **Coerção de Tipo** (ou conversão de tipo), e isso pode gerar **muita** dor de cabeça para você. Sério!

Irei explicar sobre coerção de tipo no próximo tópico, mas antes vamos tentar entender pelo código, de forma prática, como cada operador trabalha, e depois irei explicar de forma teórica e encerrar o assunto. Vamos lá!

```
'13' == 13 // -> true
'13' != 13 // -> false

'13' === 13 // -> false
'13' !== 13 // -> true
```

Vale ressaltar mais uma vez… Entender o que está acontecendo no código é super importante, e muitas vezes temos que realmente ler várias vezes e/ou ficar vários minutos para conseguir entendê-lo. E sim, isso é algo super importante, não tenha “dó” de **investir** tempo nisso!

Neste próximo tópico irei lhe explicar o que aconteceu no código anterior, então, vamos para o próximo!

Coerção de Tipos

O nome assusta um pouco né?! E de fato devemos ter cuidado com isso pois surgem muitas ~~pérolas~~ bugs no seu código a partir deste esforço que o JavaScript faz para efetuar todas as instruções recebidas.

Sempre que você usa um operador no JavaScript (no caso então, faz uma operação), você irá receber um valor de retorno, nunca um erro, mesmo que você tente trabalhar com tipos diferentes de dados.

A **coerção de tipos** (ou **conversão** de tipos) é justamente isso! Quando tentamos fazer operações com tipos de dados **diferentes**, o JavaScript nos agracia com a conversão de um dos valores para algum outro tipo… Só que o grande e terrível problema disso é que dificilmente você consegue prever qual será este novo valor, pois as regras para tal conversão **não** são intuitivas, o que pode causar alguma(s) falha(s) em sua aplicação.

Como Evitar a Coerção de Tipos

Para evitar toda essa preocupação, você **deve** usar os operadores `===` e `!==`, que fazem a mesma verificação de igualdade que seus "primos", porém **NÃO** fazem coerção de tipos!

ps: Por questões de segurança, sempre opte por usar os comparadores de igualdade **restrita**.

Parabéns! Agora você finalmente sabe o que é essa bendita *coerção de tipos* e como evitá-la! lol

Quando algo que não é obviamente um número é convertido para tal, o valor dessa operação é convertido para `NaN`. Portanto, sempre que encontrar `NaN` em seu código, procure por coerções de tipo acidentais.

Operadores Lógicos

Sempre que pensamos na parte lógica do programa devemos pensar em valores booleanos, e para gerar estes valores booleanos a partir dos valores que nossas variáveis carregam, iremos usar os *operadores lógicos*, que são um tipo de *operador binário*, ou seja, necessitam de dois valores para geração de um terceiro.

ps: O operador `!` é um operador lógico pois retorna um valor booleano, porém é uma exceção a regra por ser um operador *unário*.

Conheça agora os operadores lógicos:

- `&&` : este operador é chamado de "**E**" lógico
- `||` : este operador é chamado de "**OU**" lógico
- `!` : este operador é chamado de "**NÃO**" lógico

`&&` E lógico

Este operador lógico e binário retorna `true` se ambos os valores (ou operandos) forem verdadeiros. Isso no caso será utilizado em estruturas condicionais, assunto que iremos

abordar no próximo capítulo. Por enquanto você só precisa saber disso.

`||` OU lógico

Este operador lógico e binário retorna `true` se um dos valores forem verdadeiros.

`!` NÃO lógico

Como já vimos anteriormente, este operador transforma o operando em booleano, porém com a peculiaridade de inverter seu valor *truthy/falsy* para um booleano de fato.

Curiosidades sobre os Operadores Lógicos

Quando utilizamos os operadores lógicos **fora** de estruturas condicionais, eles tem uma forma peculiar de trabalhar, que vale muito a pena ser entendida para que possamos deixar nosso código mais elegante e conciso.

Sempre que usamos `&&` e `||`, eles convertem o valor do lado esquerdo para um booleano, para saberem qual valor será retornado dessa operação.

Entendendo o Operador `||`

O operador `||` retorna o valor da sua esquerda quando ele pode ser convertido para `true`, ou seja, quando seu valor é do tipo *truthy*, e sempre retorna o valor da direita caso contrário, independente de qual valor seja esse, até mesmo outro valor *falsy*. Veja isso na prática.

```
var esquerda = '' // -> valor do tipo falsy
var direita = 13 // -> valor do tipo truthy

esquerda || direita // -> 13

var direita = 0 // -> agora vamos atribuir este valor falsy na direita e ver o qu\
e acontece

esquerda || direita // -> 0

// Agora vamos deixar ambos os valores truthy

esquerda = 13
direita = 31

esquerda || direita // -> 13
```

Viram? Mesmo o valor da direita sendo *falsy*, ele é retornado pois a regra é: “se o valor da esquerda for falsy, retorne o da direita sem nem ver qual é”. E depois, quando alteramos o valor da esquerda para um valor *truthy*, ele foi retornado. Legal né?! =)

Vamos ver agora o outro operador.

Entendendo o Operador `&&`

Este operador faz o contrário. Quando o valor da esquerda é *falsy*, ele é retornado, independente de qual valor seja o da sua direita. E o valor da direita **sempre** será retornado quando o da esquerda for *truthy*, mesmo que este valor da direita seja *falsy*.

Vamos ver o código para ficar mais claro.

```
var esquerda = NaN // -> valor do tipo falsy
var direita = 13 // -> valor do tipo truthy

esquerda && direita // -> NaN

var esquerda = 31 // agora vamos atribuir um valor truthy na esquerda e ver o que\
 acontece

esquerda && direita // -> 13

// Agora vamos deixar o valor da direita falsy

direita = ''

esquerda && direita // -> ''
```

Muito bom! Estamos quase terminando, e a essa altura você já sabe de várias peculiaridades do JavaScript! Vamos em frente =)

Operador Ternário

Este operador é de fato muito poderoso, pois evita verbosidade no código, deixando-o então bem mais elegante.

Ele é chamado *operador ternário* pois envolve três peças em sua operação. Vamos analisá-lo abaixo. Primeiramente irei mostrá-lo em funcionamento e depois irei explicá-lo.

```
var usuario = ''

// usando o operador ternário
usuario ? usuario : 'convidado' // -> "convidado"

// atribuindo um valor a variável usuario
usuario = 'Eric'

// usando o operador ternário novamente
usuario ? usuario : 'convidado' // -> "Eric"
```

Como funciona isso? Lhe explico já!

Primeiramente criamos a variável `usuario` e atribuimos uma string vazia a ela. Ou seja, um valor *falsy*. Após isso, usamos o operador ternário que funciona da seguinte forma:

1. Indicamos a variável ou condição que deve ser avaliada. No caso, usamos a variável `usuario`.
2. Este valor a ser analisado será transformado em um booleano, a partir de seu valor *truthy/false* (está vendo a importância da teoria?!)
3. Caso seu valor seja *truthy*, então o operador retorna a instrução **após** o sinal `?`. Caso o valor seja *falsy*, o valor retornado será o que está **após** o sinal `:`.
4. Você pode retornar qualquer expressão JavaScript como valor de retorno dessa avaliação, veja outro exemplo:

   var cozinheiro = false

   cozinheiro ? { receitas : [‘carnes’,’doces’,’tortas’] }

```
console.log('mexido e olhe lá!') // -> "mexido e olhe lá!"
```

Entendeu como funciona? Esse operador é **muito** legal! =D

ps: Altere o valor de cozinheiro para `true` e veja o que acontece!

Regras de Precedência

Para saber qual operação irá ocorrer primeiro, você precisa conhecer a ordem de precedência dos operadores. Confira abaixo a lista na ordem do maior (no topo) para o menor (embaixo) operador em relação a sua precedência.

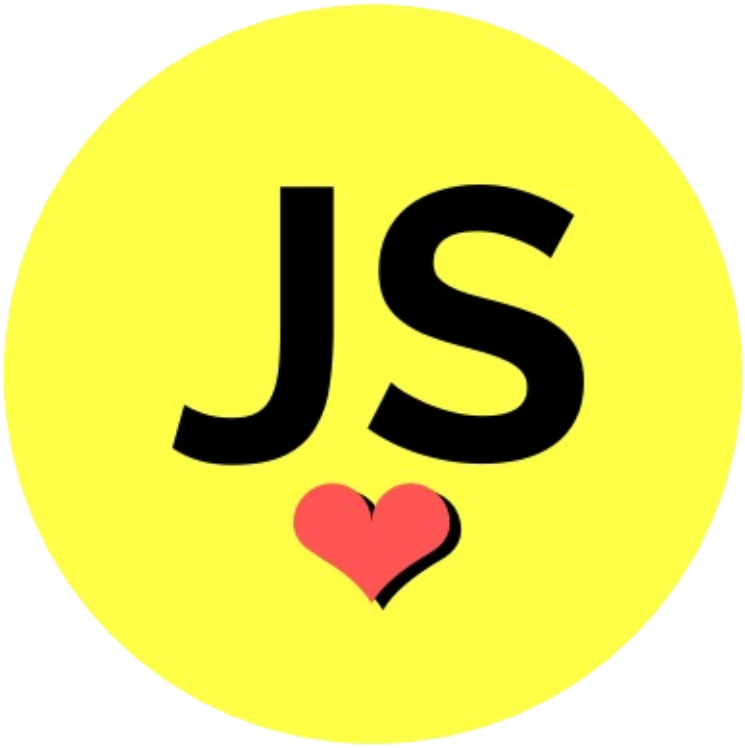
- `++`, `--` pré/pós incremento/decremento
- `!` altera valor booleano
- `typeof`, determina o tipo do operando
- `*`, `/` e `%` multiplica, divide e resto
- `+` e `-` soma e subtrai
- `+` concatena strings
- `<`, `<=`, `>` e `>=` comparação de ordem numérica
- `<`, `<=`, `>` e `>=` comparação de ordem alfabética
- `==` teste de igualdade
- `!=` teste de desigualdade
- `===` teste de igualdade restrita
- `!==` teste de desigualdade restrita
- `&&` E lógico
- `||` OU lógico
- `?:` operador ternário
- `=` atribuição à variável ou propriedade
- `*=`, `/=`, `%=`, `+=` e `-=` operação e atribuição

Veja mais sobre regras de precendência aqui

Conclusão

Finalmente terminamos! Bom, esse foi o maior capítulo até agora, e com certeza foi o mais interessante, por termos visto várias peculiaridades da linguagem JavaScript e termos sedimentado uma base sólida para os capítulos seguintes.

Expressões JavaScript



Vamos continuar com nossa longa jornada, rumo à maestria em JavaScript! Neste capítulo falaremos exclusivamente sobre **expressões**. Confira um resumo do que será abordado:

1. O que é uma Expressão?
2. Expressões Básicas
3. Expressões “Complexas”
    - Expressões de Inicialização
    - Expressões de Definição
    - Expressões para Acessar Propriedades
    - Expressões de Invocação
    - Expressões de Criação
4. Mais Expressões
    - Expressões Matemáticas
    - Expressões de Relação
    - Expressões Lógicas
    - Expressões de Atribuição

O que é uma Expressão

Expressão é um fragmento de código JavaScript que pode ser interpretado para produção de um valor. Fazendo uma breve analogia, a expressão está para seu programa JavaScript

assim como uma frase está para um texto. Um exemplo de expressão é uma atribuição de valor à uma variável.

```
var atribui = 'e a névoa começa a se dissipar...'
```

Podemos combinar expressões para formar outras mais elaboradas.

Vimos várias expressões no capítulo anterior, enquanto falávamos sobre os operadores. Todos aqueles exemplos são ótimos para o entendimento de expressões. Na realidade, o que iremos ver hoje é apenas uma formalização do estudo passado, apenas nomeando coisas que já sabemos intuitivamente.

Aproveite pois o capítulo de hoje será bem leve, mas fique sabendo que nossa base para criação de programas JavaScript está quase pronta, e em breve a diversão irá começar de fato ;D

Expressões Básicas

Esse tipo de expressões são assim denominadas pois não incluem outras expressões, ou seja, são o menor fragmento possível do nosso código. Vejamos alguns exemplos:

```
// Expressões Literais
13
"JavaScript Furtivo"

// Algumas Palavras Reservadas
true
false
null

// Referências à variáveis
total
i
```

Expressões de Inicialização

Essas expressões são **muito** importantes, pois nos possibilitam a criação de *objetos* e *arrays* de forma *literal*, e isso nos dá uma facilidade e praticidade muito grande na hora de tais tarefas. Veja abaixo a forma de se criar um objeto e array de forma literal e da forma utilizada com o auxílios de seus *construtores* (iremos conhecer mais sobre construtores futuramente).

```
/* Inicializando de forma literal  */
var meuObjeto = {}; // criado um objeto vazio
var meuArray = []; // criado um array vazio

/* Inicializando com Construtores  */
var meuObjeto2 = new Object();
var meuArray2 = new Array();

// Quando criamos objetos/arrays vazios, podemos omitir os parênteses
var objetoVazio = new Object;
var arrayVazio = new Array;
```

Viram o quão econômica e elegante é a forma literal de inicialização de objetos e arrays? Use-a sem moderação!

Um dos motivos de enquadrarmos esta e as expressões seguintes no campo de *expressões*

complexas, é que elas podem receber as expressões básicas para comporem seu valor, pois obviamente, um objeto, um array, só são úteis por carregarem em si valores agrupados de acordo com algum propósito. Além de valores básicos, podemos também passar valores "complexos" para expressões complexas. Traduzindo, ter arrays dentro de arrays, objetos dentro de objetos, arrays dentro de objetos, objetos dentro de arrays, e por ai vai…

Vamos agora então, dar sentido a existência de nossos objetos e arrays!

```
var estudos = {
   JavaScript: [
      'NodeJS',
      'AngularJS',
      'ExpressJS',
      'MongoDB'
   ],
   Outros: [ 'Jade', 'Stylus' ]
};

var livros = [
   'Padrões JavaScript',
   'JavaScript: O Guia Definitivo',
   'O Melhor do JavaScript'
];
```

No exemplo acima, na variável `estudos` (que é um *objeto*, veja pelo sinal `{}` que *abraça* os demais valores), temos as *propriedades* `JavaScript` e `Outros` recebendo arrays como valores, e estes arrays recebem *strings*, por sua vez.

No caso da variável `livros`, que é um array `[]`, ela recebe três *strings* como itens de sua lista de valores.

Brinque um pouco agora criando arrays e objetos de diferentes tipos, preenchidos com diversos valores!

Expressões de Definição

Usamos essa expressão para definir uma função em nosso programa. Para criarmos essa expressão devemos seguir os seguintes passos:

1. Usar a palavra chave `function`
2. [opcional] Dar um nome para a função (caso não tenha nome, iremos chamá-la de *função anônima*)
3. Passar os argumentos que serão usados na função, que podem ser de 0 (zero) até quantos você quiser, todos colocados dentro dos parênteses `()` e separados por vírgulas. Ex: `(valor1, valor2, valor3)`
4. Criação do *corpo da função*, que conterá outras expressões, que são as *tarefas* que deverão ser feitas todas as vezes que a função for chamada. O corpo da função é delimitado por chaves `{ ... mais expressões aqui ... }`

Para chamarmos nossa função posteriormente, devemos ou atribuí-la a uma variável ou então nomeá-la.

Veja alguns exemplos:

```
function nomeada() {
  // outras expressões aqui
}

var outraFuncao = function(parametro1, parametro2) {
  // mais expressões aqui
}
```

Expressões para Acessar Propriedades

Usamos esse tipo de expressão para obter o valor de alguma propriedade/item de um objeto ou um array. Podemos fazer isso usando dois tipos de sintaxe diferentes, a de notação por ponto (`expressao.chave`) ou a de notação por colchetes (`expressão['chave']`). Vamos ver com mais detalhes cada uma delas.

Acessando Propriedades com ponto

`expressão.chave`

Essa forma é utilizada exclusivamente para acessar *propriedades* e *métodos* de **objetos**. Onde temos escrito *expressão*, iremos utilizar o nome do nosso objeto, e onde temos *chave*, iremos passar o nome da propriedade/método que queremos avaliar. Vamos para o console e tornar isso mais claro.

```
var guitarras = {
  modelo1: 'music man',
  modelo2: 'ibanez',
  modelo3: 'prs',
  cordas: [ 'elixir', 'daddario' ]
};

// Se quisermos então saber o valor do modelo2...
guitarras.modelo2 // -> 'ibanez'

// O valor do segundo item do array "cordas"
guitarras.cordas.2 // -> SyntaxError: Unexpected number
```

Notem que tivemos um erro ao tentar acessar o segundo item da propriedade cordas! Isso se deve pelo motivo que para acessar itens de arrays, utilizamos outro tipo de notação, que irei lhe explicar agora!

Acessando Itens e Propriedades com Colchetes

`expressão['chave']`

Com esse tipo de notação podemos acessar tanto os itens de um array quanto as propriedades de um objeto. Onde temos o valor *expressão* iremos substituir pelo nome do array ou objeto, e onde temos chave, iremos substituir pela posição do item (caso seja um array) ou pelo nome da propriedade, caso seja um objeto. Vamos agora utilizar do nosso exemplo anterior, porém acessando o objeto e o array com esta notação por colchetes.

```
var guitarras = {
  modelo1: 'music man',
  modelo2: 'ibanez',
  modelo3: 'prs',
  cordas: [ 'elixir', 'daddario' ]
```

```
};

// Se quisermos então saber o valor do modelo2...
guitarras['modelo2'] // -> 'ibanez'

// O valor do segundo item do array "cordas"
guitarras.cordas[1] // -> 'daddario'

// uma peculiaridade dessa notação: podemos passar um valor
// contido em uma variável como a chave para buscar um item/propriedade
var modeloPreferido = 'modelo1';
guitarras[modeloPreferido] // -> 'music man'
```

Dicas de “Acessibilidade”

- Caso você tente acessar propriedades em `null` e `undefined`, você obterá uma exceção *TypeError*, ou seja, um erro, pois esses valores não podem ter propriedades.
- Caso acesse uma propriedade ou índice que não esteja definido, o valor retornado será `undefined`.
- A notação de ponto, por ser mais direta, deve ser utilizada quando já se sabe exatamente qual propriedade será acessada.
- A notação por colchetes permite que seja passado variáveis como valor para acesso, pois caso a propriedade tenha sua *chave* (identificador) sendo um número, contendo espaço no nome ou sendo uma palavra reservada de JavaScript, a única forma de acessá-la é assim, através da notação de colchetes, utilizando ou não uma variável para *contenção* desse valor.

Expressões de Invocação

Essa expressão é utilizada para chamar (invocar, executar) uma função ou *método*.

Quando uma função está atribuída à uma chave em um objeto, essa passa a se chamar **método**.

Para usar essa expressão devemos indicar o nome da função que desejamos invocar seguida de parênteses. Caso a função receba argumentos, os mesmos devem ser passados dentro dos parênteses.

ps: Lembre-se de usar no seu código o `;` depois de chamar a função! Somente omitiremos o ponto e vírgula no console para agilizar os testes.

Vamos ver um exemplo:

```
// definindo uma função: expressão de definição
function multiplica( valor1, valor2 ) {
  return valor1 * valor2;
}

// chamando a função: expressão de invocação
multiplica( 5, 8 ); // -> 40
multiplica( 3, 13 ); // -> 39
```

Entendendo o Processo de Invocação

Irei explicar sucintamente como funciona este processo, o qual será devidamente aprofundado quando estivermos falando especificamente sobre funções. Veja quais são as etapas:

1. Verificação da existência de uma função (ou objeto, visto que a função é um tipo de objeto) com o nome passado. Caso não exista tal objeto, um erro é lançado.
2. Criação de uma lista de valores a partir dos argumentos passados, que são avaliados para produzirem tais valores.
3. Atribuição dos valores aos respectivos parâmetros da função.
4. Execução do corpo da função.
5. Se a função tiver `return`, esse valor será o retornado como resultado da avaliação da função.

Expressões de Criação

São utilizadas como uma alternativa na forma de se criar objetos. É bom ter conhecimento dessas expressões, mas saiba que dificilmente você irá utilizá-las (para arrays e objetos), pois a forma literal é bem mais direta.

Nessas expressões utilizamos as *funções construtoras* para criar novos objetos, juntamente com a palavra chave `new` (iremos estudar sobre funções construtoras mais a frente).

Alguns exemplos da utilização dessas expressões:

```
1 var cestaDeCompras = new Object();
2 var dataDeAgora = new Date();
3 var listaDeCompras = new Array();
```

Mais Expressões

No último capítulo já vimos vários exemplos de expressões que utilizam operadores. Irei deixar aqui indicado o nome da expressão com seus respectivos operadores, e após vista a teoria aqui, vá para o outro capítulo e procure as respectivas expressões em uso.

Os outros tipos de expressões que temos são:

- **Expressões Matemáticas ou Aritméticas**: `+`, `-`, `/`, `*`, `%`
- **Expressões Relacionais**: `==`, `===`, `!=`, `!==`, `<`, `>`, `<=`, `>=`
- **Expressões Lógicas**: `&&`, `||`, `!`
- **Expressões de Atribuição**: `+=`, `-=`, `*=`, `/=`, `%=`