Capítulo 3

Tipos, valores e variáveis

Os programas de computador funcionam manipulando *valores*, como o número 3,14 ou o texto "Olá Mundo". Os tipos de valores que podem ser representados e manipulados em uma linguagem de programação são conhecidos como *tipos* e uma das características mais fundamentais de uma linguagem de programação é o conjunto de tipos que ela aceita. Quando um programa precisa manter um valor para uso futuro, ele atribui o valor (ou "armazena" o valor) a uma *variável*. Uma variável define um nome simbólico para um valor e permite que o valor seja referido pelo nome. O funcionamento das variáveis é outra característica fundamental de qualquer linguagem de programação. Este capítulo explica os tipos, valores e variáveis em JavaScript. Os parágrafos introdutórios fornecem uma visão geral, sendo que talvez você ache útil consultar a Seção 1.1 enquanto os lê. As seções a seguir abordam esses temas em profundidade.

Os tipos de JavaScript podem ser divididos em duas categorias: *tipos primitivos* e *tipos de objeto*. Os tipos primitivos de JavaScript incluem números, sequências de texto (conhecidas como *strings*) e valores de verdade (conhecidos como *booleanos*). Uma parte significativa deste capítulo é dedicada a uma explicação detalhada dos tipos numéricos (Seção 3.1) e de string (Seção 3.2) em JavaScript. Os booleanos são abordados na Seção 3.3.

Os valores especiais `null` e `undefined` de JavaScript são valores primitivos, mas não são números, nem strings e nem booleanos. Cada valor normalmente é considerado como membro único de seu próprio tipo especial. A Seção 3.4 tem mais informações sobre `null` e `undefined`.

Qualquer valor em JavaScript que não seja número, string, booleano, `null` ou `undefined` é um objeto. Um objeto (isto é, um membro do tipo *objeto*) é um conjunto de *propriedades,* em que cada propriedade tem um nome e um valor (ou um valor primitivo, como um número ou string, ou um objeto). Um objeto muito especial, o *objeto global*, é estudado na Seção 3.5, mas uma abordagem mais geral e detalhada sobre objetos aparece no Capítulo 6.

Um objeto normal em JavaScript é um conjunto não ordenado de valores nomeados. A linguagem também define um tipo especial de objeto, conhecido como *array*, que representa um conjunto ordenado de valores numerados. A linguagem JavaScript contém sintaxe especial para trabalhar com arrays, sendo que os arrays têm um comportamento especial que os diferencia dos objetos normais. Os arrays são o tema do Capítulo 7.

JavaScript define outro tipo especial de objeto, conhecido como *função*. Uma função é um objeto que tem código executável associado. Uma função pode ser *chamada* para executar esse código executável e retornar um valor calculado. Assim como os arrays, as funções se comportam de maneira diferente dos outros tipos de objetos, sendo que JavaScript define uma sintaxe especial para trabalhar com elas. O mais importante a respeito das funções em JavaScript é que elas são valores reais e os programas em JavaScript podem tratá-las como objetos normais. As funções são abordadas no Capítulo 8.

As funções que são escritas para serem usadas (com o operador `new`) para inicializar um objeto criado recentemente são conhecidas como *construtoras*. Cada construtora define uma *classe* de objetos – o conjunto de objetos inicializados por essa construtora. As classes podem ser consideradas como subtipos do tipo de objeto. Além das classes Array e Function, JavaScript básica define outras três classes úteis. A classe Date define objetos que representam datas. A classe RegExp define objetos que representam expressões regulares (uma poderosa ferramenta de comparação de padrões, descrita no Capítulo 10). E a classe Error define objetos que representam erros de sintaxe e de execução que podem ocorrer em um programa JavaScript. Você pode estabelecer suas próprias classes de objetos, definindo funções construtoras apropriadas. Isso está explicado no Capítulo 9.

O interpretador JavaScript realiza a *coleta de lixo* automática para gerenciamento de memória. Isso significa que um programa pode criar objetos conforme for necessário e o programador nunca precisa se preocupar com a destruição ou desalocação desses objetos. Quando um objeto não pode mais ser acessado – quando um programa não tem mais maneira alguma de se referir a ele –, o interpretador sabe que ele nunca mais pode ser utilizado e recupera automaticamente o espaço de memória que ele estava ocupando.

JavaScript é uma linguagem orientada a objetos. Isso significa que, em vez de ter funções definidas globalmente para operar em valores de vários tipos, os próprios tipos definem *métodos* para trabalhar com valores. Para classificar os elementos de um array `a`, por exemplo, não passamos `a` para uma função `sort()`. Em vez disso, chamamos o método `sort()` de `a`:

```
a.sort();    // A versão orientada a objetos de sort(a).
```

A definição de método é abordada no Capítulo 9. Tecnicamente, em JavaScript apenas os objetos possuem métodos. Mas números, strings e valores booleanos se comportam como se tivessem métodos (a Seção 3.6 explica como isso funciona). Em JavaScript, `null` e `undefined` são os únicos valores em que métodos não podem ser chamados.

Os tipos de JavaScript podem ser divididos em tipos primitivos e tipos de objeto. E podem ser divididos em tipos com métodos e tipos sem métodos. Também podem ser classificados como tipos *mutáveis* e *imutáveis*. Um valor de um tipo mutável pode mudar. Objetos e arrays são mutáveis: um programa JavaScript pode alterar os valores de propriedades do objeto e de elementos de arrays. Números, booleanos, `null` e `undefined` são imutáveis – nem mesmo faria sentido falar sobre alterar o valor de um número, por exemplo. As strings podem ser consideradas arrays de caracteres, sendo que se poderia esperar que fossem mutáveis. No entanto, em JavaScript as strings são imutáveis: você pode acessar o texto de determinado índice de uma string, mas JavaScript não fornece uma maneira

de alterar o texto de uma string existente. As diferenças entre valores mutáveis e imutáveis são exploradas mais a fundo na Seção 3.7.

JavaScript converte valores de um tipo para outro de forma livre. Se um programa espera uma string, por exemplo, e você fornece um número, ele converte o número em string automaticamente. Se você usa um valor não booleano onde é esperado um booleano, JavaScript converte adequadamente. As regras de conversão de valor são explicadas na Seção 3.8. As regras de conversão de valor liberais de JavaScript afetam sua definição de igualdade, sendo que o operador de igualdade `==` realiza conversões de tipo conforme descrito na Seção 3.8.1.

As variáveis em JavaScript são *não tipadas*: você pode atribuir um valor de qualquer tipo a uma variável e, posteriormente, atribuir um valor de tipo diferente para a mesma variável. As variáveis são *declaradas* com a palavra-chave `var`. JavaScript utiliza *escopo léxico*. As variáveis declaradas fora de uma função são *variáveis globais* e são visíveis por toda parte em um programa JavaScript. As variáveis declaradas dentro de uma função têm *escopo de função* e são visíveis apenas para o código que aparece dentro dessa função. A declaração e o escopo de variáveis são abordados na Seção 3.9 e na Seção 3.10.

3.1 Números

Ao contrário de muitas linguagens, JavaScript não faz distinção entre valores inteiros e valores em ponto flutuante. Todos os números em JavaScript são representados como valores em ponto flutuante. JavaScript representa números usando o formato de ponto flutuante de 64 bits definido pelo padrão IEEE 754[1], isso significa que pode representar números tão grandes quanto $\pm 1{,}7976931348623157 \times 10^{308}$ e tão pequenos quanto $\pm 5 \times 10^{-324}$.

O formato numérico de JavaScript permite representar exatamente todos os inteiros entre -9007199254740992 (-2^{53}) e 9007199254740992 (2^{53}), inclusive. Se você usar valores inteiros maiores do que isso, poderá perder a precisão nos dígitos à direita. Note, entretanto, que certas operações em JavaScript (como indexação de arrays e os operadores bit a bit descritos no Capítulo 4) são efetuadas com inteiros de 32 bits.

Quando um número aparece diretamente em um programa JavaScript, ele é chamado de *literal numérico*. JavaScript aceita literais numéricos em vários formatos, conforme descrito nas seções a seguir. Note que qualquer literal numérico pode ser precedido por um sinal de subtração (-) para tornar o número negativo. Tecnicamente, contudo, - é o operador de negação unário (consulte o Capítulo 4) e não faz parte da sintaxe de literal numérico.

[1] Esse formato deve ser conhecido dos programadores Java como formato do tipo `double`. Também é o formato `double` usado em quase todas as implementações modernas de C e C++.

3.1.1 Literais inteiros

Em um programa JavaScript, um inteiro de base 10 é escrito como uma sequência de dígitos. Por exemplo:

```
0
3
10000000
```

Além dos literais inteiros de base 10, JavaScript reconhece valores hexadecimais (base 16). Um literal hexadecimal começa com "0x" ou "0X", seguido por uma sequência de dígitos hexadecimais. Um dígito hexadecimal é um dos algarismos de 0 a 9 ou as letras a (ou A) até f (ou F), as quais representam valores de 10 a 15. Aqui estão exemplos de literais inteiros hexadecimais:

```
0xff     // 15*16 + 15 = 255 (base 10)
0xCAFE911
```

Embora o padrão ECMAScript não ofereça suporte para isso, algumas implementações de JavaScript permitem especificar literais inteiros no formato octal (base 8). Um literal em octal começa com o dígito 0 e é seguido por uma sequência de dígitos, cada um entre 0 e 7. Por exemplo:

```
0377     // 3*64 + 7*8 + 7 = 255 (base 10)
```

Como algumas implementações aceitam literais em octal e algumas não, você nunca deve escrever um literal inteiro com um zero à esquerda; nesse caso, não dá para saber se uma implementação vai interpretá-la como um valor octal ou decimal. No modo restrito de ECMAScript 5 (Seção 5.7.3), os literais em octal são proibidos explicitamente.

3.1.2 Literais em ponto flutuante

Os literais em ponto flutuante podem ter um ponto decimal; eles usam a sintaxe tradicional dos números reais. Um valor real é representado como a parte inteira do número, seguida de um ponto decimal e a parte fracionária do número.

Os literais em ponto flutuante também podem ser representados usando-se notação exponencial: um número real seguido da letra e (ou E), seguido por um sinal de adição ou subtração opcional, seguido por um expoente inteiro. Essa notação representa o número real multiplicado por 10, elevado à potência do expoente.

Mais sucintamente, a sintaxe é:

```
[dígitos][.dígitos][(E|e)[(+|-)]dígitos]
```

Por exemplo:

```
3.14
2345.789
.333333333333333333
6.02e23          // 6.02 × 10^23
1.4738223E-32    // 1.4738223 × 10^-32
```

3.1.3 Aritmética em JavaScript

Os programas JavaScript trabalham com números usando os operadores aritméticos fornecidos pela linguagem. Isso inclui + para adição, – para subtração, * para multiplicação, / para divisão e % para módulo (resto da divisão). Mais detalhes sobre esses e outros operadores podem ser encontrados no Capítulo 4.

Além desses operadores aritméticos básicos, JavaScript aceita operações matemáticas mais complexas por meio de um conjunto de funções e constantes definidas como propriedades do objeto `Math`:

```
Math.pow(2,53)          // => 9007199254740992: 2 elevado à potência 53
Math.round(.6)          // => 1.0: arredonda para o inteiro mais próximo
Math.ceil(.6)           // => 1.0: arredonda para cima para um inteiro
Math.floor(.6)          // => 0.0: arredonda para baixo para um inteiro
Math.abs(-5)            // => 5: valor absoluto
Math.max(x,y,z)         // Retorna o maior argumento
Math.min(x,y,z)         // Retorna o menor argumento
Math.random()           // Número pseudoaleatório x, onde 0 <= x < 1.0
Math.PI                 // π: circunferência de um círculo / diâmetro
Math.E                  // e: A base do logaritmo natural
Math.sqrt(3)            // A raiz quadrada de 3
Math.pow(3, 1/3)        // A raiz cúbica de 3
Math.sin(0)             // Trigonometria: também Math.cos, Math.atan, etc.
Math.log(10)            // Logaritmo natural de 10
Math.log(100)/Math.LN10 // Logaritmo de base 10 de 100
Math.log(512)/Math.LN2  // Logaritmo de base 2 de 512
Math.exp(3)             // Math.E ao cubo
```

Consulte o objeto Math na seção de referência para ver detalhes completos sobre todas as funções matemáticas suportadas por JavaScript.

A aritmética em JavaScript não gera erros em casos de estouro, estouro negativo ou divisão por zero. Quando o resultado de uma operação numérica é maior do que o maior número representável (estouro), o resultado é um valor infinito especial, que JavaScript indica como `Infinity`. Da mesma forma, quando um valor negativo se torna maior do que o maior número negativo representável, o resultado é infinito negativo, indicado como `-Infinity`. Os valores infinitos se comportam conforme o esperado: somá-los, subtraí-los, multiplicá-los ou dividi-los por qualquer coisa resulta em um valor infinito (possivelmente com o sinal invertido).

O estouro negativo ocorre quando o resultado de uma operação numérica é mais próximo de zero do que o menor número representável. Nesse caso, JavaScript retorna 0. Se o estouro negativo ocorre a partir de um número negativo, JavaScript retorna um valor especial conhecido como "zero negativo". Esse valor é quase completamente indistinguível do zero normal e os programadores JavaScript raramente precisam detectá-lo.

Divisão por zero não é erro em JavaScript: ela simplesmente retorna infinito ou infinito negativo. Contudo, há uma exceção: zero dividido por zero não tem um valor bem definido e o resultado dessa operação é o valor especial not-a-number, impresso como `NaN`. `NaN` também surge se você tenta dividir infinito por infinito, extrai a raiz quadrada de um número negativo, ou usa operadores aritméticos com operandos não numéricos que não podem ser convertidos em números.

JavaScript predefine as variáveis globais `Infinity` e `NaN` para conter o infinito positivo e o valor not-a-number. Em ECMAScript 3, esses são valores de leitura/gravação e podem ser alterados. ECMAScript 5 corrige isso e coloca os valores no modo somente para leitura. O objeto `Number` define alternativas que são somente para leitura até em ECMAScript 3. Aqui estão alguns exemplos:

```
Infinity                    // Uma variável de leitura/gravação inicializada como
                            // Infinity.
Number.POSITIVE_INFINITY    // O mesmo valor, somente para leitura.
1/0                         // Este também é o mesmo valor.
Number.MAX_VALUE + 1        // Isso também é avaliado como Infinity.

Number.NEGATIVE_INFINITY    // Essas expressões são infinito negativo.
-Infinity
-1/0
-Number.MAX_VALUE - 1

NaN                         // Uma variável de leitura/gravação inicializada como NaN.
Number.NaN                  // Uma propriedade somente para leitura contendo o mesmo
                            // valor.
0/0                         // Avaliado como NaN.

Number.MIN_VALUE/2          // Estouro negativo: avaliado como 0
-Number.MIN_VALUE/2         // Zero negativo
-1/Infinity                 // Também 0 negativo
-0
```

O valor not-a-number tem uma característica incomum em JavaScript: não é comparado como igual a qualquer outro valor, incluindo ele mesmo. Isso significa que você não pode escrever `x == NaN` para determinar se o valor de uma variável `x` é `NaN`. Em vez disso, deve escrever `x != x`. Essa expressão será verdadeira se, e somente se, `x` for `NaN`. A função `isNaN()` é semelhante. Ela retorna `true` se seu argumento for `NaN` ou se esse argumento for um valor não numérico, como uma string ou um objeto. A função relacionada `isFinite()` retorna `true` se seu argumento for um número que não seja `NaN`, `Infinity` ou `-Infinity`.

O valor zero negativo também é um pouco incomum. Ele é comparado como igual (mesmo usando-se o teste restrito de igualdade de JavaScript) ao zero positivo, isto é, os dois valores são quase indistinguíveis, exceto quando usados como divisores:

```
var zero = 0;       // Zero normal
var negz = -0;      // Zero negativo
zero === negz       // => verdadeiro: zero e zero negativo são iguais
1/zero === 1/negz   // => falso: infinito e -infinito não são iguais
```

3.1.4 Ponto flutuante binário e erros de arredondamento

Existem infinitos números reais, mas apenas uma quantidade finita deles (18437736874454810627, para ser exato) pode ser representada de forma exata pelo formato de ponto flutuante de JavaScript. Isso significa que, quando se está trabalhando com números reais em JavaScript, a representação do número frequentemente será uma aproximação dele.

A representação em ponto flutuante IEEE-754 utilizada em JavaScript (e por praticamente todas as outras linguagens de programação modernas) é uma representação binária que pode descrever frações como `1/2`, `1/8` e `1/1024` com exatidão. Infelizmente, as frações que usamos mais comumente (especialmente ao executarmos cálculos financeiros) são decimais: `1/10`, `1/100`, etc. As representações em ponto flutuante binárias não conseguem representar números simples como `0.1` com exatidão.

Os números em JavaScript têm muita precisão e podem se aproximar bastante de `0.1`. Mas o fato de esse número não poder ser representado de forma exata pode causar problemas. Considere este código:

```
var x = .3 - .2;      // trinta centavos menos 20 centavos
var y = .2 - .1;      // vinte centavos menos 10 centavos
x == y                // => falso: os dois valores não são os mesmos!
x == .1               // => falso: .3-.2 não é igual a .1
y == .1               // => verdadeiro: .2-.1 é igual a .1
```

Devido ao erro de arredondamento, a diferença entre as aproximações de .3 e .2 não é exatamente igual à diferença entre as aproximações de .2 e .1. É importante entender que esse problema não é específico da linguagem JavaScript: ele afeta qualquer linguagem de programação que utilize números binários em ponto flutuante. Além disso, note que os valores `x` e `y` no código anterior são *muito* próximos entre si e do valor correto. Os valores calculados são adequados para quase todas as finalidades – o problema surge quando tentamos comparar a igualdade de valores.

Uma futura versão de JavaScript poderá suportar um tipo numérico decimal que evite esses problemas de arredondamento. Até então, talvez você queira efetuar cálculos financeiros importantes usando inteiros adaptados. Por exemplo, você poderia manipular valores monetários como centavos inteiros, em vez de frações de moeda.

3.1.5 Datas e horas

JavaScript básico inclui uma construtora `Date()` para criar objetos que representam datas e horas. Esses objetos Date têm métodos que fornecem uma API para cálculos simples de data. Os objetos Date não são um tipo fundamental como os números. Esta seção apresenta um estudo rápido sobre o trabalho com datas. Detalhes completos podem ser encontrados na seção de referência:

```
var then = new Date(2010, 0, 1);  // O 1º dia do 1º mês de 2010
var later = new Date(2010, 0, 1,  // O mesmo dia, às 5:10:30 da tarde, hora local
                     17, 10, 30);
var now = new Date();             // A data e hora atuais
var elapsed = now - then;         // Subtração de data: intervalo em milissegundos

later.getFullYear()               // => 2010
later.getMonth()                  // => 0: meses com base em zero
later.getDate()                   // => 1: dias com base em um
later.getDay()                    // => 5: dia da semana. 0 é domingo, 5 é sexta-feira.
later.getHours()                  // => 17: 5 da tarde, hora local
later.getUTCHours()               // Horas em UTC; depende do fuso horário
```

```
later.toString()           // => "Sexta-feira, 01 de janeiro de 2010, 17:10:30 GMT-0800
                           // (PST)"
later.toUTCString()        // => "Sábado, 02 de janeiro de 2010, 01:10:30 GMT"
later.toLocaleDateString() // => "01/01/2010"
later.toLocaleTimeString() // => "05:10:30 PM"
later.toISOString()        // => "2010-01-02T01:10:30.000Z"; somente ES5
```

3.2 Texto

Uma *string* é uma sequência ordenada imutável de valores de 16 bits, cada um dos quais normalmente representa um caractere Unicode – as strings são um tipo de JavaScript usado para representar texto. O *comprimento* de uma string é o número de valores de 16 bits que ela contém. As strings (e seus arrays) de JavaScript utilizam indexação com base em zero: o primeiro valor de 16 bits está na posição 0, o segundo na posição 1 e assim por diante. A *string vazia* é a string de comprimento 0. JavaScript não tem um tipo especial que represente um único elemento de uma string. Para representar um único valor de 16 bits, basta usar uma string que tenha comprimento 1.

Caracteres, posições de código e strings em JavaScript

JavaScript usa a codificação UTF-16 do conjunto de caracteres Unicode e as strings em JavaScript são sequências de valores de 16 bits sem sinal. Os caracteres Unicode mais comumente usados (os do "plano básico multilíngue") têm posições de código que cabem em 16 bits e podem ser representados por um único elemento de uma string. Os caracteres Unicode cujas posições de código não cabem em 16 bits são codificados de acordo com as regras da UTF-16 como uma sequência (conhecida como "par substituto") de dois valores de 16 bits. Isso significa que uma string JavaScript de comprimento 2 (dois valores de 16 bits) pode representar apenas um caractere Unicode:

```
var p = "π"; // π é 1 caractere com posição de código de 16 bits 0x03c0
var e = "𝑒"; // 𝑒 é 1 caractere com posição de código de 17 bits 0x1d452
p.length     // => 1: p consiste em 1 elemento de 16 bits
e.length     // => 2: a codificação UTF-16 de 𝑒 são 2 valores de 16 bits: "\ud835\
             // udc52"
```

Os diversos métodos de manipulação de strings definidos em JavaScript operam sobre valores de 16 bits e não sobre caracteres. Eles não tratam pares substitutos de forma especial, não fazem a normalização da string e nem mesmo garantem que uma string seja UTF-16 bem formada.

3.2.1 Strings literais

Para incluir uma string literalmente em um programa JavaScript, basta colocar os caracteres da string dentro de um par combinado de aspas simples ou duplas (' ou "). Os caracteres de aspas duplas podem estar contidos dentro de strings delimitadas por caracteres de aspas simples e estes podem estar contidos dentro de strings delimitadas por aspas duplas. Aqui estão exemplos de strings literais:

```
""  // A string vazia: ela tem zero caracteres
'testing'
"3.14"
```

```
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\nhas two lines"
"π is the ratio of a circle's circumference to its diameter"
```

Em ECMAScript 3, as strings literais devem ser escritas em uma única linha. Em ECMAScript 5, no entanto, pode-se dividir uma string literal em várias linhas, finalizando cada uma delas, menos a última, com uma barra invertida (\). Nem a barra invertida nem a terminação de linha que vem depois dela fazem parte da string literal. Se precisar incluir um caractere de nova linha em uma string literal, use a sequência de caracteres `\n` (documentada a seguir):

```
"two\nlines" // Uma string representando 2 linhas escritas em uma linha
"one\        // Uma string de uma linha escrita em 3 linhas. Somente ECMAScript 5.
long\
line"
```

Note que, ao usar aspas simples para delimitar suas strings, você deve tomar cuidado com as contrações e os possessivos do idioma inglês, como *can't* e *O'Reilly's*. Como o apóstrofo é igual ao caractere de aspas simples, deve-se usar o caractere de barra invertida (\) para fazer o "escape" de qualquer apóstrofo que apareça em strings com aspas simples (os escapes estão explicados na próxima seção).

Na programação JavaScript do lado do cliente, o código JavaScript pode conter strings de código HTML e o código HTML pode conter strings de código JavaScript. Assim como JavaScript, HTML utiliza aspas simples ou duplas para delimitar suas strings. Assim, ao se combinar JavaScript e HTML, é uma boa ideia usar um estilo de aspas para JavaScript e outro para HTML. No exemplo a seguir, a string "Thank you" está entre aspas simples dentro de uma expressão JavaScript, a qual é colocada entre aspas duplas dentro de um atributo de rotina de tratamento de evento em HTML:

```
<button onclick="alert('Thank you')">Click Me</button>
```

3.2.2 Sequências de escape em strings literais

O caractere de barra invertida (\) tem um propósito especial nas strings em JavaScript. Combinado com o caractere que vem a seguir, ele representa um caractere que não pode ser representado de outra forma dentro da string. Por exemplo, `\n` é uma *sequência de escape* que representa um caractere de nova linha.

Outro exemplo, mencionado anteriormente, é o escape \', que representa o caractere de aspas simples (ou apóstrofo). Essa sequência de escape é útil quando se precisa incluir um apóstrofo em uma string literal que está contida dentro de aspas simples. Você pode ver por que elas são chamadas de sequências de escape: a barra invertida permite escapar da interpretação normal do caractere de aspas simples. Em vez de utilizá-lo para marcar o final da string, você o utiliza como um apóstrofo:

```
'You\'re right, it can\'t be a quote'
```

A Tabela 3-1 lista as sequências de escape em JavaScript e os caracteres que representam. Duas sequências de escape são genéricas e podem ser usadas para representar qualquer caractere, especificando-se seu código de caractere Latin-1 ou Unicode como um número hexadecimal. Por exemplo, a sequência `\xA9` representa o símbolo de direitos autorais, o qual tem a codificação Latin-1 dada pelo número hexadecimal A9. Da mesma forma, o escape `\u` representa um caractere Unicode arbitrário especificado por quatro dígitos hexadecimais; `\u03c0` representa o caractere π, por exemplo.

Tabela 3-1 Sequências de escape em JavaScript

Sequência	Caractere representado
\0	O caractere NUL (\u0000)
\b	Retrocesso (\u0008)
\t	Tabulação horizontal (\u0009)
\n	Nova linha (\u000A)
\v	Tabulação vertical (\u000B)
\f	Avanço de página (\u000C)
\r	Retorno de carro (\u000D)
\"	Aspas duplas (\u0022)
\'	Apóstrofo ou aspas simples (\u0027)
\\	Barra invertida (\u005C)
\x *XX*	O caractere Latin-1 especificado pelos dois dígitos hexadecimais *XX*
\u *XXXX*	O caractere Unicode especificado pelos quatro dígitos hexadecimais *XXXX*

Se o caractere \ precede qualquer outro caractere que não seja um dos mostrados na Tabela 3-1, a barra invertida é simplesmente ignorada (embora, é claro, versões futuras da linguagem possam definir novas sequências de escape). Por exemplo, \# é o mesmo que #. Por fim, conforme observado anteriormente, a ECMAScript 5 permite que uma barra invertida antes de uma quebra de linha divida uma string literal em várias linhas.

3.2.3 Trabalhando com strings

Um dos recursos incorporados a JavaScript é a capacidade de *concatenar* strings. Se o operador + é utilizado com números, ele os soma. Mas se esse operador é usado em strings, ele as une, anexando a segunda na primeira. Por exemplo:

```
msg = "Hello, " + "world";      // Produz a string "Hello, world"
greeting = "Welcome to my blog," + " " + name;
```

Para determinar o comprimento de uma string – o número de valores de 16 bits que ela contém – use sua propriedade `length`. Determine o comprimento de uma string `s` como segue:

```
s.length
```

Além dessa propriedade `length`, existem vários métodos que podem ser chamados em strings (como sempre, consulte a seção de referência para ver detalhes completos):

```
var s = "hello, world"          // Começa com um texto.
s.charAt(0)                     // => "h": o primeiro caractere.
s.charAt(s.length-1)            // => "d": o último caractere.
s.substring(1,4)                // => "ell": o 2º, 3º e 4º caracteres.
s.slice(1,4)                    // => "ell": a mesma coisa
s.slice(-3)                     // => "rld": os últimos 3 caracteres
s.indexOf("l")                  // => 2: posição da primeira letra l.
s.lastIndexOf("l")              // => 10: posição da última letra l.
s.indexOf("l", 3)               // => 3: posição do primeiro "l" em ou após 3
```

```
s.split(", ")                  // => ["hello", "world"] divide em substrings
s.replace("h", "H")            // => "Hello, world": substitui todas as instâncias
s.toUpperCase()                // => "HELLO, WORLD"
```

Lembre-se de que as strings são imutáveis em JavaScript. Métodos como `replace()` e `toUpperCase()` retornam novas strings – eles não modificam a string em que são chamados.

Em ECMAScript 5, as strings podem ser tratadas como arrays somente para leitura e é possível acessar caracteres individuais (valores de 16 bits) de uma string usando colchetes em lugar do método `charAt()`:

```
s = "hello, world";
s[0]                           // => "h"
s[s.length-1]                  // => "d"
```

Os navegadores Web baseados no Mozilla, como o Firefox, permitem que as strings sejam indexadas dessa maneira há muito tempo. A maioria dos navegadores modernos (com a notável exceção do IE) seguiu o exemplo do Mozilla mesmo antes que esse recurso fosse padronizado em ECMAScript 5.

3.2.4 Comparação de padrões

JavaScript define uma construtora `RegExp()` para criar objetos que representam padrões textuais. Esses padrões são descritos com *expressões regulares,* sendo que JavaScript adota a sintaxe da Perl para expressões regulares. Tanto as strings como os objetos RegExp têm métodos para fazer comparação de padrões e executar operações de busca e troca usando expressões regulares.

Os objetos RegExp não são um dos tipos fundamentais de JavaScript. Assim como os objetos Date, eles são simplesmente um tipo de objeto especializado, com uma API útil. A gramática da expressão regular é complexa e a API não é trivial. Elas estão documentadas em detalhes no Capítulo 10. No entanto, como os objetos RegExp são poderosos e utilizados comumente para processamento de texto, esta seção fornece uma breve visão geral.

Embora os objetos RegExp não sejam um dos tipos de dados fundamentais da linguagem, eles têm uma sintaxe literal e podem ser codificados diretamente nos programas JavaScript. O texto entre um par de barras normais constitui uma expressão regular literal. A segunda barra normal do par também pode ser seguida por uma ou mais letras, as quais modificam o significado do padrão. Por exemplo:

```
/^HTML/            // Corresponde às letras H T M L no início de uma string
/[1-9][0-9]*/      // Corresponde a um dígito diferente de zero, seguido de qualquer
                   // número de dígitos
/\bjavascript\b/i  // Corresponde a "javascript" como uma palavra, sem considerar letras
                   // maiúsculas e minúsculas
```

Os objetos RegExp definem vários métodos úteis e as strings também têm métodos que aceitam argumentos de RegExp. Por exemplo:

```
var text = "testing: 1, 2, 3";  // Exemplo de texto
var pattern = /\d+/g            // Corresponde a todas as instâncias de um ou mais
                                // dígitos
pattern.test(text)              // => verdadeiro: existe uma correspondência
text.search(pattern)            // => 9: posição da primeira correspondência
text.match(pattern)             // => ["1", "2", "3"]: array de todas as correspondências
text.replace(pattern, "#");     // => "testing: #, #, #"
text.split(/\D+/);              // => ["","1","2","3"]: divide em não dígitos
```

3.3 Valores booleanos

Um valor booleano representa verdadeiro ou falso, ligado ou desligado, sim ou não. Só existem dois valores possíveis desse tipo. As palavras reservadas `true` e `false` são avaliadas nesses dois valores.

Geralmente, os valores booleanos resultam de comparações feitas nos programas JavaScript. Por exemplo:

```
a == 4
```

Esse código faz um teste para ver se o valor da variável `a` é igual ao número `4`. Se for, o resultado dessa comparação é o valor booleano `true`. Se `a` não é igual a `4`, o resultado da comparação é `false`.

Os valores booleanos são comumente usados em estruturas de controle em JavaScript. Por exemplo, a instrução `if/else` de JavaScript executa uma ação se um valor booleano é `true` e outra ação se o valor é `false`. Normalmente, uma comparação que gera um valor booleano é combinada diretamente com a instrução que o utiliza. O resultado é o seguinte:

```
if (a == 4)
  b = b + 1;
else
  a = a + 1;
```

Esse código verifica se `a` é igual a `4`. Se for, ele soma `1` a `b`; caso contrário, ele soma `1` a `a`. Conforme discutiremos na Seção 3.8, em JavaScript qualquer valor pode ser convertido em um valor booleano. Os valores a seguir são convertidos (e, portanto, funcionam como) em `false`:

```
undefined
null
0
-0
NaN
""   // a string vazia
```

Todos os outros valores, incluindo todos os objetos (e arrays) são convertidos (e funcionam como) em `true`. `false` e os seis valores assim convertidos, às vezes são chamados de valores *falsos* e todos os outros valores são chamados de `verdadeiros`. Sempre que JavaScript espera um valor booleano, um valor falso funciona como `false` e um valor verdadeiro funciona como `true`.

Como exemplo, suponha que a variável `o` contém um objeto ou o valor `null`. Você pode testar explicitamente para ver se `o` é não nulo, com uma instrução `if`, como segue:

```
if (o !== null) ...
```

O operador de desigualdade `!==` compara `o` com `null` e é avaliado como `true` ou como `false`. Mas você pode omitir a comparação e, em vez disso, contar com o fato de que `null` é falso e os objetos são verdadeiros:

```
if (o) ...
```

No primeiro caso, o corpo da instrução `if` só vai ser executado se `o` não for `null`. O segundo caso é menos rigoroso: ele executa o corpo da instrução `if` somente se o não é `false` ou qualquer valor falso (como `null` ou `undefined`). A instrução `if` apropriada para seu programa depende de quais valores você espera atribuir para `o`. Se você precisa diferenciar `null` de `0` e `""`, então deve utilizar uma comparação explícita.

Os valores booleanos têm um método `toString()` que pode ser usado para convertê-los nas strings "true" ou "false", mas não possuem qualquer outro método útil. Apesar da API trivial, existem três operadores booleanos importantes.

O operador `&&` executa a operação booleana E. Ele é avaliado como um valor verdadeiro se, e somente se, seus dois operandos são verdadeiros; caso contrário, é avaliado como um valor falso. O operador `||` é a operação booleana OU: ele é avaliado como um valor verdadeiro se um ou outro (ou ambos) de seus operandos é verdadeiro e é avaliado como um valor falso se os dois operandos são falsos. Por fim, o operador unário `!` executa a operação booleana NÃO: ele é avaliado como `true` se seu operando é falso e é avaliado como `false` se seu operando é verdadeiro. Por exemplo:

```
if ((x == 0 && y == 0) || !(z == 0)) {
    // x e y são ambos zero ou z não é zero
}
```

Os detalhes completos sobre esses operadores estão na Seção 4.10.

3.4 null e undefined

`null` é uma palavra-chave da linguagem avaliada com um valor especial, normalmente utilizado para indicar a ausência de um valor. Usar o operador `typeof` em `null` retorna a string "object", indicando que `null` pode ser considerado um valor de objeto especial que significa "nenhum objeto". Na prática, contudo, `null` normalmente é considerado como o único membro de seu próprio tipo e pode ser usado para indicar "nenhum valor" para números e strings, assim como para objetos. A maioria das linguagens de programação tem um equivalente para o `null` de JavaScript: talvez você já o conheça como `null` ou `nil`.

JavaScript também tem um segundo valor que indica ausência de valor. O valor indefinido representa uma ausência mais profunda. É o valor de variáveis que não foram inicializadas e o valor obtido quando se consulta o valor de uma propriedade de objeto ou elemento de array que não existe. O valor indefinido também é retornado por funções que não têm valor de retorno e o valor de parâmetros de função quando os quais nenhum argumento é fornecido. `undefined` é uma variável global predefinida (e não uma palavra-chave da linguagem, como `null`) que é inicializada com o valor indefinido. Em ECMAScript 3, `undefined` é uma variável de leitura/gravação e pode ser configurada com qualquer valor. Esse erro foi corrigido em ECMAScript 5 e `undefined` é somente para leitura nessa versão da linguagem. Se você aplicar o operador `typeof` no valor indefinido, ele vai retornar "undefined", indicando que esse valor é o único membro de um tipo especial.

Apesar dessas diferenças, tanto `null` quanto `undefined` indicam uma ausência de valor e muitas vezes podem ser usados indistintamente. O operador de igualdade `==` os considera iguais. (Para diferenciá-los, use o operador de igualdade restrito `===`.) Ambos são valores falsos – eles se comportam como `false` quando um valor booleano é exigido. Nem `null` nem `undefined` tem propriedades ou métodos. Na verdade, usar `.` ou `[]` para acessar uma propriedade ou um método desses valores causa um TypeError.

Você pode pensar em usar `undefined` para representar uma ausência de valor em nível de sistema, inesperada ou como um erro e `null` para representar ausência de valor em nível de programa, normal ou esperada. Se precisar atribuir um desses valores a uma variável ou propriedade ou passar um desses valores para uma função, `null` quase sempre é a escolha certa.

3.5 O objeto global

As seções anteriores explicaram os tipos primitivos e valores em JavaScript. Os tipos de objeto – objetos, arrays e funções – são abordados em seus próprios capítulos, posteriormente neste livro. Porém, existe um valor de objeto muito importante que precisamos abordar agora. O *objeto global* é um objeto normal de JavaScript que tem um objetivo muito importante: as propriedades desse objeto são os símbolos definidos globalmente que estão disponíveis para um programa JavaScript. Quando o interpretador JavaScript começa (ou quando um navegador Web carrega uma nova página), ele cria um novo objeto global e dá a ele um conjunto inicial de propriedades que define:

- propriedades globais, como `undefined`, `Infinity` e `NaN`
- funções globais, como `isNaN()`, `parseInt()` (Seção 3.8.2) e `eval()` (Seção 4.12).
- funções construtoras, como `Date()`, `RegExp()`, `String()`, `Object()` e `Array()` (Seção 3.8.2)
- objetos globais, como Math e JSON (Seção 6.9)

As propriedades iniciais do objeto global não são palavras reservadas, mas merecem ser tratadas como se fossem. A Seção 2.4.1 lista cada uma dessas propriedades. Este capítulo já descreveu algumas dessas propriedades globais. A maioria das outras será abordada em outras partes deste livro. E você pode procurá-las pelo nome na seção de referência de JavaScript básico ou procurar o próprio objeto global sob o nome "Global". Em JavaScript do lado do cliente, o objeto Window define outros globais que podem ser pesquisados na seção de referência do lado do cliente.

No código de nível superior – código JavaScript que não faz parte de uma função –, pode-se usar a palavra-chave `this` de JavaScript para se referir ao objeto global:

```
var global = this;    // Define uma variável global para se referir ao objeto global
```

Em JavaScript do lado do cliente, o objeto Window serve como objeto global para todo código JavaScript contido na janela do navegador que ele representa. Esse objeto global Window tem uma propriedade de autoreferência `window` que pode ser usada no lugar de `this` para se referir ao objeto global. O objeto Window define as propriedades globais básicas, mas também define muitos outros globais que são específicos para navegadores Web e para JavaScript do lado do cliente.

Ao ser criado, o objeto global define todos os valores globais predefinidos de JavaScript. Mas esse objeto especial também contém globais definidos pelo programa. Se seu código declara uma variável global, essa variável é uma propriedade do objeto global. A Seção 3.10.2 explica isso com mais detalhes.

3.6 Objetos wrapper

Os objetos JavaScript são valores compostos: eles são um conjunto de propriedades ou valores nomeados. Ao usarmos a notação `.` fazemos referência ao valor de uma propriedade. Quando o valor de uma propriedade é uma função, a chamamos de *método*. Para chamar o método `m` de um objeto `o`, escrevemos `o.m()`.

Também vimos que as strings têm propriedades e métodos:

```
var s = "hello world!";                                  // Uma string
var word = s.substring(s.indexOf(" ")+1, s.length);      // Usa propriedades da string
```

Contudo, as strings não são objetos. Então, por que elas têm propriedades? Quando você tenta se referir a uma propriedade de uma string `s`, JavaScript converte o valor da string em um objeto como se estivesse chamando `new String(s)`. Esse objeto herda (consulte a Seção 6.2.2) métodos da string e é utilizado para solucionar a referência da propriedade. Uma vez solucionada a propriedade, o objeto recentemente criado é descartado. (As implementações não são obrigadas a criar e descartar esse objeto transitório – contudo, devem se comportar como se fossem.)

Números e valores booleanos têm métodos pelo mesmo motivo que as strings: um objeto temporário é criado com a construtora `Number()` ou `Boolean()` e o método é solucionado por meio desse objeto temporário. Não existem objetos empacotadores (wrapper) para os valores `null` e `undefined`: qualquer tentativa de acessar uma propriedade de um desses valores causa um `TypeError`.

Considere o código a seguir e pense no que acontece quando ele é executado:

```
var s = "test";       // Começa com um valor de string.
s.len = 4;            // Configura uma propriedade nele.
var t = s.len;        // Agora consulta a propriedade.
```

Quando esse código é executado, o valor de `t` é `undefined`. A segunda linha de código cria um objeto String temporário, configura sua propriedade `len` como 4 e, em seguida, descarta esse objeto. A terceira linha cria um novo objeto String a partir do valor da string original (não modificado) e, então, tenta ler a propriedade `len`. Essa propriedade não existe e a expressão é avaliada como `undefined`. Esse código demonstra que strings, números e valores booleanos se comportam como objetos quando se tenta ler o valor de uma propriedade (ou método) deles. Mas se você tenta definir o valor de uma propriedade, essa tentativa é ignorada silenciosamente: a alteração é feita em um objeto temporário e não persiste.

Os objetos temporários criados ao se acessar uma propriedade de uma string, número ou valor booleano são conhecidos como *objetos empacotadores (wrapper)* e ocasionalmente pode ser necessário diferenciar um valor de string de um objeto String ou um número ou valor booleano de um objeto Number ou Boolean. Normalmente, contudo, os objetos wrapper podem ser considerados como

um detalhe de implementação e não é necessário pensar neles. Basta saber que string, número e valores booleanos diferem de objetos pois suas propriedades são somente para leitura e que não é possível definir novas propriedades neles.

Note que é possível (mas quase nunca necessário ou útil) criar objetos wrapper explicitamente, chamando as construtoras `String()`, `Number()` ou `Boolean()`:

```
var s = "test", n = 1, b = true;     // Uma string, um número e um valor booleano.
var S = new String(s);               // Um objeto String
var N = new Number(n);               // Um objeto Number
var B = new Boolean(b);              // Um objeto Boolean
```

JavaScript converte objetos wrapper no valor primitivo empacotado, quando necessário; portanto, os objetos `S`, `N` e `B` anteriores normalmente (mas nem sempre) se comportam exatamente como os valores `s`, `n` e `b`. O operador de igualdade `==` trata um valor e seu objeto wrapper como iguais, mas é possível diferenciá-los com o operador de igualdade restrito `===`. O operador `typeof` também mostra a diferença entre um valor primitivo e seu objeto wrapper.

3.7 Valores primitivos imutáveis e referências de objeto mutáveis

Em JavaScript existe uma diferença fundamental entre valores primitivos (`undefined`, `null`, booleanos, números e strings) e objetos (incluindo arrays e funções). Os valores primitivos são imutáveis: não há como alterar (ou "mudar") um valor primitivo. Isso é óbvio para números e booleanos – nem mesmo faz sentido mudar o valor de um número. No entanto, não é tão óbvio para strings. Como as strings são como arrays de caracteres, você poderia pensar que é possível alterar o caractere em qualquer índice especificado. Na verdade, JavaScript não permite isso e todos os métodos de string que parecem retornar uma string modificada estão na verdade retornando um novo valor de string. Por exemplo:

```
var s = "hello";      // Começa com um texto em letras minúsculas
s.toUpperCase();      // Retorna "HELLO", mas não altera s
s                     // => "hello": a string original não mudou
```

Os valores primitivos também são comparados *por valor*: dois valores são iguais somente se têm o mesmo valor. Isso parece recorrente para números, booleanos, `null` e `undefined`: não há outra maneira de compará-los. Novamente, contudo, não é tão óbvio para strings. Se dois valores distintos de string são comparados, JavaScript os trata como iguais se, e somente se, tiverem o mesmo comprimento e se o caractere em cada índice for o mesmo.

Os objetos são diferentes dos valores primitivos. Primeiramente, eles são *mutáveis* – seus valores podem mudar:

```
var o = { x:1 };      // Começa com um objeto
o.x = 2;              // Muda-o, alterando o valor de uma propriedade
o.y = 3;              // Muda-o novamente, adicionando uma nova propriedade

var a = [1,2,3]       // Os arrays também são mutáveis
a[0] = 0;             // Muda o valor de um elemento do array
a[3] = 4;             // Adiciona um novo elemento no array
```

Objetos não são comparados por valor: dois objetos não são iguais mesmo que tenham as mesmas propriedades e valores. E dois arrays não são iguais mesmo que tenham os mesmos elementos na mesma ordem:

```
var o = {x:1}, p = {x:1};       // Dois objetos com as mesmas propriedades
o === p                         // => falso: objetos distintos nunca são iguais
var a = [], b = [];             // Dois arrays vazios diferentes
a === b                         // => falso: arrays diferentes nunca são iguais
```

Às vezes os objetos são chamados de *tipos de referência* para distingui-los dos tipos primitivos de JavaScript. Usando essa terminologia, os valores de objeto são *referências* e dizemos que os objetos são comparados *por referência*: dois valores de objeto são iguais se, e somente se, eles se *referem* ao mesmo objeto básico.

```
var a = [];  // A variável a se refere a um array vazio.
var b = a;   // Agora b se refere ao mesmo array.
b[0] = 1;    // Muda o array referido pela variável b.
a[0]         // => 1: a mudança também é visível por meio da variável a.
a === b      // => verdadeiro: a e b se referem ao mesmo objeto; portanto, são iguais.
```

Como você pode ver no código anterior, atribuir um objeto (ou array) a uma variável simplesmente atribui a referência: isso não cria uma nova cópia do objeto. Se quiser fazer uma nova cópia de um objeto ou array, você precisa copiar explicitamente as propriedades do objeto ou dos elementos do array. Este exemplo demonstra o uso de um laço `for` (Seção 5.5.3):

```
var a = ['a','b','c'];              // Um array que queremos copiar
var b = [];                         // Um array diferente no qual vamos copiar
for(var i = 0; i < a.length; i++) {  // Para cada índice de []
    b[i] = a[i];                    // Copia um elemento de a em b
}
```

Da mesma forma, se queremos comparar dois objetos ou arrays distintos, devemos comparar suas propriedades ou seus elementos. Este código define uma função para comparar dois arrays:

```
function equalArrays(a,b) {
    if (a.length != b.length) return false;     // Arrays de tamanho diferente não são
                                                // iguais
    for(var i = 0; i < a.length; i++)           // Itera por todos os elementos
    if (a[i] !== b[i]) return false;            // Se algum difere, os arrays não são
                                                // iguais
    return true;                                // Caso contrário, eles são iguais
}
```

3.8 Conversões de tipo

A JavaScript é muito flexível quanto aos tipos de valores que exige. Vimos isso no caso dos booleanos: quando a JavaScript espera um valor booleano, você pode fornecer um valor de qualquer tipo – ela o converte conforme for necessário. Alguns valores (valores "verdadeiros") são convertidos em `true` e outros (valores "falsos") são convertidos em `false`. O mesmo vale para outros tipos: se a JavaScript quer uma string, ela converte qualquer valor fornecido em uma string. Se a JavaScript quer um número, ela tenta converter o valor fornecido para um número (ou para `NaN`, caso não consiga fazer uma conversão significativa). Alguns exemplos:

```
10 + " objects"       // => "10 objects". O número 10 é convertido em uma string
"7" * "4"             // => 28: as duas strings são convertidas em números
```

```
var n = 1 - "x";      // => NaN: a string "x" não pode ser convertida em um número
n + " objects "       // => "NaN objects": NaN é convertido na string "NaN"
```

A Tabela 3-2 resume como os valores são convertidos de um tipo para outro em JavaScript. As entradas em negrito na tabela destacam as conversões que talvez você ache surpreendentes. As células vazias indicam que nenhuma conversão é necessária e nada é feito.

Tabela 3-2 Conversões de tipo da JavaScript

Valor	Convertido em:			
	String	**Número**	**Booleano**	**Objeto**
`undefined`	`"undefined"`	`NaN`	`false`	*lança TypeError*
`null`	`"null"`	**`0`**	`false`	*lança TypeError*
`true`	`"true"`	**`1`**		`new Boolean(true)`
`false`	`"false"`	**`0`**		`new Boolean(false)`
`""` (string vazia)		**`0`**	**`false`**	`new String("")`
`"1.2"` (não vazio, numérico)		`1.2`	`true`	`new String("1.2")`
`"one"` (não vazio, não numérico)		`NaN`	`true`	`new String("one")`
`0`	`"0"`		**`false`**	`new Number(0)`
`-0`	`"0"`		**`false`**	`new Number(-0)`
`NaN`	`"NaN"`		**`false`**	`new Number(NaN)`
`Infinity`	`"Infinity"`		`true`	`new Number(Infinity)`
`-Infinity`	`"-Infinity"`		`true`	`new Number(-Infinity)`
`1` (finito, não zero)	`"1"`		`true`	`new Number(1)`
`{}` (qualquer objeto)	*consulte a Seção 3.8.3*	*consulte a Seção 3.8.3*	`true`	
`[]` (array vazio)	`""`	**`0`**	`true`	
`[9]` (1 elt numérico)	`"9"`	**`9`**	`true`	
`['a']` (qualquer outro array)	*use o método join()*	`NaN`	`true`	
`function(){}` (qualquer função)	*consulte a Seção 3.8.3*	`NaN`	`true`	

As conversões de valor primitivo para valor primitivo mostradas na tabela são relativamente simples. A conversão para booleano já foi discutida na Seção 3.3. A conversão para strings é bem definida para todos os valores primitivos. A conversão para números é apenas um pouco mais complicada. As strings que podem ser analisadas como números são convertidas nesses números. Espaços antes e depois são permitidos, mas qualquer caractere que não seja espaço antes ou depois e que não faça parte de um literal numérico faz a conversão de string para número produzir `NaN`. Algumas conver-

sões numéricas podem parecer surpreendentes: `true` é convertido em 1 e `false` e a string vazia `""` são convertidos em `0`.

As conversões de valor primitivo para objeto são diretas: os valores primitivos são convertidos em seus objetos wrapper (Seção 3.6), como se estivessem chamando a construtora `String()`, `Number()` ou `Boolean()`.

As exceções são `null` e `undefined`: qualquer tentativa de usar esses valores onde é esperado um objeto dispara um TypeError, em vez de realizar uma conversão.

As conversões de objeto para valor primitivo são um pouco mais complicadas e são o tema da Seção 3.8.3.

3.8.1 Conversões e igualdade

Como JavaScript pode converter valores com flexibilidade, seu operador de igualdade `==` também é flexível em sua noção de igualdade. Todas as comparações a seguir são verdadeiras, por exemplo:

```
null == undefined     // Esses dois valores são tratados como iguais.
"0" == 0              // A string é convertida em um número antes da comparação.
0 == false            // O booleano é convertido em número antes da comparação.
"0" == false          // Os dois operandos são convertidos em números antes da
                      // comparação.
```

A Seção 4.9.1 explica exatamente quais conversões são realizadas pelo operador `==` para determinar se dois valores devem ser considerados iguais e também descreve o operador de igualdade restrito `===`, que não faz conversões ao testar a igualdade.

Lembre-se de que a capacidade de conversão de um valor para outro não implica na igualdade desses dois valores. Se `undefined` for usado onde é esperado um valor booleano, por exemplo, ele será convertido em `false`. Mas isso não significa que `undefined == false`. Os operadores e as instruções em JavaScript esperam valores de vários tipos e fazem as conversões para esses tipos. A instrução `if` converte `undefined` em `false`, mas o operador `==` nunca tenta converter seus operandos para booleanos.

3.8.2 Conversões explícitas

Embora JavaScript faça muitas conversões de tipo automaticamente, às vezes será necessário realizar uma conversão explícita ou talvez você prefira usar as conversões de forma explícita para manter o código mais claro.

O modo mais simples de fazer uma conversão de tipo explícita é usar as funções `Boolean()`, `Number()`, `String()` ou `Object()`. Já vimos essas funções como construtoras para objetos wrapper (na Seção 3.6). Contudo, quando chamadas sem o operador `new`, elas funcionam como funções de conversão e fazem as conversões resumidas na Tabela 3-2:

```
Number("3")           // => 3
String(false)         // => "false" Ou use false.toString()
Boolean([])           // => verdadeiro
Object(3)             // => novo Number(3)
```

Note que qualquer valor que não seja `null` ou `undefined` tem um método `toString()` e o resultado desse método normalmente é igual ao retornado pela função `String()`. Note também que a Tabela 3-2 mostra um TypeError se você tenta converter `null` ou `undefined` em um objeto. A função `Object()` não levanta uma exceção nesse caso: em vez disso, ela simplesmente retorna um objeto vazio recentemente criado.

Certos operadores de JavaScript fazem conversões de tipo implícitas e às vezes são usados para propósitos de conversão de tipo. Se um operando do operador `+` é uma string, ele converte o outro em uma string. O operador unário `+` converte seu operando em um número. E o operador unário `!` converte seu operando em um valor booleano e o nega. Esses fatos levam aos seguintes idiomas de conversão de tipo que podem ser vistos em algum código:

```
x + ""       // O mesmo que String(x)
+x           // O mesmo que Number(x). Você também poderá ver x-0
!!x          // O mesmo que Boolean(x). Observe o duplo !
```

Formatar e analisar números são tarefas comuns em programas de computador e JavaScript tem funções e métodos especializados que oferecem controle mais preciso sobre conversões de número para string e de string para número.

O método `toString()` definido pela classe Number aceita um argumento opcional que especifica uma raiz (ou base) para a conversão. Se o argumento não é especificado, a conversão é feita na base 10. Contudo, também é possível converter números em outras bases (entre 2 e 36). Por exemplo:

```
var n = 17;
binary_string = n.toString(2);       // É avaliado como"10001"
octal_string = "0" + n.toString(8);  // É avaliado como"021"
hex_string = "0x" + n.toString(16);  // É avaliado como"0x11"
```

Ao trabalhar com dados financeiros ou científicos, talvez você queira converter números em strings de maneiras que ofereçam controle sobre o número de casas decimais ou sobre o número de dígitos significativos na saída; ou então, talvez queira controlar o uso de notação exponencial. A classe Number define três métodos para esses tipos de conversões de número para string. `toFixed()` converte um número em uma string com um número especificado de dígitos após a casa decimal. Ele nunca usa notação exponencial. `toExponential()` converte um número em uma string usando notação exponencial, com um dígito antes da casa decimal e um número especificado de dígitos após a casa decimal (ou seja, o número de dígitos significativos é um a mais do que o valor especificado). `toPrecision()` converte um número em uma string com o número de dígitos significativos especificado. Ela usa notação exponencial se o número de dígitos significativos não for grande o suficiente para exibir toda a parte inteira do número. Note que todos os três métodos arredondam os dígitos à direita ou preenchem com zeros, conforme for apropriado. Considere os exemplos a seguir:

```
var n = 123456.789;
n.toFixed(0);         // "123457"
n.toFixed(2);         // "123456.79"
n.toFixed(5);         // "123456.78900"
n.toExponential(1);   // "1.2e+5"
n.toExponential(3);   // "1.235e+5"
n.toPrecision(4);     // "1.235e+5"
n.toPrecision(7);     // "123456.8"
n.toPrecision(10);    // "123456.7890"
```

Se uma string é passada para a função de conversão `Number()`, ela tenta analisar essa string como um inteiro ou literal em ponto flutuante. Essa função só trabalha com inteiros de base 10 e não permite caracteres à direita que não façam parte da literal. As funções `parseInt()` e `parseFloat()` (essas são funções globais e não métodos de qualquer classe) são mais flexíveis. `parseInt()` analisa somente inteiros, enquanto `parseFloat()` analisa inteiros e números em ponto flutuante. Se uma string começa com "0x" ou "0X", `parseInt()` a interpreta como um número hexadecimal[2]. Tanto `parseInt()` como `parseFloat()` pulam espaços em branco à esquerda, analisam o máximo de caracteres numéricos que podem e ignoram tudo que vem em seguida. Se o primeiro caractere que não é espaço não faz parte de uma literal numérica válida, elas retornam NaN:

```
parseInt("3 blind mice")        // => 3
parseFloat(" 3.14 meters")      // => 3.14
parseInt("-12.34")              // => -12
parseInt("0xFF")                // => 255
parseInt("0xff")                // => 255
parseInt("-0XFF")               // => -255
parseFloat(".1")                // => 0.1
parseInt("0.1")                 // => 0
parseInt(".1")                  // => NaN: inteiros não podem começar com "."
parseFloat("$72.47");           // => NaN: números não podem começar com "$"
```

`parseInt()` aceita um segundo argumento opcional especificando a raiz (base) do número a ser analisado. Os valores válidos estão entre 2 e 36. Por exemplo:

```
parseInt("11", 2);              // => 3 (1*2 + 1)
parseInt("ff", 16);             // => 255 (15*16 + 15)
parseInt("zz", 36);             // => 1295 (35*36 + 35)
parseInt("077", 8);             // => 63 (7*8 + 7)
parseInt("077", 10);            // => 77 (7*10 + 7)
```

3.8.3 Conversões de objeto para valores primitivos

As conversões de objeto para valores booleanos são simples: todos os objetos (inclusive arrays e funções) são convertidos em `true`. Isso vale até para objetos wrapper: `new Boolean(false)` é um objeto e não um valor primitivo e também é convertido em `true`.

As conversões de objeto para string e de objeto para número são feitas chamando-se um método do objeto a ser convertido. Isso é complicado pelo fato de que os objetos em JavaScript têm dois métodos diferentes que realizam conversões e também é complicado por alguns casos especiais descritos a seguir. Note que as regras de conversão de strings e números descritas aqui se aplicam apenas a objetos nativos. Os objetos hospedeiros (definidos pelos navegadores Web, por exemplo) podem ser convertidos em números e strings de acordo com seus próprios algoritmos.

[2] Em ECMAScript 3, `parseInt()` pode analisar uma string que começa com "0" (mas não com "0x" ou "0X") como um número octal ou como um número decimal. Como o comportamento não é especificado, você nunca deve usar `parseInt()` para analisar números com zeros à esquerda, a não ser que especifique explicitamente a raiz a ser usada! Em ECMAScript 5, `parseInt()` só analisa números octais se você passa 8 como segundo argumento explicitamente.

Todos os objetos herdam dois métodos de conversão. O primeiro é chamado `toString()` e sua tarefa é retornar uma representação de string do objeto. O método padrão `toString()` não retorna um valor muito interessante (embora o achemos útil no Exemplo 6-4):

```
({x:1, y:2}).toString()             // => "[object Object]"
```

Muitas classes definem versões mais específicas do método `toString()`. O método `toString()` da classe Array, por exemplo, converte cada elemento do array em uma string e une as strings resultantes com vírgulas entre elas. O método `toString()` da classe Function retorna uma representação definida pela implementação de uma função. Na prática, as implementações normalmente convertem as funções definidas pelo usuário em strings de código-fonte JavaScript. A classe Date define um método `toString()` que retorna uma string de data e hora legível para seres humanos (e que pode ser analisada por JavaScript). A classe RegExp define um método `toString()` que converte objetos RegExp em uma string semelhante a um literal RegExp:

```
[1,2,3].toString()                  // => "1,2,3"
(function(x) { f(x); }).toString()  // => "function(x) {\n    f(x);\n}"
/\d+/g.toString()                   // => "/\\d+/g"
new Date(2010,0,1).toString()       // => "Sexta-feira 01 de Janeiro de 2010 00:00:00
                                    //     GMT-0800 (PST)"
```

A outra função de conversão de objeto é chamada `valueOf()`. A tarefa desse método é menos bem definida: ele deve converter um objeto em um valor primitivo que represente o objeto, caso exista tal valor primitivo. Os objetos são valores compostos e a maioria deles não pode ser representada por um único valor primitivo; portanto, o método padrão `valueOf()` simplesmente retorna o próprio objeto, em vez de retornar um valor primitivo. As classes wrapper definem métodos `valueOf()` que retornam o valor primitivo empacotado. Os arrays, as funções e as expressões regulares simplesmente herdam o método padrão. Chamar `valueOf()` para instâncias desses tipos simplesmente retorna o próprio objeto. A classe Date define um método `valueOf()` que retorna a data em sua representação interna: o número de milissegundos desde 1º de janeiro de 1970:

```
var d = new Date(2010, 0, 1);    // 1º de janeiro de 2010, (hora do Pacífico)
d.valueOf()                      // => 1262332800000
```

Explicados os métodos `toString()` e `valueOf()`, podemos agora abordar as conversões de objeto para string e de objeto para número. Note, contudo, que existem alguns casos especiais nos quais JavaScript realiza uma conversão diferente de objeto para valor primitivo. Esses casos especiais estão abordados no final desta seção.

Para converter um objeto em uma string, JavaScript executa estas etapas:

- Se o objeto tem um método `toString()`, JavaScript o chama. Se ele retorna um valor primitivo, JavaScript converte esse valor em uma string (se já não for uma string) e retorna o resultado dessa conversão. Note que as conversões de valor primitivo para string estão todas bem definidas na Tabela 3-2.
- Se o objeto não tem método `toString()` ou se esse método não retorna um valor primitivo, então JavaScript procura um método `valueOf()`. Se o método existe, JavaScript o chama. Se o valor de retorno é primitivo, a JavaScript converte esse valor em uma string (se ainda não for) e retorna o valor convertido.
- Caso contrário, JavaScript não pode obter um valor primitivo nem de `toString()` nem de `valueOf()`; portanto, lança um TypeError.

Para converter um objeto em um número, JavaScript faz a mesma coisa, mas tenta primeiro o método `valueOf()`:

- Se o objeto tem um método `valueOf()` que retorna um valor primitivo, JavaScript converte (se necessário) esse valor primitivo em um número e retorna o resultado.
- Caso contrário, se o objeto tem um método `toString()` que retorna um valor primitivo, JavaScript converte e retorna o valor.
- Caso contrário, JavaScript lança um TypeError.

Os detalhes dessa conversão de objeto para número explicam porque um array vazio é convertido no número 0 e porque um array com um único elemento também pode ser convertido em um número. Os arrays herdam o método padrão `valueOf()` que retorna um objeto, em vez de um valor primitivo, de modo que a conversão de array para número conta com o método `toString()`. Os arrays vazios são convertidos na string vazia. E a string vazia é convertida no número 0. Um array com um único elemento é convertido na mesma string em que esse único elemento é convertido. Se um array contém um único número, esse número é convertido em uma string e, então, de volta para um número.

Em JavaScript, o operador `+` efetua adição numérica e concatenação de strings. Se um de seus operandos é um objeto, JavaScript converte o objeto usando uma conversão de objeto para valor primitivo especial, em vez da conversão de objeto para número utilizada pelos outros operadores aritméticos. O operador de igualdade `==` é semelhante. Se solicitado a comparar um objeto com um valor primitivo, ele converte o objeto usando a conversão de objeto para valor primitivo.

A conversão de objeto para valor primitivo utilizada por `+` e `==` inclui um caso especial para objetos Date. A classe Date é o único tipo predefinido de JavaScript básica que define conversões significativas para strings e para números. A conversão de objeto para valor primitivo é basicamente uma conversão de objeto para número (`valueof()` primeiro) para todos os objetos que não são datas e uma conversão de objeto para string (`toString()` primeiro) para objetos Date. Contudo, a conversão não é exatamente igual àquelas explicadas anteriormente: o valor primitivo retornado por `valueOf()` ou por `toString()` é usado diretamente, sem ser forçado a ser um número ou uma string.

O operador `<` e os outros operadores relacionais realizam conversões de objeto para valores primitivos, assim como `==`, mas sem o caso especial para objetos Date: todo objeto é convertido tentando `valueOf()` primeiro e depois `toString()`. Seja qual for o valor primitivo obtido, é utilizado diretamente sem ser convertido em um número ou em uma string.

`+`, `==`, `!=` e os operadores relacionais são os únicos que realizam esses tipos especiais de conversões de string para valores primitivos. Os outros operadores convertem mais explicitamente para um tipo especificado e não têm qualquer caso especial para objetos Date. O operador `-`, por exemplo, converte seus operandos em números. O código a seguir demonstra o comportamento de `+`, `-`, `==` e `>` com objetos Date:

```
var now = new Date(); // Cria um objeto Date
typeof (now + 1)      // => "string": + converte datas em strings
typeof (now - 1)      // => "number": - usa conversão de objeto para número
```

```
now == now.toString() // => verdadeiro: conversões de string implícitas e explícitas
now > (now -1)        // => verdadeiro: > converte um objeto Date em número
```

3.9 Declaração de variável

Antes de utilizar uma variável em um programa JavaScript, você deve *declará-la*. As variáveis são declaradas com a palavra-chave `var`, como segue:

```
var i;
var sum;
```

Também é possível declarar várias variáveis com a mesma palavra-chave `var`:

```
var i, sum;
```

E pode-se combinar a declaração da variável com sua inicialização:

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

Se não for especificado um valor inicial para uma variável com a instrução `var`, a variável será declarada, mas seu valor será `undefined` até que o código armazene um valor nela.

Note que a instrução `var` também pode aparecer como parte dos laços `for` e `for/in` (apresentados no Capítulo 5), permitindo declarar a variável do laço sucintamente como parte da própria sintaxe do laço. Por exemplo:

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var p in o) console.log(p);
```

Se você está acostumado com linguagens tipadas estaticamente, como C ou Java, terá notado que não existe tipo algum associado às declarações de variável em JavaScript. Uma variável em JavaScript pode conter um valor de qualquer tipo. Por exemplo, em JavaScript é perfeitamente válido atribuir um número a uma variável e posteriormente atribuir uma string a essa variável:

```
var i = 10;
i = "ten";
```

3.9.1 Declarações repetidas e omitidas

É válido e inofensivo declarar uma variável mais de uma vez com a instrução `var`. Se a declaração repetida tem um inicializador, ela atua como se fosse simplesmente uma instrução de atribuição.

Se você tenta ler o valor de uma variável não declarada, JavaScript gera um erro. No modo restrito de ECMAScript 5 (Seção 5.7.3), também é um erro atribuir um valor a uma variável não declarada. Historicamente, contudo, e no modo não restrito, se você atribui um valor a uma variável não declarada, JavaScript cria essa variável como uma propriedade do objeto global e ela funciona de forma muito parecida (mas não exatamente igual, consulte a Seção 3.10.2) a uma variável global declarada corretamente. Isso significa que você pode deixar suas variáveis globais sem declaração. No entanto, esse é um hábito ruim e uma fonte de erros – você sempre deve declarar suas variáveis com `var`.

3.10 Escopo de variável

O *escopo* de uma variável é a região do código-fonte de seu programa em que ela está definida. Uma *variável global* tem escopo global; ela está definida em toda parte de seu código JavaScript. Por outro lado, as variáveis declaradas dentro de uma função estão definidas somente dentro do corpo da função. Elas são *variáveis locais* e têm escopo local. Os parâmetros de função também contam como variáveis locais e estão definidos somente dentro do corpo da função.

Dentro do corpo de uma função, uma variável local tem precedência sobre uma variável global com o mesmo nome. Se você declara uma variável local ou um parâmetro de função com o mesmo nome de uma variável global, ela efetivamente oculta a variável global:

```
var scope = "global";           // Declara uma variável global
function checkscope() {
    var scope = "local";        // Declara uma variável local com o mesmo nome
    return scope;               // Retorna o valor local, não o global
}
checkscope()                    // => "local"
```

Embora seja possível não utilizar a instrução `var` ao escrever código no escopo global, `var` sempre deve ser usada para declarar variáveis locais. Considere o que acontece se você não faz isso:

```
scope = "global";               // Declara uma variável global, mesmo sem var.
function checkscope2() {
    scope = "local";            // Opa! Simplesmente alteramos a variável global.
    myscope = "local";          // Isso declara uma nova variável global implicitamente.
    return [scope, myscope];    // Retorna dois valores.
}
checkscope2()                   // => ["local", "local"]: tem efeitos colaterais!
scope                           // => "local": a variável global mudou.
myscope                         // => "local": namespace global desordenado.
```

As definições de função podem ser aninhadas. Cada função tem seu próprio escopo local; portanto, é possível ter várias camadas de escopo local aninhadas. Por exemplo:

```
var scope = "global scope";          // Uma variável global
function checkscope() {
    var scope = "local scope";       // Uma variável local
    function nested() {
        var scope = "nested scope"; // Um escopo aninhado de variáveis locais
        return scope;               // Retorna o valor em scope aqui
    }
    return nested();
}
checkscope()                         // => "nested scope"
```

3.10.1 Escopo de função e içamento

Em algumas linguagens de programação semelhantes ao C, cada bloco de código dentro de chaves tem seu escopo próprio e as variáveis não são visíveis fora do bloco em que são declaradas. Isso é chamado de *escopo de bloco* e JavaScript *não* tem esse conceito. Em vez disso, JavaScript utiliza *escopo*

de função: as variáveis são visíveis dentro da função em que são definidas e dentro de qualquer função que esteja aninhada dentro dessa função.

No código a seguir, as variáveis `i`, `j` e `k` são declaradas em diferentes pontos, mas todas têm o mesmo escopo – todas as três estão definidas para todo o corpo da função:

```
function test(o) {
    var i = 0;                        // i está definida para toda a função
    if (typeof o == "object") {
        var j = 0;                    // j está definida por toda parte e não apenas no
                                      // bloco
        for(var k=0; k < 10; k++) {   // k está definida por toda parte e não apenas no
                                      // laço
            console.log(k);           // imprime os números de 0 a 9
        }
        console.log(k);               // k ainda está definida: imprime 10
    }
    console.log(j);                   // j está definida, mas não pode ser inicializada
}
```

O escopo de função em JavaScript significa que todas as variáveis declaradas dentro de uma função são visíveis *por todo* o corpo da função. Curiosamente, isso significa que as variáveis são visíveis mesmo antes de serem declaradas. Essa característica de JavaScript é informalmente conhecida como *içamento*: o código JavaScript se comporta como se todas as declarações de variável em uma função (mas não em qualquer atribuição associada) fossem "içadas" para o topo da função. Considere o código a seguir:

```
var scope = "global";
function f() {
    console.log(scope);          // Imprime "undefined" e não "global"
    var scope = "local";         // Variável inicializada aqui, mas definida por toda
                                 // parte console.log(scope);
                                 // Imprime "local"
}
```

Você poderia pensar que a primeira linha da função imprimiria "global", pois a instrução `var` que declara a variável local ainda não foi executada. Contudo, devido às regras de escopo de função, não é isso que acontece. A variável local está definida em todo o corpo da função, ou seja, a variável global de mesmo nome fica oculta por toda a função. Embora a variável local seja definida em toda parte, ela não é inicializada até que a instrução `var` seja executada. Assim, a função anterior é equivalente à seguinte, na qual a declaração da variável é "içada" para o topo e a inicialização da variável é deixada onde está:

```
function f() {
    var scope;                   // A variável local é declarada no topo da função
    console.log(scope);          // Ela existe aqui, mas ainda tem valor "indefinido"
    scope = "local";             // Agora a inicializamos e fornecemos a ela um valor
    console.log(scope);          // E aqui ela tem o valor que esperamos
}
```

Nas linguagens de programação com escopo de bloco, geralmente é considerada uma boa prática de programação declarar as variáveis o mais próximo possível de onde elas são usadas e com o escopo mais limitado possível. Como JavaScript não tem escopo de bloco, alguns programadores fazem

questão de declarar todas as suas variáveis no início da função, em vez de tentar declará-las mais próximas ao ponto em que são utilizadas. Essa técnica faz o código-fonte refletir precisamente o verdadeiro escopo das variáveis.

3.10.2 Variáveis como propriedades

Quando se declara uma variável global em JavaScript, o que se está fazendo realmente é definindo uma propriedade do objeto global (Seção 3.5). Se `var` é utilizada para declarar a variável, a propriedade criada não pode ser configurada (consulte a Seção 6.7), ou seja, não pode ser excluída com o operador `delete`. Já observamos que, se o modo restrito não está sendo usado e um valor é atribuído a uma variável não declarada, JavaScript cria uma variável global automaticamente. As variáveis criadas dessa maneira são propriedades normais e configuráveis do objeto global e podem ser excluídas:

```
var truevar = 1;        // Uma variável global declarada corretamente e que não pode ser
                        // excluída.
fakevar = 2;            // Cria uma propriedade que pode ser excluída do objeto global.
this.fakevar2 = 3;      // Isso faz a mesma coisa.
delete truevar          // => falso: a variável não é excluída
delete fakevar          // => verdadeiro: a variável é excluída
delete this.fakevar2    // => verdadeiro: a variável é excluída
```

As variáveis globais em JavaScript são propriedades do objeto global e isso é imposto pela especificação ECMAScript. Não existe esse requisito para variáveis locais, mas você pode imaginar as variáveis locais como propriedades de um objeto associado à cada chamada de função. A especificação ECMAScript 3 se referia a esse objeto como "objeto de chamada" e a especificação ECMAScript 5 o chama de "registro declarado do ambiente de execução". JavaScript nos permite fazer referência ao objeto global com a palavra-chave `this`, mas não nos fornece qualquer maneira de referenciar o objeto no qual as variáveis locais são armazenadas. A natureza precisa desses objetos que contêm variáveis locais é um detalhe da implementação que não precisa nos preocupar. Contudo, a ideia de que esses objetos de variável local existem é importante e isso será mais bem explicado na próxima seção.

3.10.3 O encadeamento de escopo

JavaScript é uma linguagem com *escopo léxico*: o escopo de uma variável pode ser considerado como o conjunto de linhas de código-fonte para as quais a variável está definida. As variáveis globais estão definidas para todo o programa. As variáveis locais estão definidas para toda a função na qual são declaradas e também dentro de qualquer função aninhada dentro dessa função.

Se pensarmos nas variáveis locais como propriedades de algum tipo de objeto definido pela implementação, então há outro modo de considerarmos o escopo das variáveis. Cada trecho de código JavaScript (código ou funções globais) tem um *encadeamento de escopo* associado. Esse encadeamento de escopo é uma lista ou encadeamento de objetos que define as variáveis que estão "no escopo" para esse código. Quando JavaScript precisa pesquisar o valor de uma variável `x` (um processo chamado *solução de variável*), ela começa examinando o primeiro objeto do encadeamento. Se esse objeto tem uma propriedade chamada `x`, o valor dessa propriedade é usado. Se o primeiro objeto não tem uma propriedade chamada `x`, JavaScript continua a busca no próximo objeto do encadeamento. Se o segundo objeto não tem uma propriedade chamada `x`, a busca passa para o objeto seguinte e assim por diante. Se `x` não for uma propriedade de nenhum dos objetos do encadeamento de escopo, então `x` não está no escopo desse código e ocorre um ReferenceError.

No código JavaScript de nível superior (isto é, código não contido dentro de qualquer definição de função), o encadeamento de escopo consiste em um único objeto, o objeto global. Em uma função não aninhada, o encadeamento de escopo consiste em dois objetos. O primeiro é o objeto que define os parâmetros e as variáveis locais da função e o segundo é o objeto global. Em uma função aninhada, o encadeamento de escopo tem três ou mais objetos. É importante entender como esse encadeamento de objetos é criado. Quando uma função é definida, ela armazena o encadeamento de escopo que está em vigor. Quando essa função é chamada, ela cria um novo objeto para armazenar suas variáveis locais e adiciona esse novo objeto no encadeamento de escopo armazenado para criar um novo encadeamento maior, representando o escopo dessa chamada de função. Isso se torna mais interessante para funções aninhadas, pois sempre que a função externa é chamada, a função interna é novamente definida. Como o encadeamento de escopo é diferente em cada chamada da função externa, a função interna vai ser ligeiramente diferente cada vez que for definida – o código da função interna vai ser idêntico em cada chamada da função externa, mas o encadeamento de escopo associado a esse código vai ser diferente.

Essa ideia de encadeamento de escopo é útil para se entender a instrução `with` (Seção 5.7.1) e fundamental para se entender os fechamentos (Seção 8.6).