Eloquente

2ª edição

Marijn Haverbeke

JavaScript Eloquente

Uma moderna introdução ao JavaScript, programação e maravilhas digitais.

Eric Oliveira

This book is for sale at http://leanpub.com/eloquentejavascript

This version was published on 2014-02-08

Tweet This Book!

Please help Eric Oliveira by spreading the word about this book on Twitter!

The suggested hashtag for this book is #eloquente javascript.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#eloquente javascript

Esta tradução tem como único e exclusivo objetivo, facilitar a disseminação da informação, para que a mesma possa tornar-se conhecimento a ser aplicado por todos os desenvolvedores que a lerem.

Agradeço ao autor do livro original, Marijn Haverbeke, por sua incrível contribuição para comunidade, disponibilizando este livro de forma gratuita.

"Existe apenas um bem, o saber, e apenas um mal, a ignorância." Sócrates

Conteúdo

Notas da tradução

Esta tradução está sendo lançada previamente para auxílio daqueles que já se interessarem na leitura dos tópicos aqui propostos.

Ainda não foram feitas revisões, que assim serão após a conclusão das traduções.

A cada novo capítulo traduzido, será feita a atualização deste material.

As atualizações mais frequentes podem ser conferidas neste[1] repositório.

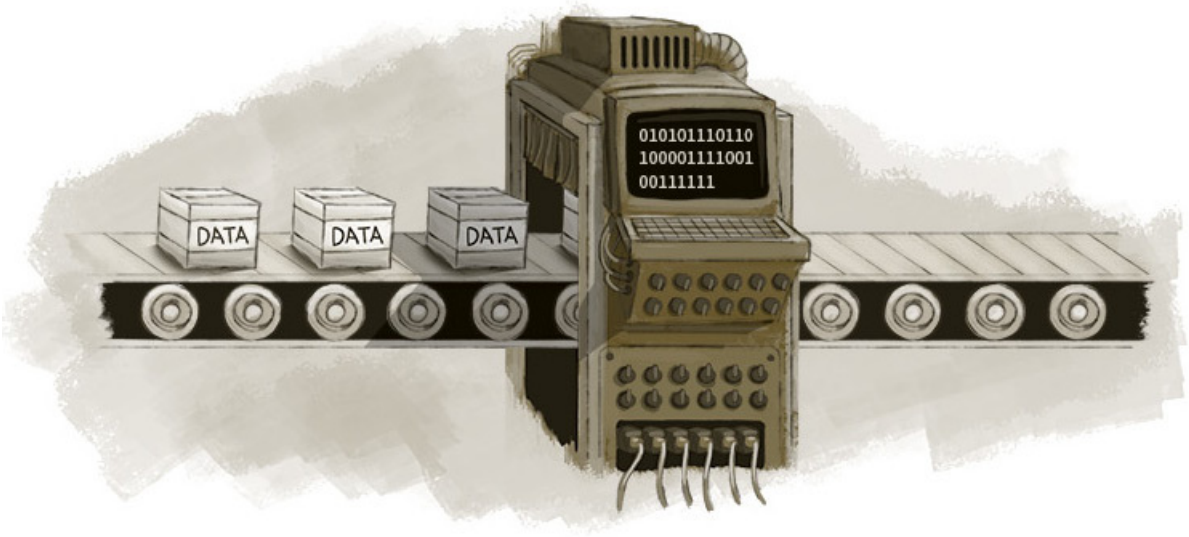
[1] https://github.com/eoop/eloquente-javascript

Introdução

Este livro é sobre como conversar com computadores. Computadores se tornaram uma das ferramentas fundamentais do nosso tempo. Ser capaz de controlá-los efetivamente é uma habilidade extremamente útil. Com a mentalidade certa, também pode ser muito divertido!

Há uma lacuna entre nós, organismos biológicos models com um talento para o raciocínio espacial e social, e o computador, um simples manipulador de dados insignificantes, sem nossos preconceitos e instintos. Felizmente, tivemos um grande progresso em preencher essa lacuna nos últimos sessenta anos.

A história da interação humano-computador tem se afastado da ideia fria e reducionista do mundo sobre o computador, apresentando camadas mais amigáveis em cima disso. As duas ideias mais importantes neste processo tem sido o uso de linguagens de computador, que mapeiam bem para o nosso cérebro por se parecer com as linguagens que usamos para conversas uns com os outros, e interfaces gráficas apontar e clicar (ou toque), que nós facilmente por causa de imitar o mundo tangível fora da máquina.



Mysterious Computer

Interfaces gráficas tendem a ser mais fáceis descobrir que linguagens - detectar um botão é mais rápido do que aprender uma gramática. Por essa razão, eles se tornaram a forma dominante de interagir com sistemas orientados ao consumidor. Compare com os telefones atuais, onde você pode realizar todo tipo de tarefa tocando e passando os elementos que aparecem na tela, com o *Commodore 64* de 1982, o aparelho que me introduziu a computação, onde tudo que você recebia era um cursor piscando, e você conseguia isto digitando comandos.

Obviamente, o telefone *touchscreen* é mais acessível, e é inteiramente apropriado que esses dispositivos utilizem uma interface gráfica. Mas as interfaces baseadas em linguagens têm outra vantagem - uma vez que você aprenda esta linguagem, ela tende a ser mais expressiva, tornando mais fácil compor a funcionalidade fornecida pelo sistema de novas maneiras, e até mesmo criando seus próprios blocos de construção.

Com o *Commodore 64*, quase todas as tarefas no sistema eram realizadas dando comandos embutidos na linguagem da máquina (um dialeto da linguagem de programação BASIC). Isto permitia aos usuários gradualmente progredir de simplesmente usar o computador (carregando programas) para de fato programá-los. Você estava *dentro* de um ambiente de programação desde o início, em vez de ter que procurar ativamente por um.

Isto foi perdido pela mudança para as interfaces gráficas de usuário. Mas as interfaces baseadas em linguagem, na forma de linguagens de programação, continuam aqui, em cada máquina, em grande parte escondida do usuário comum. Uma tal linguagem, JavaScript, está disponível em quase todos os dispositivos de consumidores como parte de um navegador web.

Este livro pretende torná-lo familiar com esta linguagem o suficiente para que você seja capaz de fazer com o computador o que você quiser.

Na Programação

> Eu não esclareço os que não estão prontos para aprender, nem desperto aqueles que não estão ansiosos para dar uma explicação a si próprios. Se eu apresentei um canto da praça, e eles não podem voltar para mim com os outros três, eu não deveria passar por estes pontos novamentes. **Confúcio**

Antes de explicar JavaScript, eu também quero introduzir os princípios básicos de programação. Programação, ao que parece, é difícil. As regras fundamentais são claras e simples. Mas programas, criados em cima destas regras básicas, tendem a tornar-se complexos o suficiente para introduzir suas próprias regras e complexidades. Você está construindo seu próprio labirinto, e pode simplesmente perder-se nele.

Para tirar algum proveito deste livro, mais do que apenas uma leitura passiva é necessário. Trate de ficar atento, faça um esforço para entender o exemplo de código, e somente continue quando você estiver razoavelmente seguro que você entendeu o material que veio antes.

> O programador de computadores é um criador de universos no qual ele é o único responsável. Universos de complexidade virtualmente ilimitada podem ser criados sob a forma de programas de computador. **Joseph Weizenbaum**, **Computer Power and Human Reason**

Um programa é muitas coisas. É um pedaço de texto digitado por um programador, que é a força direta que faz que o computador faça o que faz, são dados na memória do computador, mas ele

controla as ações realizadas nesta mesma memória. Analogias que tentam comparar programas com objetos que somos familiares tendem a ser insuficientes. Uma conexão superficial é a com uma máquina - muitas partes separadas tendem a ser envolvidas, e para fazer o conjunto todo precisamos considerar as maneiras que estas partes se interconectam e contribuam para a operação do todo.

Um computador é uma máquina feita para atuar como um hospedeiro para estas máquinas imateriais. Computadores por si próprios podem somente fazer coisas estúpidas e simples. A razão deles serem tão úteis que eles fazem coisas em uma velocidade incrível. Um programa pode ingenuamente combinar enormes números de simples ações ao invés de fazer coisas complicadas.

Para muitos de nós, escrever programas de computador é um fascinante jogo. Um programa é uma construção do pensamento. Não tem custos de construção, é leve e cresce facilmente ante nossas digitações.

Se não formos cuidadosos, seu tamanho e complexidade vão aumentar fora de controle, confundindo até a pessoa que o criou. Este é o principal problema da programação: manter os programas sobre controle. Quando um programa funciona, ele é lindo. A arte de programar é a habilidade de controlar a complexidade. Um grande programa é suave, é simples em sua complexidade.

Alguns programadores acreditam que esta complexidade é melhor gerenciada usando somente um pequeno conjunto de técnicas bem entendidas em seus programas. Eles compõe regras rígidas (*"boas práticas"*) prescrevendo a forma que programas devem ter, e os mais zelosos sobre isso vão considerar aqueles que saem desta pequena zona de segurança *maus programadores*.

Quanta hostilidade perante a riqueza da programação - tentar reduzir a algo simples e previsível, colocando um tabu em todos os lindos e misteriosos programas! A paisagem das técnicas de programação é enorme, fascinante em sua diversidade, e permanece largamente inexplorada. É sem dúvida perigoso ir neste caminho, atraindo o programador inexperiente em todo tipo de confusão, mas isso só significa que você deve proceder com cautela e manter o juízo. Conforme você aprende, sempre haverá novos desafios e novos territórios a ser explorados. Programadores que recusam de manter-se explorando vão estagnar, esquecer sua alegria, e ficar entediado com seu trabalho.

Porque linguagens importam?

No começo, no nascimento da programação, não havia linguagens de programação. Programas pareciam algo desta forma:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Este é um programa que soma os números do 1 ao 10 e imprimi o resultado (1 + 2 + ... 10 = 55). Isso pode rodar em uma muito simples, uma máquina hipotética. Para programar os primeiros computadores, era necessário configurar grandes arrays de chaves na posição certa, ou fazer furos em cartões e alimentá-los no computador. Você pode imaginar como isso era tedioso, e um procedimento propenso ao erro. Mesmo escrever simples programas requeriam muita habilidade e disciplina. Os complexos eram quase inconcebíveis.

Claro, inserindo manualmente estes padrões misteriosos de bits (1 e 0) fez que o programador tivesse uma profunda sensação de ser um poderoso feiticeiro. E isto tem que valer alguma coisa em termos de satisfação no trabalho.

Cada linha do programa contém uma simples instrução. Isto pode ser escrito assim:

```
1. Guarde o número 0 na posição da memória 0.
2. Guarde o número 1 na posição da memória 1.
3. Guarde o valor da posição da memória 1 na posição da memória 2.
4. Subtraia o número 11 do valor na posição da memória 2.
5. Se o valor na posição da memória 2 é o número 0, continue com a instrução 9.
6. Adicione o valor da posição da memória 1 para posição de memória 0.
7. Adicione o número 1 ao valor da posição de memória 1.
8. Continue com a instrução 3.
9. Retorne o valor da posição da memória 0.
```

Embora isto seja mais legível que a sopa de bits, ainda continua bastante desagradável. Pode ser de auxílio usar nomes ao invés de números para as instruções e locações de memória:

```
Configure "total" para 0
Configure "count" para 1
[loop]
Configure "compare" para "count"
Subtraia 11 de "compare"
Se "compare" é zero, continue até [fim]
Adicione "count" em "total"
Adicione 1 em "count"
Continue até [loop]
[fim]
Saída "total"
```

Neste ponto não é tão difícil ver como os programas trabalham. Você consegue? As primeiras duas linhas fornece duas locações de memória que iniciam os valores: `total` vai ser usado para construir o resultado da computação, e `count` mantém registrado o número que nós atualmente estamos olhando. As linhas usando `compare` são provavelmente as mais estranhas. O que o programa quer fazer é ver se já pode parar. Por causa da nossa máquina hipotética ser bastante primitiva, ela somente pode testar se um número é zero e fazer a decisão (salto) baseado nisto. Então, ela usa a locação de memória rotulada `compare` para computar o valor de `count` - 11 e fazer a decisão baseada neste valor. As próximas duas linhas adicionam o valor de `count` ao resultado e incrementam `count` por 1 cada vez que o programa decide que não é 11 ainda.

Aqui temos o mesmo programa em JavaScript:

```
var total = 0, count = 1;
while (count <= 10) {
        total += count;
        count += 1;
}
console.log(total);
```

Isso nos dá muitas melhorias. Mais importante, não é preciso mais especificar o caminho que nós queremos que o programa salte anteriormente ou adiante. Ele continua executando o bloco (envolvido nas chaves) até que a condição que foi dada seja: `count <= 10`, que significa "count é menor que ou igual a 10". Não temos mais que criar um valor temporário e compará-lo a zero. Isso é um detalhe desinteressante, e o poder das linguagens de programação é que elas tomam conta de detalhes desinteressantes para nós.

No final do programa, depois de `while` ser definido, a operação `console.log` é aplicada ao resultado na ordem que escrevemos isso como *output* (saída).

Finalmente, aqui temos o que o programa pode parecer se nós tivermos as operações convenientes `range` (alcance) e `sum` (soma) disponíveis, que respectivamente criam uma coleção de números com um alcance e computam a soma de uma coleção de números:

```
console.log(sum(range(1,10)));
// 55
```

A moral da história, então, é que o mesmo programa pode ser expresso de forma longa e curta, de forma legível ou não. A primeira versão do programa foi extremamente obscura, enquanto esta última é praticamente "Inglês": `log` (registre) a `sum` (soma) da `range` (extensão) dos números de 1 a 10. (Nós vamos ver nos próximos capítulos como criar coisas do tipo `sum` e `range`).

Uma boa linguagem de programação ajuda o programador permitindo-o conversr sobre ações que o computador vai realizar em *alto nível*. Isto ajuda a deixar detalhes desinteressantes implícitos, e fornece construções convenientes de blocos (como o `while` e `console.log`), permitindo a você definir seus próprios blocos (como `sum` e `range`), e tornando simples a construção destes blocos.

O que é JavaScript?

O JavaScript foi introduzido em 1995, como uma forma de adicionar programas a páginas da web no navegador Netscape. A linguagem foi adaptada pela maioria dos navegadores gráficos da web. Ele fez a atual geração de aplicações web possível - clientes de email baseado no navegador, mapas e redes sociais - e também é usado em sites mais tradicionais para fornecer várias formas de interatividade e inteligência.

É importante notar que JavaScript não tem quase nada a ver com a linguagem de programação Java. O nome similar foi inspirado por considerações de marketing, ao invés do bom senso. Quando o JavaScript foi introduzido, a linguagem Java estava sendo fortemente divulgada e ganhando popularidade. Alguém pensou ser uma boa ideia tentar trilhar junto com este sucesso. Agora estamos emperrados com este nome.

Depois da adoção fora do Netscape, um documento padrão foi escrito para descrever uma forma que a linguagem deve trabalhar, com um esforço para certificar-se que as várias partes do software que afirmavam suportar JavaScript estavam realmente falando sobre a mesma linguagem. Foi chamado de padrão ECMAScript, depois da organização ter feito a padronização. Na prática, os termos ECMAScript e JavaScript podem ser usados como sinônimos - são dois nomes para a mesma linguagem.

Tem alguns que vão dizer coisas *horríveis* sobre a linguagem JavaScript. Muitas dessas coisas são verdade. Quando eu fui obrigado a escrever algo em JavaScript, pela primeira vez, eu rapidamente vim a desprezá-lo - ele poderia interpretar qualquer coisa que eu digitei, mas interpretava de uma forma completamente diferente do que eu quis dizer. Isso teve muito a ver com o fato de que eu não tinha a menor ideia do que estava fazendo, claro, mas há uma questão real aqui: JavaScript é ridiculamente liberal no que ele permite. A ideia por trás deste padrão foi que isto tornaria a programação em JavaScript simples para iniciantes. Na realidade, na maior parte daz vezes isto torna a detecção de problemas em seus programas difícil, porque o sistema não vai apont-alo para você.

Esta flexibilidade também tem suas vantagens. Isso dá espaço para muitas técnicas que são impossíveis em linguagens mais rígidas, e, como iremos ver em capítulos posteriores, isto pode ser usado para superar algumas deficiências do JavaScript. Depois de aprender corretamente e trabalhar com o JavaScript por um tempo, eu aprendi a realmente *gostar* desta linguagem.

Tivemos várias *versões* do JavaScript. Versão 3 do ECMAScript foi a dominante, largamente suportado no tempo que o JavaScript ascendia para o domínio, aproximadamente entre 2000 e 2010. Durante este tempo, trabalho estava em andamento na versão 4 ambiciosa, que planeja um número de melhorias e extensões radicais para a linguagem. Porém, mudar de forma radical uma linguagem largamente usada pode ser politicamente difícil, e o trabalho na versão 4 foi abandonado em 2008, e conduzido para a 5ª edição que saiu em 2009. Estamos agora esperando que todos os maiores navegadores suportem a 5 edição, que é a linguagem da versão que este livro vai focar. O trabalho na 6ª edição está em curso.

Navegadores web não são as únicas plataformas que o JavaScript é usado. Alguns banco de dados, como MongoDB e CouchDB, usam JavaScript como sua linguagem de consulta e script. Muitas plataformas para desktop e de programação no servidor, mais notável o projeto *Node.JS*, sujeito do capítulo (AINDA NÃO ESCRITO), fornecem um poderoso ambiente de programação JavaScript fora do navegador.

Código, e o que fazer com ele

Código é o texto que compõe os programas. Muitos capítulos deste livro contém muito código. Em minha experiência, escrever e ler códigos é uma importante parte do aprendizado da programação. Tenta não apenas olhar sobre os códigos, leia-os atenciosamente e os entenda. Isto pode ser lento e consufo no início, mas eu prometo que você vai rapidamente pegar o jeito. O mesmo acontece para os exercícios. Não assuma que você os entendeu até que você realmente tenha escrito uma solução que funcione.

Eu recomendo que você teste suas soluções dos exercícios em um interpretador JavaScript real, para obter um feedback se o que você fez está funcionando ou não, e, esperançosamente, ser incentivado a experimentar e ir além dos exercícios.

Quando ler este livro no seu navegador, você pode editar (e rodar) os programas exemplo clicando neles.

Rodando programas JavaScript fora do contexto deste livro é possível também. Você pode optar por instalar o node.js, e ler a documentação para conhecer como usá-lo para avaliar arquivos de texto que contém programas. Ou você pode usar o console de desenvolvedores no navegador (tipicamente encontrado no menu “tools” ou “developer”) e divertir-se nele. No capítulo (CORRIGIR!), o jeito que os programas são embutidos em paǵinas web (arquivos HTML) é explicado. Entretanto, você pode verificar em http://jsbin.com por outra interface amigável para rodar código JavaScript no navegador.

Conveções Tipográficas

Neste livro, texto escrito em fonte `monoespaçada` deve ser entendido por representações de elementos dos programas - algumas vezes são fragmentos auto-suficientes, e algumas vezes eles somente referenciam para alguma parte de um programa próximo. Programas (que você já viu um pouco), são escritos assim:

```
function fac(n) {
        if (n == 0)
                return 1;
        else
                return fac(n - 1) * n;
}
```

Algumas vezes, para mostrar a saída que o programa produz, a mesma será escrita abaixo dele, com duas barras e uma seta na frente:

```
console.log(fac(8));
// → 40320
```

Boa Sorte!

Valores, Tipos e Operadores

Dentro do mundo do computador, há somente dados. Nós podemos ler dados, modificar dados, criar dados - mas coisas que não são representadas por dados simplesmente não existem. Todos estes dados são armazenados em longas sequências de bits, e isso portanto é fundamentalmente parecido.

Bits podem ser qualquer tipo de coisa com 2 valores, usualmente descrito como 0 e 1. Dentro do computador, eles tomam formas como uma carga elétrica alta ou baixa, um forte ou fraco sinal, ou um ponto brilhante ou sem brilho na superfície de um CD. Qualquer pedaço de uma discreta informação, pode ser reduzida para uma sequência de 0 e 1, e então representanda por bits.

Como um exemplo, pense sobre a maneira que o número 13 pode ser armazenado em bits. A forma usual de se fazer esta analogia é a forma de escrevermos números decimais, mas ao invés de 10 dígitos, temos apenas 2. E, ao invés de o valor de um dígito aumentar dez vezes sobre o dígito após ele, o valor aumenta por um fator 2. Estes são os bits que compõem o número treze, com o valor dos dígitos mostrados abaixo deles:

```
0   0   0   0   1   1   0   1
128 64  32  16  8   4   2   1
```

Então este é o 00001101, ou 8 + 4 + 1, que equivale a 13.

Valores

Imagine um mar de bits. Um oceano deles. Um típico computador moderno vai ter em torno de trinta bilhões de bits reservados em seu armazenamento de dados volátil (em oposição ao armazenamento não volátil - o disco rígido ou não equivalente - que tende a ter algumas ordens de magnitude mais).

Bit Sea

Para trabalhar com estes sem se perder, nós temos que estruturá-los de alguma forma. Uma forma de fazer é agrupá-los dentro de pedaços que representam uma simples parte de informação. Em um ambiente JavaScript, todo os dados são separados em coisas chamadas *valores*, grupos de bits que representam um pedaço de dado coerente.

Embora todos os valores sejam feitos da mesma coisa uniforme, eles desempenham papéis diferentes. Todo valor tem um tipo, que determina o tipo de papel que desempenha. Temos seis tipos básicos de valores no JavaScript: números, strings, booleans, objetos, funções e valores indefinidos.

> Em inglês: [number, string, boolean, object, function, undefined]

Para criar um valor, deve-se simplesmente invocar seu nome. Isto é muito conveniente. Você não tem que recolher material para construir seus valores ou pagar por eles; você só chama por um, e pronto, você o tem. Eles não são criados com ar, obviamente. Todo valor tem que ser armazenado em algum lugar, e se você quer usar uma quantidade gigante deles, ao mesmo tempo você deve rodar sobre os bits. Felizmente, este é um problema somente se você os quiser simultaneamente. Assim que você não usar mais um valor, ele será dissipado, deixando para trás os bits para serem reciclados e se tornarem materiais para a próxima geração de valores.

Este capítulo introduz os elementos atômicos dos programas JavaScript: Simples tipos de valores, e operadores que podem atuar em cada valor.

Números

Valores do tipo *numbers* são, previsivelmente, valor numéricos. Em um programa JavaScript, eles são escritos usualmente assim:

```
1 13
```

Coloque isto em um programa, e isto vai gerar o bir padrão para o número 13 começar a existir dentro do computador.

O JavaScript usa um número fixo de bits, 64 deles, para armazenar um único valor numérico. Isto significa que existe uma quantidade limite de tipos diferentes de números que podem ser representados - há muitos padrões diferentes que você pode criar com 64 bits. O conjunto de números podem ser representados por N dígitos decimais é 10^N. Similarmente, o conjunto de números que podem ser representados por 64 dígitos binários é 2^{64}, que é mais ou menos 18 quintilhões (um 18 com 18 zeros após ele).

Isto é muito. Números exponenciais tem o hábito de ficarem grandes. Já foi o tempo que as memórias eram pequenas e as pessoas tendiam a usar grupos de 8 ou 16 bits para representar estes números. Era fácil de acidentalmente "transbordarem" estes pequenos números. Hoje, temos o luxo de somente preocupar quando realmente lidamos com números astronômicos.

Todos os números abaixo de 18 quintilhões cabem no JavaScript *number*. Estes bits também armazenam números negativos, onde um destes sinais é usado para guardar o sinal do número. Uma grande questão é que números não inteiros podem ser representados. Para fazer isso, alguns bits são usados para guardar a posição do ponto decimal do número. O maior número não inteiro que pode ser armazenado está na faixa de 9 quadrilhões (15 zeros) - que continua muito grande.

Números fracionados são escritos usando o ponto:

```
1  9.81
```

Para grandes números ou números pequenos, podemos usar a notação científica adicionando um 'e', seguido do expoente:

```
1  2.998e8
```

Isto é 2.998 x 10^8 = 299800000.

Cálculos com números inteiros (também chamados *integers*) menores que os mencionados 9 quadrilhões são garantidos de sempre serem precisos. Infelizmente cálculos com números fracionários não são, geralmente. Como π (pi) não pode ser precisamente expresso por uma quantidade finita de dígitos decimais, vários números perdem a precisão quando somente 64 bits estão disponíveis para armazená-los. Isto é uma vergonha, porém causa problemas somente em situações muito específicas. A coisa importante é estar ciente disto e tratar números fracionários digitais como aproximações, não como valores precisos.

Aritmética

A principal coisa a se fazer com números é aritmética. Operações aritméticas como adição e multiplicação pegam 2 valor de números e produzem um novo número a partir deles. Aqui vemos como eles são no JavaScript:

```
1  100 + 4 * 11
```

Os símbolos + e * são chamados *operadores*. O primeiro representa adição, e o segundo representa multiplicação. Colocando um operador entre 2 valores faz com que se aplique o mesmo, produzindo um novo valor.

O próximo exemplo significa "adicione 4 e 100, e multiplique o resultado por 11", ou é a multiplicação feita antes da adição? Como você deve ter pensado, a multiplicação acontece primeiro. Mas, como na matemática, isto pode ser mudado envolvendo a adição com os parênteses:

```
(100 + 4) * 11
```

Para subtração, este é o operador `-`, e para a divisão usamos este operador `/`.

Quando operadores aparecem juntos sem parênteses, a ordem que eles vão ser aplicados é determinada pela *precedência* dos operadores. O exemplo mostra que a multiplicação vem antes da adição. `/` tem a mesma precedência de `*`. Igualmente para `+` e `-`. Quando múltiplos operadores com a mesma precedência estão próximos uns aos outros (como em `1 - 2 + 1`), eles são aplicados da esquerda para a direita.

Estas regras de precedência não é algo que você deva se preocupar. Quando em dúvida, somente adicione parênteses.

Há mais um operador aritmético, que possivelmente é menos familiar. O símbolo `%` é usado para representar o *restante* da operação. `X % Y` é o restante da divisão de `X` por `Y`. Por exemplo, `314 % 100` produz `14`, e `144 % 12` nos dá `0`. A precedência deste operador é igual a da multiplicação e divisão. Você também pode ver este operador sendo referido como "modulo" (porém tecnicamente "restante" é mais preciso).

Números Especiais

Existem 3 valores especiais no JavaScript que são considerados números, mas não comportam-se como números normais.

Os dois primeiros são `Infinity` e `-Infinity`, que são usados para representar os infinitos positivo e negativo. `Infinity - 1` continua sendo `Infinity`, e assim por diante. Mas não ponha muita confiança neste tipo de computação *baseada em infinito*, pois é uma matemática pesada, e vai rapidamente levar para nosso próximo número especial: `NaN`.

`NaN` significa "not a number" (não é um número). Você obtém isso quando declara `0 / 0` (zero dividido por zero), `Infinity - Infinity`, ou qualquer número de outra operação numérica que não produz um preciso e significante valor.

Strings

O próximo tipo básico de dado é a *string*. Strings são usadas para representar texto. Elas são escritas delimitando seu conteúdo entre aspas:

```
"Patch my boat with chewing gum"
'Monkeys wave goodbye'
```

Ambas as aspas simples e duplas podem ser usadas para marcar strings - contato que as aspas no início e no fim da string combinem.

Quase tudo pode ser colocado entre aspas, e o JavaScript vai fazer um valor de string com isso. Mas alguns caracteres são difíceis. Você pode imaginar como colocar aspas entre aspas deve ser difícil. Novas linhas, as coisas que você obtém quando pressiona `enter`, também não podem ser colocadas entre aspas - a string tem que estar em uma linha única.

Para ser capaz de ter estes caracteres em uma string, a convenção seguinte é usada: Sempre que um barra invertida `\` é encontrada dentro do texto entre aspas, isto indica que o caracter depois desta tem um significado especial. Uma aspa precedida de uma barra invertida não vai findar a string, mas ser parte dela. Quando um caracter 'n' correr depois de uma barra invertida, será interpretado como uma nova linha. Similarmente, um 't' depois da barra invertida significa o caracter tab. Veja a string seguinte:

```
"This is the first line\nAnd this is the second"
```

O verdadeiro texto contido é:

```
This is the first line
And this is the second
```

Existe, obviamente, situações onde você quer uma barra invertida em uma string apenas como uma barra invertida. não um código especial. Se duas barras invertidas estiverem seguidas uma da outra, elas se anulam, e somente uma vai ser deixada no valor da string resultante. Assim é como a string `A newline character is written like "\n" can be written:`

```
"A newline character is written like \"\\n\"."
```

Strings não podem ser divididas, multiplicadas ou subtraídas, mas o operador + pode ser usado nelas. Ele não adiciona, mas concatena - ele cola duas strings juntas. A linha seguinte vai produzir a string `concatenate:`

```
"con" + "cat" + "e" + "nate"
```

Existem outras maneiras de manipular strings, que nós vamos discutir quando entrarmos nós métodos no capítulo 4.

Operadores Unários

Nem todos operadores são símbolos. Alguns são palavras escritas. Um exemplo é o operador `typeof`, que produz uma string com o valor do tipo dado para fornecido para avaliação.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

Nós vamos usar `console.log` nos códigos exemplo para indicar que nós queremos ver o resultado da avaliação de algo. Quando você roda algum código, o valor produzido vai ser mostrado na tela - de alguma forma, dependendo do ambiente JavaScript que você usa para rodá-lo.

Os outros operadores que vimos sempre operam com 2 valores; `typeof` pega somente um. Operadores que usam 2 valores são chamados operadores *binários*, enquanto aqueles que pegam um são chamados operadores *unários*. O operador menos `-` pode ser usado como operador binário e unário.

```
console.log(- (10 - 2))
// → -8
```

Valores Booleanos

As vezes, você vai precisar de um valor que simplesmente distingue entre 2 possibilidades, "sim" ou "não", ou "ligado" e "desligado". Para isso o JavaScript tem um tipo *booleano*, que tem apenas dois valores, `true` e `false` (que são escritos com estas palavras mesmo).

Comparações

Aqui temos uma maneira de produzir valores booleanos:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

Os sinais > e < são tradicionalmente símbolos para "é maior que" e "é menor que". Eles são operadores binários, e o resultado da aplicação deles é um valor booleano que indica se eles são verdadeiros neste caso.

Strings podem ser comparadas da mesma forma:

```
console.log("Aardvark" < "Zoroaster")
// → true
```

A maneira que as strings são ordenadas é mais ou menos alfabética: Letras maiúsculas são sempre "menores" que as minúsculas, então ‘ "Z" < "a" ‘ é `true`, e caracteres não alfabéticos (‘!’, ‘-‘, e assim por diante) são também incluídos na ordenação. A maneira real da comparação é feita baseada no padrão *Unicode*. Este padrão atribui um número a todo caracter virtual que pode ser usado, incluindo caracteres da Grécia, Arábia, Japão, Tamil e por ai vai. Ter estes números é prático para guardar strings dentro do computador - você pode representá-los como uma sequência de números. Quando se compara strings, o JavaScript vai sobre elas da esquerda para a direita, comparando os códigos numéricos dos caracteres um por um.

Outros operadores similares são `>=` (maior que ou igual a), `<=` (menor que ou igual a), `==` (igual a) e `!==` (não igual a).

```
console.log("Itchy" != "Scratchy")
// → true
```

Há somente um valor no JavaScript que não é igual a ele mesmo, que é o `NaN` (not a number). Pois ele denota o resultado de uma computação sem sentido, e nunca igual a nenhum de resultado de nenhuma *outra* computação absurda.

```
console.log(NaN == NaN)
// → false
```

Operadores Lógicos

Temos também algumas operações que podem ser aplicadas aos valores booleanos. O JavaScript suporta 3 operadores lógicos: *e*, *ou* ou *não*.

O operador `&&` representa o **e** lógico. É um operador binário, e seu resultado é `true` (verdadeiro) somente se ambos os valores dados a ele forem `true`.

```
console.log(true && false);
// → false

console.log(true && true);
// → true
```

O operador `||` denota ao **ou** lógico. Ele produz `true` se algum dos valores fornecidos for `true`:

```
console.log(false || true);
// → true

console.log(false || false);
// → false
```

Não é escrito com uma exclamação `!`. É um operador unário que inverte o valor dado a ele - `!true` produz `false` e `!false` produz `true`.

Quando misturamos estes operadores booleanos com operadores aritméticos e outros operadores, não é sempre óbvio quando o parênteses é necessário. Na prática, você precisa conhecer sobre os operadores que vimos antes, e que `||` tem o mais baixo nível de precedência, seguido do `&&`, e então os operadores de comparação (`>`, `==`, e outros), e depois o resto. Sendo assim, como vemos na expressão abaixo, os parênteses poucas vezes são necessários.

```
1 + 1 == 2 || 10 * 10 <= 100
```

Valores Indefinidos

Temos dois valores especiais, `null` e `undefined`, que são usados para denotar a ausência de valores significativos. Eles são por si próprios valores, porém valores que não levam informação.

Muitas operações na linguagem que não produzem valores significativos (vamos ver algumas no próximo capítulo) vão produzir `undefined`, simplesmente porque elas tem que retornar *algum* valor.

A diferença de significado entre `undefined` e `null` é em grande parte desinteressante e um acidente no projeto do JavaScript. Nos casos que você realmente tiver que se preocupar com estes valores, eu recomendo tratá-los como substituíveis (mais sobre isso em um momento).

Conversão Automática de Tipo

Na introdução, eu mencionei que o JavaScript não iria atrapalhá-lo e aceitaria quase qualquer coisa que você o fornecesse, mesmo quando isso é confuso e errado. Isto é muito bem demonstrado por esta expressão:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

Quando um operador é aplicado a um tipo de valor "errado", ele vai silenciosamente converter este valor para o tipo que quiser, usando um conjunto de regras que frequentemente não são as que você expera. O `null`na primeira expressão torna-se `0`, o `"5"` na segunda expressão se torna `5` (de string para número), ainda na terceira expressão, o `+` tenta a concatenação de strings antes de tentar a adição numérica, o `1`é convertido em `"1"` (de número para string).

Quando algo que não pode ser mapeado como um número de forma óbvia, do tipo `"five"` ou `undefined` é convertido para um número, o valor `NaN` é produzido. Operações aritméticas com `NaN` continuam produzindo `NaN`, então se você encontrar alguns destes resultados em algum local inesperado, procure por conversões acidentais de tipo.

Quando comparamos coisas do mesmo tipo usando `==`, o resultado é bastante fácil de se prever - você vai obter `true` quando ambos os valores forem os mesmos. Mas quando os tipos diferem, o JavaScript usa um complicado e confuso conjunto de regras para determinar o que fazer. Eu não vou explicar isto precisamente, mas na maioria dos casos irá ocorrer a tentativa de converção de um dos valores para o tipo do outro valor. Contudo, quando `null`ou `undefined`ocorrem em algum dos lados do operador, isso produzirá `true` somente se ambos os lados forem `null` ou `undefined`.

A última parte do comportamento é frequentemente muito útil. Quando você quer testar se um valor tem um valor real, em vez de ser `null` ou `undefined`, você pode simplesmente compará-lo a `null` com o operador `==` (ou `!=`).

Mas e se você quiser testar se algo se refere ao valor preciso `false`? As regras de converção de strings e números para valores booleanos afirmam que `0`, `NaN` e empty strings contam como `false`, enquanto todos os outros valores contam como `true`. Por causa disso, expressões como `0 == false` e `"" == false` retornam `true`. Para casos assim, onde você **não** quer qualquer conversão automática de tipos acontecendo, existem dois tipos extras de operadores: `===` e `!==`. O primeiro teste se o valor é precisamente igual ao outro, e o segundo testa se ele não é precisamente igual. Então `"" === false` é falso como esperado.

Usar os operadores de comparação de três caracteres defensivamente, para prevenir inesperadas conversões de tipo que o farão tropeçar, é algo que eu recomendo. Mas quando você tem certeza de que os tipos de ambos os lados serão iguais, ou que eles vão ser ambos `null`/`undefined`, não há problemas em usar os operadores curtos.

O Curto-Circuito de && e ||

Os operadores lógicos `&&` e `||` tem uma maneira peculiar de lidar com valores de tipos diferentes. Eles vão converter o valor à sua esquerda para o tipo booleano a fim de decidir o que fazer, mas então, dependendo do operador e do resultado da conversão, eles ou retornam o valor à esquerda *original*, ou o valor à direita.

O operador `||` vai retornar o valor à sua esquerda quando ele puder ser convertido em `true`, ou valor à sua direita caso contrário. Ele faz a coisa certa para valores booleanos, e vai fazer algo análogo para valores de outros tipos. Isso é muito útil, pois permite que o operador seja usado para voltar um determinado valor predefinido.

```
console.log(null || "user")
// → user
console.log("Karl" || "user")
// → Karl
```

O operador `&&` trabalha similarmente, mas ao contrário. Quando o valor à sua esquerda é algo que se torne `false`, ele retorna o valor, e caso contrário ele retorna o valor à sua direita.

Outro importante propriedade destes 2 operadores é que a expressão a sua direita é avaliada somente quando necessário. No caso de `true || X`, não importa o que `X` é - pode ser uma expressão que faça algo *terrível* - o resultado vai ser verdadeiro, e `X` nunca é avaliado. O mesmo acontece para `false && X`, que é falso, e vai ignorar `X`.

Resumo

Nós vimos 4 tipos de valores do JavaScript neste capítulo. Números, strings, booleanos e valores indefinidos.

Alguns valores são criados digitando seu nome (`true`, `null`) ou valores (`13`, `"abc"`). Eles podem ser combinados e transformados com operadores. Nós vimos operadores binários para aritmética (`+`, `-`, `*`, `/`, e `%`), um para concatenação de string (`+`), comparação (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`) e lógica (`&&`, `||`), como também vários operadores unários (`-` para negativar um número, `!` para negar uma lógica, e `typeof` para encontrar o tipo do valor).

Isto lhe dá informação suficiente para usar o JavaScript como uma calculadora de bolso, mas não muito mais. O próximo capítulo vai começar a amarrar essas operações básicas conjuntamente dentro de programas básicos.

Estrutura do Programa

Este é o ponto onde nós começamos a fazer coisas que podem realmente ser chamadas *programação*. Nós vamos expandir nosso domínio da linguagem JavaScript, para além dos substantivos e fragmentos de sentenças que nós vimos anteriormente, para o ponto onde poderemos realmente expressar algo mais significativo.

Expressões e Afirmações

No capítulo anterior, nós criamos alguns valores e então aplicamos operadores para então obter novos valores. Criar valores desta forma é uma parte essencial de todo programa JavaScript, mas isso é somente uma parte. Um fragmento de código que produz um valor é chamado de *expressão*. Todo valor que é escrito literalmente (como `22` ou `"psychoanalysis"`) é uma expressão. Uma expressão entre parênteses é também uma expressão, e também um operador binário aplicado a duas expressões, ou um unário aplicado a uma.

Isso mostra parte da beleza da interface baseada na linguagem. Expressões podem ser encadeadas de forma muito semelhante das sub-frases usadas na liguagem humana - uma sub-frase pode conter sua própria sub-frase, e assim por diante. Isto nos permite combinar expressções para expressar arbitrariamente computações complexas.

Se uma expressão corresponde a um fragmento de sentença, uma *afirmação*, no JavaScript, corresponde a uma frase completa em linguagem humana. Um programa é simplesmente uma lista de afirmações.

O tipo mais simples de afirmação é uma expressão com um ponto e vírgula depois dela. Este é o programa:

```
1;
!false;
```

É um programa inútil, entretanto. Uma expressão pode ser apenas para produzir um valor, que pode então ser usado para fechar a expressão. Uma declaração vale por si só, e só equivale a alguma coisa, se ela afeta em algo. Ela pode mostrar algo na tela - que conta como mudar algo - ou pode mudar internamente o stato da máquina de uma forma que vai afetar outras declarações que irão vir. Estas mudanças são chamadas *efeitos colaterais*. As afirmações nos exemplos anterios somente produzem o valor `1` e `true` e então imediatamente os jogam fora novamente. Não deixam nenhuma impressão no mundo. Quando executamos o programa, nada acontece.

Ponto e vírgula

Em alguns casos, o JavaScript permite que você omita o ponto e vírgula no fim de uma declaração. Em outros casos, ele deve estar lá, ou coisas estranhas irão acontecer. As regras para quando ele pode ser seguramente omitido são um pouco complexas e propensas a erro. Neste livro, todas as declarações que precisam de ponto e vírgula vão sempre terminar com um. Eu recomendo a você fazer o mesmo em seus programas, ao menos até que você aprenda mais sobre as sutilezas envolvidas em retirar o ponto e vírgula.

Variáveis

Como um programa mantém um estado interno? Como ele se lembra das coisas? Nós vimos como produzir novos valores com valores antigos, mas isso não altera os valores antigos, e o valor novo deve ser imediatamente usado ou vai ser dissipado. Para pegar e guardar valores, o JavaScript fornece um coisa chamada *variável*.

```
var caught = 5 * 5;
```

E isso nos dá um segundo tipo de declaração. A palavra especial (*palavra-chave*) `var` indica que esta sentença vai definir uma variável. Ela é seguida pelo nome da variável e, se nós queremos dá-la imediatamente um valor, por um operador = e uma expressão.

A declaração anterior criou uma variável chamada `caught` e usou-a para armazenar o valor que foi produzido pela multiplicação 5 por 5.

Depois de uma variável ter sido definida, seu nome pode ser usado como uma expressão. O valor como uma expressão é o valor atual mantido pela variável. Aqui temos um exemplo:

```
var ten = 10;
console.log(ten * ten);
// 100
```

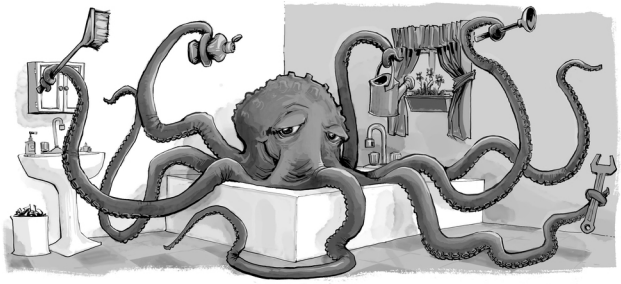
Nomes de variáveis podem ser quase qualquer palavra, menos as reservadas para palavras-chave (como `var`). Não pode haver espaços incluídos. Dígitos podem também ser parte dos nomes de variáveis - `catch22` é um nome válido, por exemplo - mas um nome não pode iniciar com um dígito. O nome de uma variável não pode incluir pontuação, exceto pelos caracteres $ e _.

Quando uma variável aponta para um valor, isso não significa que estará ligada ao valor para sempre. O operador = pode ser usado a qualquer hora em variáveis existentes para desconectá-las de seu valor atual e então apontá-las para um novo:

```
var mood = "light";
console.log(mood);
// ligth
mood = "dark";
console.log(mood);
// dark
```

Você deve imaginar variáveis como tentáculos, ao invés de caixas. Elas não *contém* valores; elas os *agarram* - duas variáveis podem referenciar o mesmo valor. Somente os valores que o programa mantém tem o poder de ser acessado por ele. Quando você precisa de lembrar de algo, você aumenta o tentáculo para segurar ou recoloca um de seus tentáculos existentes para fazer isso.

Quando você define uma variável sem fornecer um valor a ela, o tentáculo fica conceitualmente no ar - ele não tem nada para segurar. Quando você pergunta por um valor em um lugar vazio, você recebe o valor `undefined`.



Polvo

Um exemplo. Para lembrar da quantidade de dólares que Luigi ainda lhe deve, você cria uma variável. E então quando ele lhe paga 35 dólares, você dá a essa variável um novo valor.

```
var luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// 105
```

Palavras-chave e Palavras Reservadas

Nomes que tem um significado especial, como `var`, não podem ser usados como nomes de variáveis. Estes são chamados *keywords* (palavras-chave). Existe também um número de palavras que são "reservadas para uso" em futuras versões do JavaScript. Estas também não são oficialmente autorizadas a serem utilizadas como nomes de variáveis, embora alguns ambientes JavaScript as permitam. A lista completa de palavras-chave e palavras reservadas é bastante longa:

```
break case catch continue debugger default delete do else false finally for function if
implements in instanceof interface let new null package private protected public return
static switch throw true try typeof var void while with yield this
```

Não se preocupe em memorizá-las, mas lembre-se que este pode ser o problema quando algo não funcionar como o esperado.

O Ambiente

A coleção de variáveis e seus valores que existem em um determinado tempo é chamado de `environment` (ambiente). Quando um programa inicia, o ambiente não está vazio. Ele irá conter no mínimo o número de variáveis que fazem parte do padrão da linguagem. E na maioria das vezes haverá um conjunto adicional de variáveis que fornecem maneiras de interagir com o sistema envolvido. Por exemplo, em um navegador, existem variáveis que apontam para funcionalidades que permitem a você inspecionar e influenciar no atual carregamento do website, e ler a entrada do mouse e teclado da pessoa que está usando o navegador.

Funções

Muitos dos valores fornecidos no ambiente padrão são do tipo *function* (função). Uma função é um pedaço de programa envolvido por um valor. Este valor pode ser aplicado, a fim de executar o programa envolvido. Por exemplo, no ambiente do navegador, a variável `alert` detém uma função que mostra uma pequena caixa de diálogo com uma mensagem. É usada assim:

```
1 alert("Bom dia!");
```

JSFiddle[2]

Executar uma função é denominado *invocando*, *chamando* ou *aplicando* uma função. A notação para fazer isso é colocar um parênteses depois de uma expressão que produza um valor de uma função. Normalmente você vai referenciar diretamente a uma variável que detém uma função. Os valores entre os parênteses são dados ao programa dentro da função. No exemplo, a função `alert` usou a string que foi dada como o texto para ser mostrado na caixa de diálogo. Valores dados a funções são chamados *argumentos*. A função `alert` precisa somente de um, mas outras funções podem precisar de diferentes quantidades ou tipos de argumentos.

A Função `console.log`

A função `alert` pode ser útil como saída do dispositivo quando experimentada, mas clicar sempre em todas estas pequenas janelas vai lhe irritar. Nos exemplos passados, nós usamos `console.log` para retornar valores. A maioria dos sistemas JavaScript (incluindo todos os navegadores modernos e o node.js), fornecem uma função `console.log` que escrevem seus argumentos em algum texto na saída

[2] http://jsfiddle.net/K3Fe3/

do dispositivo. Nos navegadores, a saída fica no console JavaScript, que é uma parte da interface do usuário escondida por padrão, mas que pode ser encontrada navegando no menu e encontrando um item do tipo "web console" ou "developer tools" (ferramenta do desenvolvedor), usualmente dentro do sub-menu "Tools" (ferramentas) ou "Developer" (desenvolvedor).

Quando rodamos os exemplos, ou seu próprio código, nas páginas deste livro, o `console.log` vai mostrar embaixo o exemplo, ao invés de ser no console JavaScript.

```
var x = 30;
console.log("o valro de x é ", x);
// o valor de x é 30
```

Embora eu tenha afirmado que nomes de variáveis não podem conter pontos, `console.log` claramente contém um ponto. Eu não tinha mentido para você. Esta não é uma simples variável, mas na verdade uma expressão que retorna o campo `log` do valor contido na variável `console`. Nós vamos entender o que isso significa no capítulo 4.

Retornando Valores

Mostrar uma caixa de diálogo ou escrever texto na tela é um efeito colateral. Muitas funções são úteis por causa dos efeitos que elas produzem. É também possível para uma função produzir um valor, no caso dela não ser necessário um efeito colateral. Por exemplo, temos a função `Math.max`, que pega dois números e retorna o maior entre eles:

```
console.log(Math.max(2, 4));
```

Quando uma função produz um valor, é dito que ela *retorna* (return) ele. Por coisas que produzem valores sempre serem expressões no JavaScript, funções chamadas podem ser usadas como parte de uma grande expressão:

```
console.log(Math.min(2, 4) + 100);
```

O próximo capítulo explica como nós podemos escrever nossas próprias funções.

Solicitar e Confirmar

O ambiente fornecido pelos navegadores contém algumas outras funções para mostrar janelas. Você pode perguntar a um usuário uma questão "Ok/Cancel" usando `confirm`. Isto retorna um booleano: `true` se o usuário clica em OK e `false` se o usuário clica em Cancel.

```
confirm("");
```

JSFiddle[3]

`prompt` pode ser usado para criar uma questão "aberta". O primeiro argumento é a questão; o segundo é o texto que o usuário inicia. Uma linha do texto pode ser escrita dentro da janela de diálogo, e a função vai retornar isso como uma string.

```
prompt("Diga-me algo que você saiba.", "...");
```

JSFiddle[4]

Estas duas funções não são muito usadas na programação moderna para web, principalmente porque você não tem controle sobre o modo que a janela vai aparecer, mas ela são úteis para programas para brincar e experimentos.

Fluxo de Controle

Quando seu programa contém mais que uma declaração, as declarações são executadas, previsivelmente, de cima para baixo. Como um exemplo básico, este programa tem duas declarações. A primeira pergunta ao usuário por um número, e a segunda, que é executada posteriormente, mostra o quadrado deste número:

```
var theNumber = Number(prompt("Pick a number", ""));
alert("Your number is the square root de " + theNumber * theNumber);
```

A função `Number` converte o valor para um número, que nós usamos porque o resultado de `prompt` é um valor `string`, e nós queremos um número. Existem funções similares chamadas `String` e `Boolean` que convertem valores para estes tipos.

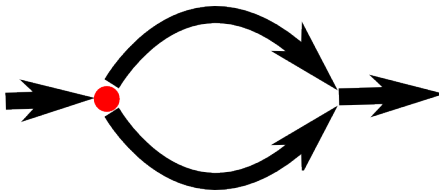
Aqui podemos ver uma representação bem trivial do fluxo de controle em linha reta:

Linha de Fluxo de Controle

[3] http://jsfiddle.net/4gX5m/
[4] http://jsfiddle.net/bsfxA/

Execução Condicional

Executando declarações em ordem linear não é a única opção que temos. Uma requisição comum é a execução condicional, onde nós escolhemos entre duas rotas diferentes baseado em um valor booleano.



Fluxo de Controle If

A execução condicional é escrita em JavaScript com a palavra-chave `if`. De forma simplificada, nós somente queremos que algum código seja executado se (`if`), e somente se, uma certa condição existir. Por exemplo, no programa anterior, nós poderíamos querer mostrar o quadrado da entrada somente se a entrada for realmente um número.

```
var theNumber = Number(prompt("Digite um número", ""));
if (!isNaN(theNumber))
        alert("Seu número é a raiz quadrada de " +
                        theNumber * theNumber);
```

Com essa modificação, se você entrar com “queijo” - ou não digitar nada - nenhuma saída será retornada.

A palavra chave `if` é usada para executar ou pular uma declaração dependendo do valor da expressão booleana. Ela é sempre seguida por uma expressão entre parênteses, e então uma declaração.

A função `isNaN` é uma função padrão que retorna `true` se o argumento dado a ela é `NaN`. A função `Number` retorna `NaN` quando você fornece a ela uma string que não representa um número válido. Então, a condição expressa “salvo que `theNumber` não seja um número, faça isso”.

Frequentemente você tem não apenas código que deve ser executado quando uma certa condição é verificada, mas também código que manipula outros casos, quando a condição não confere. A palavra-chave `else` pode ser usada, juntamente com `if`, para criar dois caminhos separados e paralelos que executam de acordo com suas condições:

```
var theNumber = Number(prompt("Digite um número", ""));
if (!isNaN(theNumber))
        alert("Seu número é a raiz quadrada de " +
                        theNumber * theNumber);
else
        alert("Ei! Por que você não me deu um número?");
```

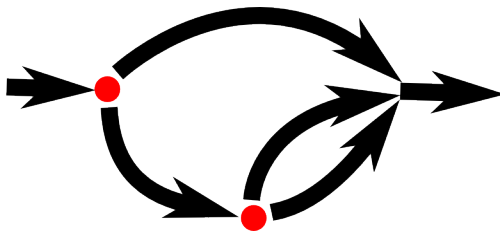
Se nós tivermos mais que dois caminhos que queremos escolher, múltiplos pares de `if`/`else` podem ser "encadeados" conjuntamente. Aqui temos um exemplo:

```
var num = Number(prompt("Digite um número", "0"));

if (num < 0)
        alert("Pequeno");
else if (num < 100)
        alert("Médio");
else
        alert("Grande");
```

Este programa vai primeiramente checar se `num` é menor que 10. Se ele for, ele escolhe essa ramificação, e mostra "Pequeno", e pronto. Se não for, ele pega a ramificação `else`, que contém o segundo `if`. Se a segunda condição (`< 100`) for verdadeira, isso significa que o número está entre 10 e 100, e `Médio` será mostrado. Se não, o segundo e último `else` será escolhido.

O esquema de setas para este programa parece com algo assim:



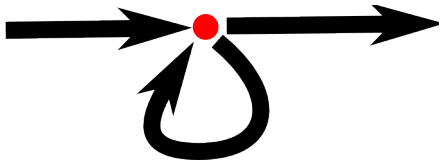
Fluxo de Controle do Aninhamento if

Loops While e Do

Considere um programa que imprimi todos os numéros pares de 0 a 12. Uma forma de se escrever isto é a seguinte:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

Isto funciona, mas a ideia de escrever um programa é a de fazer algo ser *menos* trabalhoso, e não mais. Se nós precisarmos de todos os números pares menores que 1.000, o programa anterior se torna impraticável. O que nós precisamos é de uma forma de repetir código - um *loop*.



Fluxo de Controle do Loop

O fluxo de controle do loop nos permite voltar a um mesmo ponto do programa onde estávamos anteriormente, e repetí-lo no contexto da nossa declaração atual do programa. Se nós combinarmos isso com uma variável que conta, nós podemos fazer isso:

```
var number = 0;
while (number <= 12) {
        console.log(number);
        number = number + 2;
}
// 0
// 2
// etc...
```

Uma declaração iniciando com a palavra `while` cria um loop. A palavra `while` é seguida por uma expressão em parênteses e então uma declaração, como um `if`. O loop age continuando a executar a declaração enquanto a expressão produzir um valor que é `true`, quando convertido para o tipo booleano.

Por causa de querermos fazer duas coisas dentro do loop, imprimir o número atual e adicionar 2 a nossa variável, nós envolvemos as duas declarações com chaves `{}`. Chaves, para declarações, são similares aos parênteses para expressões, elas as agrupam, fazendo que sejam vistas como uma simples declaração.

Então, a variável `number` demonstra o caminho que a variável pode tomar no progresso do programa. Toda hora que o loop se repete, ele é incrementado por 2. Então, no início de toda repetição, ele é

comparado com o número 12 para decidir se o programa terminou todo o trabalho que era pretendido a se fazer.

Como exemplo de algo realmente útil, podemos agora escrever um programa que calcula e mostra o valor de 2^{10} (2 elevado a décima potência). Nós usamos duas variáveis: uma para manter o registro do nosso resultado e uma para contar quantas vezes multiplicamos este resultado por 2. O loop teste se a segunda variável já atingiu 10 e então atualiza ambas variáveis.

```
var result = 1;
var counter = 0;
while (counter < 10) {
        result = result * 2;
        counter = counter + 1;
}
console.log(result);
// 1024
```

O contador pode também iniciar com 1 e checar por `<= 10`, mas, por razões que vamos ver mais a frente, é uma boa ideia usar a contagem iniciando com 0.

Uma estrutura muito similar é o loop `do`. Ele difere somente em um ponto do loop `while`: ele sempre vai executar este a declaração uma vez, e somente então vai iniciar o teste e verificar se precisa pausar. Para demonstrar, o teste é escrito abaixo do corpo do loop:

```
do {
        var name = prompt("Who are you?");
} while (!name) {
        console.log(name);
}
```

Isto vai forçar você a entrar com um nome, perguntando de novo e de novo até que pegue algo que não é uma string vazia. (Aplicando o operador `!` vamos converter o valor para o tipo booleano negando o mesmo, e todas as strings exceto `""` convertem em `true`).

Indentando Código

Você deve ter notado os espaços que coloco na frente das declarações. Eles não são necessários - o computador vai aceitar o programa muito bem sem eles. De fato, as quebras de linhas nos programas também são opcionais. Você pode escrevê-las como uma linha simples se quiser. O papel da indentação dentro dos blocos é fazer com que se destaque a estrutura do código. Por causa de novos blocos poderem ser abertos dentro de outros blocos, em códigos complexos isto pode se tornar difícil de se ver onde um bloco terminar e outro inicia. Com uma indentação correta, o formato visual de um programa corresponde a forma do bloco dentro dele. Eu gosto de usar 2 espaços para cada bloco aberto, mas gostos são diferentes - algumas pessoas usam quatro espaços, e algumas usam caracteres tab.

Loops For

Loops sempre seguem o mesmo padrão, como visto nos exemplos acima do `while`. Primeiro, uma variável "contadora" é criada. Esta variável registra o progresso do loop. Então, vem o loop `while`, que testa a expressão geralmente checando se o contador alcançou algum limite. No fim do corpo do loop, o contador é atualizado para o registro do progresso.

Por este padrão ser muito comum, o JavaScript e linguagens similares oferecem uma forma ligeiramente mais curta e compreensiva:

```
for (var number = 0; number <= 12; number = number + 2)
        console.log(number);
// 0
// 2
// etc
```

Este programa é exatamente o equivalente ao exemplo anterior de *número-sempre-impresso*. A única alteração é que todas as declarações que estão relacionadas ao "estado" do loop estão agora em somente uma linha.

O parênteses após a palavra-chave `for` é obrigado a conter dois "ponto e vírgula" (`;`). A parte antes do primeiro ponto e vírgula *inicializa* o loop, normalmente definindo uma variável. A segunda parte é a expressão que *checa* se o loop vai continuar. A parte final *atualiza* o estado do loop após cada interação. Na maioria dos casos, isto é mais enxuto e limpo que uma construção `while`.

Aqui temos o código que computa 2^{10}, usando `for` ao invés de `while`:

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1)
  result = result * 2;
console.log(result);
//1024
```

Note que mesmo se não abrirmos um bloco com `{`, a declaraão no loop continua indentada com dois espaços para deixar claro que ela "pertence" a linha anterior a ela.

Saindo de um Loop

Ter uma condição que produza `false` não é a única maneira que um loop pode parar. Existe uma declaração especial, `break`, que tem o efeito de pular imediatamente fora do loop em questão.

Este programa encontra o primeiro número que é maior ou igual a 20, e divisível por 7:

```
for (var current = 20; ; current++) {
        if (current % 7 == 0)
                break;
}
console.log(current);
// 21
```

O truque com o operador de resto `%` é uma maneira fácil de testar se um número é divisível por outro número. Se for, o resto da divisão é zero.

A construção `for` neste exemplo não tem uma parte que checa pelo fim do loop. Isso significa que essa tarefa depende da declaração `break` dentro dela para a fazer parar.

Se a declaração `break` faltar, ou acidentalmente tivermos uma condição que sempre produz `true`, você terá o chamado *loop infinito*. Um programa rodando um loop infinito nunca vai parar de rodar, que é normalmente uma coisa ruim.

Se você criar um loop infinito em algum dos exemplos desta página, você vai ser perguntando se quer parar o script após alguns segundos. Se isso falhar, você terá que fechar a aba que você está trabalhando, ou em alguns navegadores você terá que fechar todo ele, a fim de recuperá-lo.

Atualizando variáveis Suscintamente

Um programa, especialmente quando em loop, frequentemente precisa de "atualizar" uma variável com um valor que é baseado no valor anterior.

```
counter = counter + 1;
```

O JavaScript fornece um atalho para isso:

```
counter += 1;
```

Isto também funciona para várias outras operações, como `result *= 2` para dobrar `result`, ou `counter -= 1` para contar abaixo.

Isto nos permite encurtar nosso exemplo de contagem um pouco mais:

```
for (var number = 0; number <= 12; number += 2)
        console.log(number);
```

Para `counter += 1` e `counter -= 1`, sempre temos um equivalente mais curto: `counter++` e `counter--`

Enviando um Valor com `switch`

É comum para um código se parecer com algo assim:

```
if (variable == "value1") action1();
else if (variable == "value2") action2();
else if (variable == "value3") action3();
else defaultAction();
```

Há um construtor chamado `switch` que visa resolver o "envio" de valores de forma mais direta. Infelizmente, a sintaxe JavaScript usada para isso (que é herdada da linha de linguagens de programação C e Java) é um pouco estranha - frequentemente uma cadeia de declarações `if` continua parecendo melhor. Aqui um exemplo:

```
switch (prompt("Como está o tempo?")) {
        case "rainy":
                console.log("Lembre-se de trazer um guarda-chuva.");
                break;
        case "ensolarado":
                console.log("Vista roupas leves");
        case "nublado":
                console.log("Vá lá fora.");
                break
        default:
                console.log("Tempo desconhecido");
                break;
}
```

Dentro do bloco aberto pelo `switch`, você pode colocar qualquer número de rótulos `case`. O programa vai pular para o rótulo correspondente ao valor que `switch` fornece, ou para `default` se nenhum valor for encontrado. Então ele começa a executar as declarações aqui, e continuar a passar pelos rótulos, até que encontra uma declaração `break`. Em alguns casos, como o caso `"ensolarado"` do exemplo, podemos usá-lo para compartilhar algum código entre casos (ele recomenda *ir lá fora* para ambos os tempos "ensolarado" e "numblado"). Mas cuidado, é muito fácil se esquecer de um `break`, que vai causar ao programa a execução de código que você não gostaria de executar.

Capitalização

Nomes de variáveis não podem conter espaços, no entanto é muito útil usar múltiplas palavras para descrever claramente o quê a variável representa. Suas escolhas para escrever nomes de variáveis com muitas palavras são como estas:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

O primeiro estilo é difícil de ler. Pessoalmente, eu gosto de usar underscores _, embora seja um pouco doloroso de escrever. Entretanto, o padrão das funções JavaScript, e da maioria dos programadores JavaScript, é o de seguir o último estilo - eles capitalizam toda palavra exceto a primeira. Isso não é difícil se acostumar com pequenas coisas assim, e o código com os estilos de nomes mistos pode se tornar desagrádavel para leitura, então vamos seguir esta convenção.

Em poucos casos, como a função `Number`, a primeira letra da variável é também capitalizada. Isso é feito para marcar a função como um construtor. O que um construtor é vai ser esclarecido no capítulo 6. Por agora, o importante é não ser incomodado com esta aparente falta de consistência.

Comentários

Frequentemente, código puro não transmite todas as informações necessárias que você gostaria que tivessem para leitura de humanos, ou transmitem de forma oculta de maneira que você se preocupa se as pessoas realmente vão entendê-lo. Em outros momentos, você está apenas se sentindo poético ou quer anotar alguns pensamentos como parte de seu programa. Isto é o quê eles são e para quê os comentários existem.

Um comentário é um pedaço de texto que é parte de um programa, mas completamente ignorado pelo computador. No JavaScript temos duas maneiras de escrever comentários. Para escrever uma linha única de comentário, você pode usar dois caracteres barra (`//`) e então o comentário após isso.

```
var accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
var report = new Report();
// Where the sun on the proud mountain rings:
addToReport(accountBalance, report);
// It's a little valley, foaming like light in a glass.
```

Um `// comentário` vai até o fim da linha. Uma seção de texto entre `/*` e `*/` vai ser ignorada, independentemente se ela contém quebras de linha. Isto é útil para adicionar blocos de informação sobre um arquivo ou um pedaço do programa.

```
/*
 I first found this number scrawled on the back of one of
 my notebooks a few years ago. Since then, it has
 occasionally dropped by, showing up in phone numbers and
 the serial numbers of products that I bought. It
 obviously likes me, so I've decided to keep it.
*/

var theNumber = 11213;
```

Resumo

Você agora sabe que um programa é construído de declarações, que as vezes contém mais declarações. Declarações tendem a conter expressões, que podem também ser feitas de pequenas expressões.

Colocar declarações uma embaixo da outra nos dá um programa que é executado de cima para baixo. Você pode introduzir distúrbios no fluxo de controle usando declarações condicionais (`if`, `else` e `swtich`) e loops (`while`, `do` e `for`).

Variáveis podem ser usadas para arquivar pedaços de dados sob um nome, e são úteis para rastrear o estado de um programa. O ambiente é o conjunto de variáveis que são definidas em um programa. O sistema JavaScript sempre coloca um número padrão de variáveis úteis dentro de seu ambiente.

Funções são valores especiais que encapsulam um pedaço de programa. Você pode invocá-las escrevendo `function Name (argument1, argument2) {}`, que é uma expressão que vai pode produzir um valor.

Exercícios

Cada exercício começa com uma descrição de um problema. Tente lê-lo e resolvê-lo. Se você dificuldades, você pode considerar ler as dicas abaixo do exercício. As soluções completas e as dicas para os exercícios não estarão inclusas neste livro, mas serão indicadas após a apresentação dos mesmos. Se você quiser aprender algo, eu recomendo que veja as soluções somente depois que você tiver resolvido o exercício, ou pelo menos tentado resolvê-lo por um longo tempo, até que o mesmo tenha lhe provocado uma pequena dor de cabeça.

Criando um Triângulo com Loop

Escreva um programa que faça 7 chamadas a `console.log()` para retornar o seguinte triângulo.

```
#
##
###
####
#####
######
#######
```

Confira uma dica de como resolver aqui[5] e a resolução aqui[6].

FizzBuzz

Escreva um programa que imprima (usando `console.log()`) todos os números de 1 a 100, exceto que, para números divisíveis por 3, ele imprima `Fizz` ao invés do número, e para números divisíveis por 5 (e não 3), ele imprima `Buzz`.

Quando você tiver o programa funcionando, modifique-o para imprimir `FizzBuzz` para números que são divisíveis por ambos os números 3 e 5.

(Isto é na verdade uma pergunta de entrevista usada para eliminar uma porcentagem significativa de candidatos programadores. Então, se você resolvê-la, você está autorizado de se sentir bem consigo mesmo).

Confira uma dica de como resolver aqui[7] e a resolução aqui[8].

Tabuleiro de Xadrez

Escreva um programa que cria uma string que representa uma grade 8x8, usando novas linhas para separar os caracteres. A cada posição da grade existe ou um espaço ou um caracter "#", de forma que estes caracteres formem um tabuleiro de xadrez.

Passando esta string para `console.log`, ela deverá se parecer com isso:

[5] https://gist.github.com/eoop/8730846
[6] https://gist.github.com/eoop/8730768
[7] https://gist.github.com/eoop/8750252
[8] https://gist.github.com/eoop/8750259

```
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
```

Quando isso funcionar, defina uma variável `tamanho = 8`, e mude o programa para que o mesmo funciona para qualquer `tamanho`, retornando uma grade com a largura e altura fornecida.

> Confira uma dica de como resolver aqui[9] e a resolução aqui[10].

[9] https://gist.github.com/eoop/8805647
[10] https://gist.github.com/eoop/8781354