

# Uma reintrodução ao JavaScript (Tutorial de JS)

## Introdução

Por que uma reintrodução? Porque [JavaScript](#) é conhecida como [a mais incompreendida linguagem de programação do mundo](#). Embora muitas vezes ridicularizada como um brinquedo, por baixo de sua simplicidade enganosa estão alguns recursos poderosos da linguagem, que agora é usado por um número incrível de aplicações de alto nível, mostrando que o conhecimento mais profundo desta tecnologia é uma habilidade importante para qualquer desenvolvedor web, mobile ou desktop.

É sempre bom começar com a história da linguagem. A JavaScript foi criada em 1995 por Brendan Eich, um engenheiro da Netscape, e lançada pela primeira vez com o Netscape 2 no início de 1996. Foi inicialmente chamada de LiveScript, mas logo foi rebatizada, em uma decisão de marketing malfeita, para tentar crescer sobre a popularidade da linguagem Java da Sun Microsystems - apesar das duas terem muito pouco em comum. Esta tem sido uma fonte de confusão desde então.

A Microsoft lançou uma versão compatível com a maior parte da linguagem, chamada de JScript, junto com o IE, três meses mais tarde. A Netscape apresentou a linguagem a Ecma International, uma organização europeia de normalização, o que resultou na primeira edição do padrão ECMAScript em 1997. O padrão recebeu uma atualização significativa com o ECMAScript Edição 3 em 1999, e manteve-se praticamente estável desde então. A quarta edição foi abandonada, devido a diferenças políticas relativas à complexidade da linguagem. Muitas partes da quarta edição formam a base da nova edição ECMAScript 5, publicado em dezembro de 2009.

Esta estabilidade foi uma grande notícia para os desenvolvedores, pois isto proporcionou que várias implementações em JavaScript tivessem muito tempo para se firmar. Eu vou focar quase exclusivamente no dialeto da edição 3. Para que seja fácil se familiarizar, vou utilizar o termo JavaScript por todo o texto.

Diferentemente da maioria das linguagens de programação, a linguagem JavaScript não possui o conceito de entrada e saída. Ela é projetada para funcionar como uma linguagem de script em um ambiente de terceiros, e cabe ao ambiente fornecer mecanismos para a comunicação com o mundo exterior. O ambiente de terceiros (hospedeiro) mais comum é o navegador, mas interpretadores JavaScript também pode ser encontrados no Adobe Acrobat, Photoshop, imagens SVG, Widget engine do Yahoo!, bem como ambientes de servidor, como Node.js. No entanto, a lista aqui apresentada das áreas nas quais a JavaScript é utilizada é apenas o começo. Ela também inclui bancos de dados NoSQL, como o código-fonte aberto Apache CouchDB, computadores embarcados, ou ambientes de trabalho completos, como o GNOME (um dos GUIs mais populares para os sistemas operacionais GNU / Linux).

## Visão Geral

A JavaScript é uma linguagem dinâmica orientada a objetos; tem tipos e operadores, objetos e métodos. Sua sintaxe vem das linguagens Java e C, por isso tantas estruturas dessas linguagens se aplicam a JavaScript também. Uma das principais diferenças é que o JavaScript não tem classes; em vez disso, a funcionalidade de classe é realizada por protótipos de objetos. A outra diferença principal é que as funções são objetos, dando as funções a capacidade para armazenar código executável e serem passadas como parametro para qualquer outro objeto.

Vamos começar pelo bloco de construção de qualquer linguagem: os tipos. Programas JavaScript manipulam valores, e esses valores todos pertencem a um tipo. Tipos de JavaScript são :

- números
- strings
- booleanos
- funções
- objetos

... Ops, e o "indefinido" e o "nulo"- , que parecem um pouco estranhos. E arrays (vetores), que são um tipo especial de objeto. E as datas e expressões regulares, que são objetos que você ganha de graça. E para ser tecnicamente preciso, as funções são apenas um tipo especial de objeto. Assim, a lista de tipos se parece mais com isto:

- números (numbers)
- strings (strings)
- booleanos (booleans)
- objetos (objects)
  - funções (functions)
  - vetores (arrays)
  - datas (dates)
  - expressoes regulares (regexp)
- nulo (null)
- indefinido (undefined)

E existem também alguns tipos para erros. As coisas são muito mais fáceis se ficarmos com a primeira lista, no entanto.

## Números

Números em JavaScript são "valores de precisão dupla no formato IEEE 754", de acordo com a especificação. Isto tem algumas consequências interessantes. Não existe essa coisa de inteiro em JavaScript, então você deve ser um pouco cuidadoso com seus cálculos se você está acostumado a matemática em C ou Java. Cuidado com coisas como:

1 |  $0.1 + 0.2 == 0.30000000000000004$

Na prática, valores inteiros são tratados como inteiros de 32 bits (e são armazenados dessa forma em algumas implementações do navegador), que podem ser importantes para as operações bit a bit. Para mais detalhes, consulte [The Complete JavaScript Number Reference](#).

Os [operadores numéricos](#) padrões são suportados, incluindo adição, subtração, módulo (ou resto) aritmético e assim por diante. Há também um objeto embutido que eu esqueci de mencionar mais cedo chamado [Math](#) para manipular funções e constantes matemáticas mais avançadas:

```
1 | Math.sin(3.5);  
2 | var d = Math.PI * r * r;
```

Você pode converter uma string em um inteiro usando a função embutida [parseInt\(\)](#). Ela tem um segundo parâmetro opcional para a base da conversão, parâmetro esse que você deveria sempre prover:

```
1 | > parseInt("123", 10)  
2 | 123
```

Se você quiser converter um número binário em um inteiro, basta mudar a base:

```
1 | > parseInt("11", 2)  
2 | 3
```

Similarmente, você pode fazer a conversão de números de ponto flutuante usando a função embutida [parseFloat\(\)](#) que usa a base 10 sempre, ao contrário de seu primo [parseInt\(\)](#).

Você também pode usar o operador unário `+` para converter valores em números:

```
1 | > + "42"  
2 | 42
```

Um valor especial chamado [NaN](#) (sigla de "Not a Number ou Não é Número") é retornado se a string não é um valor numérico:

```
1 | > parseInt("hello", 10)  
2 | NaN
```

NaN é tóxico: Se você provê-lo como uma entrada para qualquer operação matemática o resultado também será NaN:

```
1 | > NaN + 5
```

2 | NaN

Você pode testar se é NaN usando a função embutida `isNaN()`:

```
1 | > isNaN(NaN)
2 | true
```

JavaScript também tem os valores especiais `Infinity` e `-Infinity`:

```
1 | > 1 / 0
2 | Infinity
3 | > -1 / 0
4 | -Infinity
```

Você pode testar se o valor é `Infinity`, `-Infinity` e `NaN` usando a função embutida `isFinite()`:

```
1 | > isFinite(1/0)
2 | false
3 | > isFinite(-Infinity)
4 | false
5 | > isFinite(NaN)
6 | false
```

📌 Nota: As funções `parseInt()` e `parseFloat()` fazem a conversão da string até alcançarem um caractere que não é válido para o formato numérico especificado, então elas retornam o número convertido até aquele ponto. Contudo, o operador `+` simplesmente converte a string em `NaN` se tiver algum caractere inválido nela. Apenas tente por si mesmo converter a string `"10.2abc"` usando cada um desses métodos no console e entenderá melhor essas diferenças.

## Strings

Strings em JavaScript são sequências de caracteres. Para ser mais exato, elas são sequências de [Unicode characters](#), em que cada um deles é representado por um número de 16-bits. Isso deveria ser uma notícia bem-vinda para aqueles que tiveram que lidar com internacionalização.

Se você quiser representar um único caractere, você só tem que usar uma string de tamanho 1.

Para obter o tamanho de uma string, acesse sua propriedade `length`:

```
1 | > "hello".length
2 | 5
```

Essa é nossa primeira pincelada com objetos JavaScript! Eu mencionei que strings também são objetos? De

modo que elas têm métodos:

```
1 > "hello".charAt(0)
2 h
3 > "hello, world".replace("hello", "goodbye")
4 goodbye, world
5 > "hello".toUpperCase()
6 HELLO
```

## Outros tipos

No JavaScript há distinção entre `null`, que é um objeto do tipo `'object'` para indicar deliberadamente uma ausência de valor, do `undefined`, que é um objeto do tipo `'undefined'` para indicar um valor não inicializado — isto é, que um valor não foi atribuído ainda. Vamos falar sobre variáveis depois, mas em JavaScript é possível declarar uma variável sem atribuir um valor para a mesma. Se você fizer isso, a variável será do tipo `undefined`.

JavaScript tem um tipo `boolean`, ao qual são possíveis os valores `true` e `false` (ambos são palavras-chave). Qualquer valor pode ser convertido em um `boolean`, de acordo com as seguintes regras:

1. `false`, `0`, a string vazia(`""`), `NaN`, `null`, e `undefined` todos tornam-se `false`
2. todos os outros valores tornam-se `true`

Você pode fazer essa conversão explicitamente usando a função `Boolean()`:

```
1 > Boolean("")
2 false
3 > Boolean(234)
4 true
```

Contudo, essa é uma necessidade rara, uma vez que JavaScript silenciosamente fará essa conversão quando for esperado um `boolean`, como em uma instrução `if`. Por isso, algumas vezes falamos simplesmente em "valores `true`" e "valores `false`" nos referindo a valores que se tornaram `true` e `false`, respectivamente, quando convertidos em `boolean`. Alternativamente, esses valores podem ser chamados de "truthy" (verdade/verdadeiro) e "falsy" (falso/falsidade), respectivamente.

Operações booleanas como `&&` (*and* lógico), `||` (*or* lógico), e `!` (*not* lógico) são suportadas.

## Variáveis

Novas variáveis em JavaScript são declaradas usando a palavra-chave `var`:

```
1 var a;
2 var name = "simon";
```

Se você declarar uma variável sem atribuir qualquer valor a ela, seu tipo será `undefined`.

Uma diferença importante de outras linguagens como Java é que em JavaScript, blocos não tem escopo; somente funções tem escopo. De modo que se uma variável é definida usando `var` numa instrução composta (por exemplo dentro de uma estrutura de controle `if`), ela será visível por toda a função.

## Operadores

Operadores numéricos de JavaScript são `+`, `-`, `*`, `/` e `%` - que é o operador resto. Valores são atribuídos usando `=`, e temos também as instruções de atribuição compostas, como `+=` e `-=`. Essas são o mesmo que `x = x operador y`.

```
1 | x += 5
2 | x = x + 5
```

Você pode usar `++` e `--` para incrementar ou decrementar respectivamente. Eles podem ser usados como operadores tanto antes como depois.

O [operador `+`](#) também faz concatenação de string:

```
1 | > "hello" + " world"
2 | hello world
```

Se você adicionar uma string a uma número (ou outro valor) tudo será convertido em uma string primeiro. Isso talvez seja uma pegadinha para você:

```
1 | > "3" + 4 + 5
2 | 345
3 | > 3 + 4 + "5"
4 | 75
```

Adicionar uma string em branco a algo é uma maneira melhor de fazer a conversão.

[Comparações](#) em JavaScript podem ser feitas usando `<`, `>`, `<=` e `>=`. Isso funciona tanto para strings como para números. A igualdade é um pouco menos simples. O operador igual-duplo faz a coersão de tipo se você colocar tipos diferentes, algumas vezes com resultados interessantes:

```
1 | > "dog" == "dog"
2 | true
3 | > 1 == true
4 | true
```

Para evitar a coerção de tipo, use o operador igual-triplo:

```
1 > 1 === true
2 false
3 > true === true
4 true
```

Temos também os operadores `!=` e `!==`.

JavaScript também tem [operações de bit-a-bit](#). Se quiser usá-las, elas estarão lá.

## Estruturas de Controle

JavaScript tem um conjunto de estruturas de controle similar à outras linguagens na família do C. Instruções condicionais são suportadas pelo `if` e `else`; você pode encadeá-los se quiser:

```
1 var name = "kittens";
2 if (name == "puppies") {
3     name += "!";
4 } else if (name == "kittens") {
5     name += "!!";
6 } else {
7     name = "!" + name;
8 }
9 name == "kittens!!"
```

JavaScript tem as estruturas de repetição com os laços `while` e `do-while`. O primeiro é bom para repetições básicas; o segundo é para os casos em que você queira que o corpo da repetição seja executado pelo menos uma vez:

```
1 while (true) {
2     // an infinite loop!
3 }
4
5 var input;
6 do {
7     input = get_input();
8 } while (inputIsValid(input))
```

O laço `for` do JavaScript é o mesmo que no C e Java: ele lhe permite prover as informações para o seu laço em uma única linha.

```
1 for (var i = 0; i < 5; i++) {
2     // Will execute 5 times
```

```
3 | }
```

Os operadores `&&` e `||` usam a lógica de curto-circuito, o que quer dizer que, o segundo operando ser executado dependerá do primeiro operando. Isso é útil para checagem de objetos null antes de acessar seus atributos:

```
1 | var name = o && o.getName();
```

Ou para configuração de valores-padrões:

```
1 | var name = otherName || "default";
```

JavaScript tem um operador ternário para expressões condicionais:

```
1 | var allowed = (age > 18) ? "yes" : "no";
```

A instrução `switch` pode ser usada para múltiplas ramificações baseadas em um número ou uma string:

```
1 | switch(action) {  
2 |     case 'draw':  
3 |         drawit();  
4 |         break;  
5 |     case 'eat':  
6 |         eatit();  
7 |         break;  
8 |     default:  
9 |         donothing();  
10 | }
```

Se você não adicionar a instrução `break`, a execução irá "cair" no próximo nível. Isso é algo que raramente vai querer fazer — de fato vale mais a pena colocar um comentário especificando essa "queda" para o próximo nível, pois isso o ajudará na hora de fazer a depuração de seu código:

```
1 | switch(a) {  
2 |     case 1: // queda  
3 |     case 2:  
4 |         eatit();  
5 |         break;  
6 |     default:  
7 |         donothing();  
8 | }
```



A cláusula default é opcional. Se quiser, pode colocar expressões tanto no switch como nos cases; Comparações acontecem entre os dois usando o operador ===:

```
1 | switch(1 + 3) {  
2 |     case 2 + 2:  
3 |         yay();  
4 |         break;  
5 |     default:  
6 |         neverhappens();  
7 | }
```

## Objetos

Objetos JavaScript são simplesmente coleções de pares nome-valor. Como tal, eles são similares à:

- Dicionários em Python
- Hashes em Perl e Ruby
- Hash tables em C e C++
- HashMaps em Java
- Vetores associativos em PHP

Esse tipo de estrutura de dados é largamente utilizada, o que atesta sua versatilidade. Uma vez que tudo em JavaScript é um objeto (tipos básicos principais), qualquer programa JavaScript naturalmente envolve uma grande quantidade de buscas em tabelas hash, o que é algo bom, pois elas são bem rápidas!

A parte "name" é uma string JavaScript, enquanto o valor pode ser qualquer valor JavaScript — incluindo mais objetos. Isso permite que você construa estruturas de dados de qualquer complexidade.

Há basicamente duas formas de criar um objeto vazio:

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

e:

```
1 | var obj = {};
```

Elas são semanticamente equivalentes; a segunda forma é chamada de sintaxe de objeto literal e é mais conveniente. Essa sintaxe é também o coração do formato JSON e deveria ser sempre preferida.

Uma vez criada, as propriedades de um objeto podem novamente ser acessadas de uma das seguintes formas:

```
1 | obj.name = "Simon";  
2 | var name = obj.name;
```

E...

```
1 | obj["name"] = "Simon";  
2 | var name = obj["name"];
```

Estas também são semanticamente equivalentes. A segunda forma tem a vantagem de que o valor da chave é passado através de uma string, que pode ser calculada em tempo de execução, muito embora esse método previna o uso de alguns mecanismos tais como a otimização e a minificação. Outra vantagem é a possibilidade de se atribuir **palavras-reservadas** aos nomes das propriedades:

```
1 | obj.for = "Simon"; // Erro de sintaxe, pois 'for' é uma palavra reservada  
2 | obj["for"] = "Simon"; // Funciona bem
```

A sintaxe de objeto literal pode ser usada para inicializar completamente um objeto:

```
1 | var obj = {  
2 |     name: "Carrot",  
3 |     "for": "Max",  
4 |     details: {  
5 |         color: "orange",  
6 |         size: 12  
7 |     }  
8 | }
```

O acesso aos atributos podem ser encadeados:

```
1 | > obj.details.color  
2 | orange  
3 | > obj["details"]["size"]  
4 | 12
```

## Vetores

Vetores em JavaScript são, de fato, um tipo especial de objeto. Eles funcionam de forma muito similar à objetos regulares (propriedades numéricas podem naturalmente ser acessadas somente usando a sintaxe [], colchetes ), porém eles tem uma propriedade mágica chamada 'length'. Ela sempre é o maior índice de um vetor mais 1.

A forma tradicional de se criar um vetor em JavaScript é a seguinte:

```
1 > var a = new Array();
2 > a[0] = "dog";
3 > a[1] = "cat";
4 > a[2] = "hen";
5 > a.length
6 3
```

Existe uma notação mais conveniente usando um vetor literal:

```
1 > var a = ["dog", "cat", "hen"];
2 > a.length
3 3
```

Deixar uma vírgula à direita no final de um vetor literal gerará inconsistência entre os navegadores, portanto não faça isso.

Note que `array.length` não é necessariamente o número de itens em um vetor. Considere o seguinte:

```
1 > var a = ["dog", "cat", "hen"];
2 > a[100] = "fox";
3 > a.length
4 101
```

Lembre-se — o tamanho de um vetor é o maior índice mais 1.

Se você fizer referência a um índice de vetor inexistente, obterá um `undefined`:

```
1 > typeof a[90]
2 undefined
```

Você pode iterar sobre um vetor da seguinte forma:

```
1 for (var i = 0; i < a.length; i++) {
2     // Faça algo com a[i]
3 }
```

Isso é um pouco ineficaz visto que você está procurando a propriedade `length` uma vez a cada iteração. Uma melhoria poderia ser:

```
1 for (var i = 0, len = a.length; i < len; i++) {
2     // Faça algo com a[i]
```

```
3 | }
```

Uma forma mais elegante ainda poderia ser:

```
1 | for (var i = 0, item; item = a[i++];) {  
2 |     // Faça algo com item  
3 | }
```

Aqui nós estamos declarando duas variáveis. A atribuição na parte do meio do laço `for` é também testada — se for verdadeira, o laço continuaria. Uma vez que o `i` é incrementado toda vez, os itens do array serão atribuídos a variável `item` sequencialmente. A iteração é finalizada quando `item` "falsy" é encontrado (tal como o `undefined`, `false` ou zero).

Note que esse truque só deveria ser usado em vetores que você sabe não conter valores "falsy" (vetores de objeto ou nós [DOM](#) por exemplo). Se você iterar sobre dados numéricos que possam ter o 0 ou sobre dados string que possam ter uma string vazia, você deveria usar a segunda forma como alternativa.

Uma outra forma de iterar é usar o laço `for...in`. Note que se alguém adicionou novas propriedades ao `Array.prototype`, elas também podem ser iteradas usando este laço:

```
1 | for (var i in a) {  
2 |     // Do something with a[i]  
3 | }
```

Se quiser adicionar um item a um vetor, simplesmente faça desse jeito:

```
1 | a[a.length] = item;           // é o mesmo que a.push(item);
```

Vetores vem com vários métodos:

Nome do método	Descrição
<code>a.toString()</code>	Retorna uma string com o <code>toString()</code> de cada elemento separado por vírgulas.
<code>a.toLocaleString()</code>	Retorna uma string com o <code>toLocaleString()</code> de cada elemento separado por vírgulas.
<code>a.concat(item[, itemN])</code>	Retorna um novo vetor com os itens adicionados nele.
<code>a.join(sep)</code>	Converte um vetor em uma string com os valores do vetor separados pelo valor do parâmetro <code>sep</code>
<code>a.pop()</code>	Remove e retorna o último item.

<code>a.push(item[, itemN])</code>	Push adiciona um ou mais itens ao final.
<code>a.reverse()</code>	Reverte o vetor
<code>a.shift()</code>	Remove e retorna o primeiro item
<code>a.slice(start, end)</code>	Retorna um sub-vetor.
<code>a.sort([cmpfn])</code>	Prover uma função opcional para fazer a comparação.
<code>a.splice(start, delcount[, itemN])</code>	Permite que você modifique um vetor por apagar uma seção e substituí-lo com mais itens.
<code>a.unshift([item])</code>	Acrescenta itens ao começo do vetor.

## Funções

Junto com objetos, funções são os componentes principais para o entendimento do JavaScript. A função mais básica não poderia ser mais simples:

```
1 function add(x, y) {  
2     var total = x + y;  
3     return total;  
4 }
```

Isso demonstra tudo o que há para se saber sobre funções básicas. Uma função JavaScript pode ter 0 ou mais parâmetros declarados. O corpo da função pode conter tantas instruções quantas quiser e pode declarar suas próprias variáveis que são de escopo local àquela função. A instrução `return` pode ser usada para retornar um valor em qualquer parte da função, finalizando a função. Se nenhuma instrução de retorno for usada (ou um retorno vazio sem valor), o JavaScript retorna `undefined`.

Os parâmetros nomeados se parecem mais com orientações do que com outra coisa. Você pode chamar a função sem passar o parâmetro esperado, nesse caso eles receberão o valor `undefined`.

```
1 > add()  
2 NaN // Você não pode executar adição em undefined
```

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

```
1 > add(2, 3, 4)  
2 5 // adicionado os dois primeiros; 4 foi ignorado
```

Pode parecer um pouco bobo, mas no corpo da função você tem acesso a uma variável adicional chamada `arguments`, que é um objeto parecido com um vetor que contém todos os valores passados para a função. Vamos rescrever a função `add` para tomarmos tantos valores quanto quisermos:

```
1 function add() {
2     var sum = 0;
3     for (var i = 0, j = arguments.length; i < j; i++) {
4         sum += arguments[i];
5     }
6     return sum;
7 }
8
9 > add(2, 3, 4, 5)
10 14
```

Isso realmente não é muito mais útil do que escrever `2 + 3 + 4 + 5`. Vamos criar uma função para calcular média:

```
1 function avg() {
2     var sum = 0;
3     for (var i = 0, j = arguments.length; i < j; i++) {
4         sum += arguments[i];
5     }
6     return sum / arguments.length;
7 }
8 > avg(2, 3, 4, 5)
9 3.5
```

Isso é muito útil, mas introduz um novo problema. A função `avg()` precisa de uma lista de argumentos separados por vírgula — mas e se o que quiser for procurar a média de um vetor? Você poderia apenas reescrever a função da seguinte forma:

```
1 function avgArray(arr) {
2     var sum = 0;
3     for (var i = 0, j = arr.length; i < j; i++) {
4         sum += arr[i];
5     }
6     return sum / arr.length;
7 }
8 > avgArray([2, 3, 4, 5])
9 3.5
```

Porém, seria legal se pudéssemos reusar a função que já tínhamos criado. Felizmente, JavaScript lhe permite chamar a função, e chamá-la com um conjunto arbitrário de argumentos, usando o método `apply()` presente em qualquer objeto função.

```
1 > avg.apply(null, [2, 3, 4, 5])
2 3.5
```

O segundo argumento do `apply()` é o vetor para usar como argumento; o primeiro será discutido mais tarde. Isso enfatiza o fato que funções também são objetos.

JavaScript lhe permite criar funções anônimas.

```
1  var avg = function() {
2      var sum = 0;
3      for (var i = 0, j = arguments.length; i < j; i++) {
4          sum += arguments[i];
5      }
6      return sum / arguments.length;
7  }
```

Isso é semanticamente equivalente a `function avg()`. É extremamente poderoso como ele lhe permite colocar a definição completa de uma função em qualquer lugar, que você normalmente poria uma expressão. Isso lhe permite toda sorte de truques engenhosos. Aqui está uma maneira de "esconder" algumas variáveis locais — como escopo de bloco em C:

```
1  > var a = 1;
2  > var b = 2;
3  > (function() {
4      var b = 3;
5      a += b;
6  })();
7  > a
8  4
9  > b
10 2
```

JavaScript lhe permite chamar funções recursivamente. Isso é particularmente útil quando estamos lidando com estruturas de árvore, como quando estávamos navegando no [DOM](#).

```
1  function countChars(elm) {
2      if (elm.nodeType == 3) { // TEXT_NODE
3          return elm.nodeValue.length;
4      }
5      var count = 0;
6      for (var i = 0, child; child = elm.childNodes[i]; i++) {
7          count += countChars(child);
8      }
9      return count;
10 }
```


Isso destaca um problema potencial com funções anônimas: Como chamá-las recursivamente se elas não tem

um nome? JavaScript lhe permite nomear expressões de função para isso. Você pode usar EFLs nomeadas (Expressões Funcionais Imediatamente Invocadas), conforme abaixo:

```
1  var charsInBody = (function counter(elm) {
2      if (elm.nodeType == 3) { // TEXT_NODE
3          return elm.nodeValue.length;
4      }
5      var count = 0;
6      for (var i = 0, child; child = elm.childNodes[i]; i++) {
7          count += counter(child);
8      }
9      return count;
10 })(document.body);
```

O nome provido para a função anônima conforme acima só é (ou no mínimo só deveria ser) visível ao escopo da própria função. Isso tanto permite que mais otimizações sejam feitas pela engine como deixa o código mais legível.

## Objetos Personalizados

 **Nota:** Para uma discussão mais detalhada de programação orientada a objetos em JavaScript, veja [Introdução a JavaScript Orientado a Objeto](#).

Na clássica Programação Orientada a Objetos, objetos são coleções de dados e métodos que operam sobre esses dados. JavaScript é uma linguagem baseada em protótipos que não contém a estrutura de classe, como tem em C++ e Java. (Algumas vezes isso é algo confuso para o programador acostumado a linguagens com estrutura de classe) Em vez disso, JavaScript usa funções como classes. Vamos considerar um objeto pessoa com os campos primeiro e último nome. Há duas formas em que o nome talvez possa ser exibido: como "primeiro nome segundo nome" ou como "último nome, primeiro nome". Usando as funções e objetos que discutimos anteriormente, aqui está uma forma de fazer isso:

```
1  function makePerson(first, last) {
2      return {
3          first: first,
4          last: last
5      }
6  }
7  function personFullName(person) {
8      return person.first + ' ' + person.last;
9  }
10 function personFullNameReversed(person) {
11     return person.last + ', ' + person.first
12 }
13 > s = makePerson("Simon", "Willison");
14 > personFullName(s)
15 Simon Willison
```



```
16 | > personFullNameReversed(s)
17 | Willison, Simon
```

Isso funciona, mas é muito feio. Você termina com dúzias de funções em seu escopo global. O que nós realmente precisamos é uma forma de anexar uma função a um objeto. Visto que funções são objetos, isso é fácil:

```
1 | function makePerson(first, last) {
2 |     return {
3 |         first: first,
4 |         last: last,
5 |         fullName: function() {
6 |             return this.first + ' ' + this.last;
7 |         },
8 |         fullNameReversed: function() {
9 |             return this.last + ', ' + this.first;
10 |        }
11 |    }
12 | }
13 | > s = makePerson("Simon", "Willison")
14 | > s.fullName()
15 | Simon Willison
16 | > s.fullNameReversed()
17 | Willison, Simon
```

Há algo aqui que não havíamos visto anteriormente: a palavra-chave `'this'`. Usada dentro de uma função, `'this'` refere-se ao objeto corrente. O que aquilo de fato significa é especificado pelo modo em que você chamou aquela função. Se você chamou-a usando [notação ponto](#) ou [notação colchete](#) em um objeto, aquele objeto torna-se `'this'`. Se a notação ponto não foi usada pela chamada, `'this'` refere-se ao objeto global. Isso é uma frequente causa de erros. Por exemplo:

```
1 | > s = makePerson("Simon", "Willison")
2 | > var fullName = s.fullName;
3 | > fullName()
4 | undefined undefined
```

Quando chamamos `fullName()`, `'this'` está ligado ao objeto global. Visto que não há variáveis globais chamadas `first` ou `last` obtemos `undefined` para cada um.

Podemos tirar vantagem da palavra chave `'this'` para melhorar nossa função `makePerson`:

```
1 | function Person(first, last) {
2 |     this.first = first;
3 |     this.last = last;
4 |     this.fullName = function() {
```

```
4     return this.first + ' ' + this.last;
5   }
6   this.fullNameReversed = function() {
7     return this.last + ', ' + this.first;
8   }
9 }
10 var s = new Person("Simon", "Willison");
11
```

Nós introduzimos uma outra palavra-chave: `'new'`. `new` é fortemente relacionada a `'this'`. O que ele faz é criar um novo objeto vazio, e então chamar a função especificada com `'this'` para atribuir aquele novo objeto. Funções que são desenhadas para ser chamadas pelo `'new'` são chamadas de funções construtoras. Uma prática comum é capitular essas funções como um lembrete de chamá-las com o `new`.

Nossos objetos pessoa estão ficando melhor, mais ainda existem algumas arestas feias. Toda vez que criamos um objeto pessoa, nos criamos duas marcas de nova função dentro dele — não seria melhor se este código fosse compartilhado?

```
1 function personFullName() {
2   return this.first + ' ' + this.last;
3 }
4 function personFullNameReversed() {
5   return this.last + ', ' + this.first;
6 }
7 function Person(first, last) {
8   this.first = first;
9   this.last = last;
10  this.fullName = personFullName;
11  this.fullNameReversed = personFullNameReversed;
12 }
```

Assim está melhor: nós estamos criando as funções de método apenas uma vez, e atribuímos referências para elas dentro do construtor. Podemos fazer algo melhor do que isso? A resposta é sim:

```
1 function Person(first, last) {
2   this.first = first;
3   this.last = last;
4 }
5 Person.prototype.fullName = function() {
6   return this.first + ' ' + this.last;
7 }
8 Person.prototype.fullNameReversed = function() {
9   return this.last + ', ' + this.first;
10 }
```

`Person.prototype` é um objeto compartilhado por todas as instâncias de `Person`. Este forma parte da cadeia

de buscas (que tem um nome especial, cadeia de protótipos ou "prototype chain"): toda a vez que você tentar acessar uma propriedade de `Person` que não está configurada, Javascript irá verificar em `Person.prototype` para ver se esta propriedade existe por lá. Como resultado, qualquer coisa atribuída a `Person.prototype` torna-se disponível para todas as instâncias deste construtor, através do objeto `this`.

Esta é uma ferramenta incrivelmente poderosa. JavaScript permite a você modificar algo prototipado em qualquer momento no seu programa, isto significa que você pode adicionar métodos extras para objetos pré-existentes, em tempo de execução:

```
1 > s = new Person("Simon", "Willison");
2 > s.firstNameCaps();
3 TypeError on line 1: s.firstNameCaps is not a function
4 > Person.prototype.firstNameCaps = function() {
5     return this.first.toUpperCase()
6 }
7 > s.firstNameCaps()
8 SIMON
```

Curiosamente, você pode também adicionar coisas para o protótipo de objetos built-in de Javascript. Vamos adicionar um método para `String` que retorna a string invertida:

```
1 > var s = "Simon";
2 > s.reversed()
3 TypeError on line 1: s.reversed is not a function
4 > String.prototype.reversed = function() {
5     var r = "";
6     for (var i = this.length - 1; i >= 0; i--) {
7         r += this[i];
8     }
9     return r;
10 }
11 > s.reversed()
12 nomiS
```

Nosso novo método funciona inclusive em string literais!

```
1 > "This can now be reversed".reversed()
2 desrever eb won nac sihT
```

Como eu mencionei antes, o protótipo forma parte de uma cadeia. A raiz dessa cadeia é `Object.prototype`, dos quais inclui o método `toString()` — este é o método que é chamado quando você tenta representar um objeto como uma string. Isto é útil para depurar os nossos objetos `Person`:

```
1 > var s = new Person("Simon", "Willison");
```

```
2 | > s
3 | [object Object]
4 | > Person.prototype.toString = function() {
5 |     return '<Person: ' + this.fullName() + '>';
6 | }
7 | > s
8 | <Person: Simon Willison>
```

Lembra como `avg.apply()` tinha um primeiro argumento `null`? Nós podemos revisitar isto, agora. O primeiro argumento para `apply()` é o objeto que deve ser tratado como `'this'`. Por exemplo, aqui está uma implementação trivial de `'new'`:

```
1 | function trivialNew(constructor) {
2 |     var o = {}; // Create an object
3 |     constructor.apply(o, arguments);
4 |     return o;
5 | }
```

Isto não é exatamente uma réplica de `new` porque não configura a cadeia de protótipos. `apply()` is difícil de ilustrar — não é algo que você usa com frequência, mas é útil conhecer a respeito.

`apply()` tem uma função irmã de nome `call`, que novamente permite você configurar o `'this'` mas toma uma lista expandida de argumentos, ao invés de um array.

```
1 | function lastNameCaps() {
2 |     return this.last.toUpperCase();
3 | }
4 | var s = new Person("Simon", "Willison");
5 | lastNameCaps.call(s);
6 | // Is the same as:
7 | s.lastNameCaps = lastNameCaps;
8 | s.lastNameCaps();
```

## Funções Internas

Em JavaScript é permitido declarar uma função dentro de outras funções. Nós já vimos isso antes, com uma versão preliminar da função `makePerson()`. Um detalhe importante, sobre funções aninhadas em JavaScript é que elas podem acessar as variáveis do escopo das funções parente:

```
1 | function betterExampleNeeded() {
2 |     var a = 1;
3 |     function oneMoreThanA() {
4 |         return a + 1;
5 |     }
6 |     return oneMoreThanA();
7 | }
```

```
6 | }  
7 |
```

Isto permite um grande compromisso de utilidade, em escrever um código de melhor manutenibilidade. Se uma função depende de uma ou mais funções que não são úteis para outras partes do seu código, você pode aninhar estas funções utilitárias dentro da função que será chamada. Isto mantém o número de funções que estão no escopo global, baixo, o que é sempre uma boa coisa.

Isto é também um ótimo contador de atração de variáveis globais. Quando se escreve um código complexo, é sempre tentador usar as variáveis globais para compartilhar valores entre múltiplas funções — do qual leva a um código que é difícil de manter. Funções aninhadas podem compartilhar variáveis em seus parentes, então você pode usar este mecanismo para acoplar e juntar funções, quando isto fizer sentido, sem poluir o seu "namespace" global — 'globais locais' se preferir. Esta técnica deve ser usada com cautela, mas é uma habilidade a se ter.

## Clausuras (Closures)

Isto nos leva a uma das abstrações mais poderosas que JavaScript tem a oferecer — mas também a mais potencialmente confusa. O que isto faz?

```
1 | function makeAdder(a) {  
2 |     return function(b) {  
3 |         return a + b;  
4 |     }  
5 | }  
6 | x = makeAdder(5);  
7 | y = makeAdder(20);  
8 | x(6)  
9 | ?  
10 | y(7)  
11 | ?
```

O nome da função `makeAdder` já diz tudo: ela cria novas funções 'adder', na qual, quando chamada com um argumento, adiciona o argumento com a que foi criada.

O que está acontecendo aqui é muito parecido com o que estava acontecendo com as funções internas, vistas anteriormente: uma função definida dentro de uma outra função, tem acesso as variáveis da função de fora. A única diferença aqui é que a função de fora retornou, e como consequência do senso comum, deve dizer que todas as variáveis locais não existem mais. Mas elas *ainda* existem — caso contrário a função adicionadora não seria capaz de funcionar. Mais ainda, há duas "cópias" diferentes de variáveis locais para `makeAdder` — uma na qual o `a` é 5 e a outra na qual `a` é 20. Então, o resultado dessas chamadas de funções é o seguinte:

```
1 | x(6) // returns 11  
2 | y(7) // returns 27
```

Aqui está o que acontece na verdade. Sempre que JavaScript executa uma função, um objeto de 'escopo' é criado para guardar as variáveis locais criadas dentro desta função. Ela é inicializada com quaisquer variáveis passadas como parâmetros da função. isto é similar ao objeto global, em que todas as variáveis globais e funções, vivem, mas com algumas diferenças importantes: primeiramente, um novo objeto de escopo é criado toda a vez que uma função começa a executar, e depois, diferente do objeto global (que nos navegadores é acessado como `window`) estes objetos não podem ser diretamente acessados através do seu código JavaScript. Não há nenhum mecanismo para iterar sobre as propriedades do escopo corrente do objeto, por exemplo.

Então, quando `makeAdder` é chamado, um objeto de escopo é criado com uma única propriedade: `a`, no qual é o argumento passado para a função `makeAdder`. `makeAdder` então retorna uma nova função criada. Normalmente o coletor de lixo de JavaScript poderia limpar o objeto de escopo criado para `makeAdder` neste ponto, mas a função de retorno mantém uma referência ao objeto de escopo. Como resultado, o objeto de escopo não será coletado como lixo até que não haja mais referências para função objeto que `makeAdder` retornou.

Objetos de escopo formam uma cadeia chamada de cadeia de escopos, similar a cadeia de protótipos usadas no sistema de objetos de JavaScript.

Uma clausura é a combinação de uma função e o objeto de escopo na qual é criado.

Clausuras permitem você guardar estado — de tal forma, elas podem ser frequentemente utilizadas no lugar de objetos.

## Vazamentos de Memória

Infelizmente, um efeito colateral de clausuras é que ela torna trivialmente fácil criar vazamentos de memória no Internet Explorer. JavaScript é uma linguagem com coleta de lixo — os objetos são alocados em memória no momento de sua criação, e esta memória é recuperada pelo navegador, quando nenhuma referência ao objeto restar. Objetos providos pelo ambiente de hospedagem são manipulados por este ambiente.

Os navegadores (hospedeiros) necessitam gerenciar um grande número de objetos, que representam a página HTML sendo criada — os objetos do [DOM](#). É responsabilidade do navegador, gerenciar a alocação e a recuperação desses objetos.

O Internet Explorer utiliza o seu esquema próprio de coleta de lixo para isto, separado do mecanismo utilizado no JavaScript. É a interação entre esses dois esquemas é que pode causar vazamentos de memória.

Um vazamento de memória no IE ocorre toda a vez que uma referência circular é formada entre um objeto do JavaScript e um objeto nativo. Considere o que segue:

```
1 function leakMemory() {  
2     var el = document.getElementById('el');  
3     var o = { 'el': el };  
4     el.o = o;  
5 }
```

A referência circular formada acima, cria um vazamento de memória; O IE não irá liberar a memória utilizada por `e1` e o até que o navegador seja completamente reiniciado.

É provável que o caso acima passe despercebido; o vazamento de memória somente torna-se uma real preocupação em aplicações que estão rodando a muito tempo ou em aplicações que vazam uma grande quantidade de memória, devido à estruturas de dados grandes ou padrões de vazamento em laços.

Os vazamentos raramente são óbvios — frequentemente, a estrutura de dados com vazamento pode ter muitas camadas de referências, escondendo a referência circular.

Clausuras torna fácil criar um vazamento de memória sem a intenção. Considere isto:

```
1 function addHandler() {  
2     var e1 = document.getElementById('e1');  
3     e1.onclick = function() {  
4         this.style.backgroundColor = 'red';  
5     }  
6 }
```

O código acima, configura o elemento para ficar vermelho quando clicado. Ele também cria um vazamento de memória. Por que? Porque a referência para `e1` é inadvertidamente capturada na clausura criada para a função interna anônima. Isto cria uma referência circular entre um objeto JavaScript (uma função) e um objeto nativo (`e1`).

```
1 | needsTechnicalReview();
```

Existe algumas formas de contornar este problema. O mais simples é não usar a variável `e1`:

```
1 function addHandler(){  
2     document.getElementById('e1').onclick = function(){  
3         this.style.backgroundColor = 'red';  
4     }  
5 }
```

Surpreendentemente, um truque para quebrar a referência circular introduzida pela clausura é adicionar mais uma clausura:

```
1 function addHandler() {  
2     var clickHandler = function() {  
3         this.style.backgroundColor = 'red';  
4     };  
5     (function() {  
6         var e1 = document.getElementById('e1');
```

```
7 |         el.onclick = clickHandler;  
8 |     })();  
9 | }
```

A função interna é executada imediatamente, e esconde o seu conteúdo da clausura criada com `clickHandler`.

Um outro bom truque para evitar clausuras é quebrar as referências circulares durante o evento `window.onunload`. Muitas bibliotecas de eventos farão isto para você. Note que fazendo isto você também desabilita o [bfcache no Firefox 1.5](#), então você não deve registrar um escutador de `unload` no Firefox, a menos que você tenha uma outra razão para fazer isso.

#### Original Document Information

- Author: [Simon Willison](#)
- Last Updated Date: March 7, 2006
- Copyright: © 2006 Simon Willison, contributed under the Creative Commons: Attribute-Sharealike 2.0 license.
- More information: For more information about this tutorial (and for links to the original talk's slides), see Simon's [Etech weblog post](#).