Capítulo 5
Instruções

O Capítulo 4 descreveu as expressões em JavaScript como frases. De acordo com essa analogia, *instruções* são sentenças ou comandos em JavaScript. Assim como as sentenças nos idiomas humanos são terminadas e separadas por pontos-finais, as instruções em JavaScript são terminadas com ponto e vírgula (Seção 2.5). As expressões são *avaliadas* para produzir um valor, mas as instruções são *executadas* para fazer algo acontecer.

Uma maneira de "fazer algo acontecer" é avaliar uma expressão que tenha efeitos colaterais. As expressões com efeitos colaterais, como as atribuições e as chamadas de função, podem aparecer sozinhas como instruções e, quando utilizadas dessa maneira, são conhecidas como *instruções de expressão*. Uma categoria similar de instruções são as *instruções de declaração,* que declaram novas variáveis e definem novas funções.

Os programas JavaScript nada mais são do que uma sequência de instruções a serem executadas. Por padrão, o interpretador JavaScript executa essas instruções uma após a outra, na ordem em que são escritas. Outro modo de "fazer algo acontecer" é alterar essa ordem de execução padrão, sendo que JavaScript tem várias instruções ou *estruturas de controle* que fazem justamente isso:

- As *condicionais* são instruções como `if` e `switch` que fazem o interpretador JavaScript executar ou pular outras instruções, dependendo do valor de uma expressão.
- *Laços* são instruções como `while` e `for` que executam outras instruções repetidas vezes.
- *Saltos* são instruções como `break`, `return` e `throw` que fazem o interpretador pular para outra parte do programa.

As seções a seguir descrevem as várias instruções de JavaScript e explicam sua sintaxe. A Tabela 5-1, no final do capítulo, resume a sintaxe. Um programa JavaScript é simplesmente uma sequência de instruções, separadas umas das outras com pontos e vírgulas; portanto, uma vez que você conheça as instruções de JavaScript, pode começar a escrever programas em JavaScript.

5.1 Instruções de expressão

Os tipos mais simples de instruções em JavaScript são as expressões que têm efeitos colaterais. (Mas consulte a Seção 5.7.3 para ver uma importante instrução de expressão sem efeitos colaterais.) Esse tipo de instrução foi mostrado no Capítulo 4. As instruções de atribuição são uma categoria importante de instrução de expressão. Por exemplo:

```
greeting = "Hello " + name;
i *= 3;
```

Os operadores de incremento e decremento, `++` e `--`, são relacionados às instruções de atribuição. Eles têm o efeito colateral de alterar o valor de uma variável, exatamente como se fosse feita uma atribuição:

```
counter++;
```

O operador `delete` tem o importante efeito colateral de excluir uma propriedade de um objeto. Assim, ele é quase sempre utilizado como uma instrução e não como parte de uma expressão maior:

```
delete o.x;
```

As chamadas de função são outra categoria importante de instrução de expressão. Por exemplo:

```
alert(greeting);
window.close();
```

Essas chamadas de função no lado do cliente são expressões, mas têm efeitos colaterais que afetam o navegador Web e são utilizadas aqui como instruções. Se uma função não tem qualquer efeito colateral, não tem sentido chamá-la, a não ser que faça parte de uma expressão maior ou de uma instrução de atribuição. Por exemplo, você não calcularia um cosseno e simplesmente descartaria o resultado:

```
Math.cos(x);
```

Mas poderia calcular o valor e atribuí-lo a uma variável para uso futuro:

```
cx = Math.cos(x);
```

Note que cada linha de código de cada um desses exemplos é terminada com um ponto e vírgula.

5.2 Instruções compostas e vazias

Assim como o operador vírgula (Seção 4.13.5) combina várias expressões em uma, um *bloco de instruções* combina várias instruções em uma única *instrução composta*. Um bloco de instruções é simplesmente uma sequência de instruções colocadas dentro de chaves. Assim, as linhas a seguir atuam como uma única instrução e podem ser usadas em qualquer lugar em que JavaScript espere uma única instrução:

```
{
    x = Math.PI;
    cx = Math.cos(x);
    console.log("cos(π) = " + cx);
}
```

Existem algumas coisas a observar a respeito desse bloco de instruções. Primeiramente, ele *não* termina com um ponto e vírgula. As instruções primitivas dentro do bloco terminam em pontos e vírgulas, mas o bloco em si, não. Segundo, as linhas dentro do bloco são recuadas em relação às chaves que as englobam. Isso é opcional, mas torna o código mais fácil de ler e entender. Por fim, lembre-se de que JavaScript não tem escopo de bloco e as variáveis declaradas dentro de um bloco de instruções não são privativas do bloco (consulte a Seção 3.10.1 para ver os detalhes).

Combinar instruções em blocos de instrução maiores é extremamente comum na programação JavaScript. Assim como as expressões frequentemente contêm subexpressões, muitas instruções JavaScript contêm subinstruções. Formalmente, a sintaxe de JavaScript em geral permite uma única subinstrução. Por exemplo, a sintaxe do laço `while` inclui uma única instrução que serve como corpo do laço. Usando-se um bloco de instruções, é possível colocar qualquer número de instruções dentro dessa única subinstrução permitida.

Uma instrução composta permite utilizar várias instruções onde a sintaxe de JavaScript espera uma única instrução. A *instrução vazia* é o oposto: ela permite não colocar nenhuma instrução onde uma é esperada. A instrução vazia é a seguinte:

```
;
```

O interpretador JavaScript não faz nada ao executar uma instrução vazia. Ocasionalmente, a instrução vazia é útil quando se quer criar um laço com corpo vazio. Considere o laço `for` a seguir (os laços `for` vão ser abordados na Seção 5.5.3):

```
// Inicializa um array a
for(i = 0; i < a.length; a[i++] = 0) ;
```

Nesse laço, todo o trabalho é feito pela expressão `a[i++] = 0` e nenhum corpo é necessário no laço. Contudo, a sintaxe de JavaScript exige uma instrução como corpo do laço, de modo que é utilizada uma instrução vazia – apenas um ponto e vírgula.

Note que a inclusão acidental de um ponto e vírgula após o parêntese da direita de um laço `for`, laço `while` ou instrução `if` pode causar erros frustrantes e difíceis de detectar. Por exemplo, o código a seguir provavelmente não faz o que o autor pretendia:

```
if ((a == 0) || (b == 0));      // Opa! Esta linha não faz nada...
    o = null;                   // e esta linha é sempre executada.
```

Ao se usar a instrução vazia intencionalmente, é uma boa ideia comentar o código de maneira que deixe claro que isso está sendo feito de propósito. Por exemplo:

```
for(i = 0; i < a.length; a[i++] = 0) /* vazio */ ;
```

5.3 Instruções de declaração

`var` e `function` são *instruções de declaração* – elas declaram ou definem variáveis e funções. Essas instruções definem identificadores (nomes de variável e função) que podem ser usados em qualquer parte de seu programa e atribuem valores a esses identificadores. As instruções de declaração

sozinhas não fazem muita coisa, mas criando variáveis e funções, o que é importante, elas definem o significado das outras instruções de seu programa.

As subseções a seguir explicam a instrução `var` e a instrução `function`, mas não abordam as variáveis e funções amplamente. Consulte a Seção 3.9 e a Seção 3.10 para mais informações sobre variáveis. E consulte o Capítulo 8 para detalhes completos sobre funções.

5.3.1 var

A instrução `var` declara uma (ou mais) variável. Aqui está a sintaxe:

```
var nome_1 [ = valor_1] [ ,..., nome_n [= valor_n]]
```

A palavra-chave `var` é seguida por uma lista separada com vírgulas de variáveis a declarar; opcionalmente, cada variável da lista pode ter uma expressão inicializadora especificando seu valor inicial. Por exemplo:

```
var  i;                                          // Uma variável simples
var  j = 0;                                      // Uma var, um valor
var  p, q;                                       // Duas variáveis
var  greeting = "hello" + name;                  // Um inicializador complexo
var  x = 2.34, y = Math.cos(0.75), r, theta;     // Muitas variáveis
var  x = 2, y = x*x;                             // A segunda variável usa a primeira
var  x = 2,                                      // Diversas variáveis...
     f = function(x) { return x*x },             // cada uma em sua própria linha
     y = f(x);
```

Se uma instrução `var` aparece dentro do corpo de uma função, ela define variáveis locais com escopo nessa função. Quando `var` é usada em código de nível superior, ela declara variáveis globais, visíveis em todo o programa JavaScript. Conforme observado na Seção 3.10.2, as variáveis globais são propriedades do objeto global. Contudo, ao contrário das outras propriedades globais, as propriedades criadas com `var` não podem ser excluídas.

Se nenhum inicializador é especificado para uma variável com a instrução `var`, o valor inicial da variável é `undefined`. Conforme descrito na Seção 3.10.1, as variáveis são definidas por todo o script ou função na qual são declaradas – suas declarações são "içadas" para o início do script ou função. No entanto, a inicialização ocorre no local da instrução `var` e o valor da variável é `undefined` antes desse ponto no código.

Note que a instrução `var` também pode aparecer como parte dos laços `for` e `for/in`. (Essas variáveis são içadas, exatamente como as variáveis declaradas fora de um laço.) Aqui estão exemplos, repetidos da Seção 3.9:

```
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 i in o) console.log(i);
```

Note que não tem problema declarar a mesma variável várias vezes.

5.3.2 function

A palavra-chave `function` é usada para definir funções. Nós a vimos em expressões de definição de função, na Seção 4.3. Ela também pode ser usada em forma de instrução. Considere as duas funções a seguir:

```
var f = function(x) { return x+1; }  // Expressão atribuída a uma variável
function f(x) { return x+1; }        // A instrução inclui nome de variável
```

Uma instrução de declaração de função tem a seguinte sintaxe:

```
function nomefun([arg1 [, arg2 [..., argn]]]) {
    instruções
}
```

nomefun é um identificador que dá nome à função que está sendo declarada. O nome da função é seguido por uma lista separada com vírgulas de nomes de parâmetro entre parênteses. Esses identificadores podem ser usados dentro do corpo da função para se referir aos valores de argumento passados quando a função é chamada.

O corpo da função é composto de qualquer número de instruções JavaScript, contidas entre chaves. Essas instruções não são executadas quando a função é definida. Em vez disso, elas são associadas ao novo objeto função, para execução quando a função for chamada. Note que as chaves são uma parte obrigatória da instrução `function`. Ao contrário dos blocos de instrução utilizados com laços `while` e outras instruções, o corpo de uma função exige chaves, mesmo que consista em apenas uma instrução.

Aqui estão mais alguns exemplos de declarações de função:

```
function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y);    // return está documentado na próxima seção
}

function factorial(n) {             // Uma função recursiva
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

As instruções de declaração de função podem aparecer em código JavaScript de nível superior ou estar aninhadas dentro de outras funções. Quando aninhadas, contudo, as declarações de função só podem aparecer no nível superior da função dentro da qual estão aninhadas. Isto é, definições de função não podem aparecer dentro de instruções `if`, laços `while` ou qualquer outra instrução. Por causa dessa restrição sobre onde as declarações de função podem aparecer, a especificação ECMAScript não classifica as declarações de função como verdadeiras instruções. Algumas implementações de JavaScript permitem que declarações de função apareçam onde quer que uma instrução possa aparecer, mas diferentes implementações tratam dos detalhes de formas diferentes e colocar declarações de função dentro de outras instruções não é portável.

As instruções de declaração de função diferem das expressões de definição de função porque incluem um nome de função. As duas formas criam um novo objeto função, mas a instrução de declaração de função também declara o nome da função como variável e atribui o objeto função a ela. Assim como as variáveis declaradas com `var`, as funções definidas com instruções de

definição de função são implicitamente "içadas" para o topo do script ou função que as contém, de modo que sejam visíveis em todo o script ou função. Com `var`, somente a declaração da variável é içada – o código de inicialização da variável permanece onde é colocado. Contudo, com instruções de declaração de função, tanto o nome da função como o corpo da função são içados – todas as funções de um script ou todas as funções aninhadas em uma função são declaradas antes de qualquer outro código ser executado. Isso significa que é possível chamar uma função em JavaScript antes de declará-la.

Assim como a instrução `var`, as instruções de declaração de função criam variáveis que não podem ser excluídas. Contudo, essas variáveis não são somente para leitura e seus valores podem ser sobrescritos.

5.4 Condicionais

As instruções condicionais executam ou pulam outras instruções, dependendo do valor de uma expressão especificada. Essas instruções são os pontos de decisão de seu código e às vezes também são conhecidas como "ramificações". Se você imaginar um interpretador JavaScript seguindo um caminho através de seu código, as instruções condicionais são os lugares onde o código se ramifica em dois ou mais caminhos e o interpretador deve escolher qual caminho seguir.

As subseções a seguir explicam a condicional básica de JavaScript, a instrução `if/else` e também abordam `switch`, uma instrução de ramificação em múltiplos caminhos, mais complicada.

5.4.1 if

A instrução `if` é a instrução de controle fundamental que permite à linguagem JavaScript tomar decisões ou, mais precisamente, executar instruções condicionalmente. Essa instrução tem duas formas. A primeira é:

```
if (expressão)
    instrução
```

Nessa forma, a *`expressão`* é avaliada. Se o valor resultante é verdadeiro, a *`instrução`* é executada. Se a *`expressão`* é falsa, a *`instrução`* não é executada. (Consulte a Seção 3.3 para ver uma definição de valores verdadeiros e falsos.) Por exemplo:

```
if (username == null)           // Se username é null ou undefined,
    username = "John Doe";      // o define
```

Ou, de modo similar:

```
// Se username é null, undefined, false, 0, "" ou NaN, fornece a ele um novo valor
if (!username) username = "John Doe";
```

Note que os parênteses em torno da *`expressão`* são uma parte obrigatória da sintaxe da instrução `if`.

A sintaxe de JavaScript exige uma instrução após a palavra-chave `if` e a expressão entre parênteses, mas pode-se usar um bloco de instruções para combinar várias instruções em uma só. Portanto, a instrução `if` também poderia ser como segue:

```
if (!address) {
    address = "";
    message = "Please specify a mailing address.";
}
```

A segunda forma da instrução `if` introduz uma cláusula `else`, que é executada quando a *`expressão`* é `false`. Sua sintaxe é:

```
if (expressão)
    instrução1
else
    instrução2
```

Essa forma da instrução executa a *`instrução1`* se a *`expressão`* é verdadeira e executa a `instrução2` se a *`expressão`* é falsa. Por exemplo:

```
if (n == 1)
    console.log("You have 1 new message.");
else
    console.log("You have " + n + " new messages.");
```

Quando instruções `if` com cláusulas `else` forem aninhadas, é necessário um certo cuidado para garantir que a cláusula `else` combine com a instrução `if` apropriada. Considere as linhas a seguir:

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
        console.log("i equals k");
else
    console.log("i doesn't equal j");   // ERRADO!!
```

Nesse exemplo, a instrução `if` interna forma a instrução única permitida pela sintaxe da instrução `if` externa. Infelizmente, não está claro (a não ser pela dica dada pelo recuo) com qual `if` a cláusula `else` está relacionada. E, nesse exemplo, o recuo está errado, pois um interpretador JavaScript interpreta o exemplo anterior como:

```
if (i == j) {
    if (j == k)
        console.log("i equals k");
    else
        console.log("i doesn't equal j");        // OPA!
}
```

A regra em JavaScript (assim como na maioria das linguagens de programação) é que, por padrão, uma cláusula `else` faz parte da instrução `if` mais próxima. Para tornar esse exemplo menos ambíguo e mais fácil de ler, entender, manter e depurar, deve-se usar chaves:

```
if (i == j) {
    if (j == k) {
        console.log("i equals k");
    }
}
```

```
else {  // Que diferença faz a posição de uma chave!
    console.log("i doesn't equal j");
}
```

Embora não seja o estilo utilizado neste livro, muitos programadores se habituam a colocar os corpos de instruções `if` e `else` (assim como outras instruções compostas, como laços `while`) dentro de chaves, mesmo quando o corpo consiste em apenas uma instrução. Fazer isso sistematicamente pode evitar o tipo de problema que acabamos de ver.

5.4.2 else if

A instrução `if/else` avalia uma expressão e executa um código ou outro, dependendo do resultado. Mas e quando é necessário executar um entre vários códigos? Um modo de fazer isso é com a instrução `else if`. `else if` não é realmente uma instrução JavaScript, mas apenas um idioma de programação frequentemente utilizado que resulta da repetição de instruções `if/else`:

```
if (n == 1) {
    // Executa o bloco de código #1
}
else if (n == 2) {
    // Executa o bloco de código #2
}
else if (n == 3) {
    // Executa o bloco de código #3
}
else {
    // Se tudo falhar, executa o bloco #4
}
```

Não há nada de especial nesse código. Trata-se apenas de uma série de instruções `if`, onde cada `if` sucessivo faz parte da cláusula `else` da instrução anterior. Usar o idioma `else if` é preferível e mais legível do que escrever essas instruções em sua forma totalmente aninhada e sintaticamente equivalente:

```
if (n == 1) {
    // Executa o bloco de código #1
}
else {
    if (n == 2) {
        // Executa o bloco de código #2
    }
    else {
        if (n == 3) {
            // Executa o bloco de código #3
        }
        else {
            // Se tudo falhar, executa o bloco #4
        }
    }
}
```

5.4.3 switch

Uma instrução `if` causa uma ramificação no fluxo de execução de um programa e é possível usar o idioma `else if` para fazer uma ramificação de vários caminhos. Contudo, essa não é a melhor solução quando todas as ramificações dependem do valor da mesma expressão. Nesse caso, é um desperdício avaliar essa expressão repetidamente em várias instruções `if`.

A instrução `switch` trata exatamente dessa situação. A palavra-chave `switch` é seguida de uma expressão entre parênteses e de um bloco de código entre chaves:

```
switch(expressão) {
    instruções
}
```

Contudo, a sintaxe completa de uma instrução `switch` é mais complexa do que isso. Vários locais no bloco de código são rotulados com a palavra-chave `case`, seguida de uma expressão e dois-pontos. `case` é como uma instrução rotulada, exceto que, em vez de dar um nome à instrução rotulada, ela associa uma expressão à instrução. Quando uma instrução `switch` é executada, ela calcula o valor da *`expressão`* e, então, procura um rótulo `case` cuja expressão seja avaliada com o mesmo valor (onde a semelhança é determinada pelo operador `===`). Se encontra um, ela começa a executar o bloco de código da instrução rotulada por `case`. Se não encontra um `case` com um valor correspondente, ela procura uma instrução rotulada com `default:`. Se não houver um rótulo `default:`, a instrução `switch` pula o bloco de código completamente.

`switch` é uma instrução confusa para explicar; seu funcionamento se torna muito mais claro com um exemplo. A instrução `switch` a seguir é equivalente às instruções `if/else` repetidas, mostradas na seção anterior:

```
switch(n) {
    case 1:                          // Começa aqui se n === 1
        // Executa o bloco de código #1.
        break;
                    // Para aqui
    case 2:                          // Começa aqui se n === 2
        // Executa o bloco de código #2.
        break;                       // Para aqui
    case 3:                          // Começa aqui se n === 3
        // Executa o bloco de código #3.
        break;                       // Para aqui
    default:                         // Se tudo falhar...
        // Executa o bloco de código #4.
        break;                       // Para aqui
}
```

Observe a palavra-chave `break` utilizada no final de cada `case` no código anterior. A instrução `break`, descrita posteriormente neste capítulo, faz o interpretador pular para o final (ou "escapar") da instrução `switch` e continuar na instrução seguinte. As cláusulas `case` em uma instrução `switch` especificam apenas o *ponto de partida* do código desejado; elas não especificam ponto final algum. Na ausência de instruções `break`, uma instrução `switch` começa a executar seu bloco de código no rótulo `case` correspondente ao valor de sua *`expressão`* e continua a executar as instruções até atingir o final

do bloco. Em raras ocasiões, é útil escrever código como esse, que "passa" de um rótulo `case` para o seguinte, mas 99% das vezes deve-se tomar o cuidado de finalizar cada `case` com uma instrução `break`. (Entretanto, ao usar `switch` dentro de uma função, pode-se utilizar uma instrução `return`, em vez de uma instrução `break`. Ambas servem para finalizar a instrução `switch` e impedir que a execução passe para o próximo `case`.)

Aqui está um exemplo mais realista da instrução `switch`; ele converte um valor em uma string de um modo que depende do tipo do valor:

```
function convert(x) {
    switch(typeof x) {
        case 'number':              // Converte o número para um inteiro hexadecimal
            return x.toString(16);
        case 'string':              // Retorna a string colocada entre apóstrofos
            return '"' + x + '"';
        default:                    // Converte qualquer outro tipo da maneira usual
            return String(x);
    }
}
```

Note que, nos dois exemplos anteriores, as palavras-chave `case` são seguidas por literais numéricas e strings literais, respectivamente. É assim que a instrução `switch` é mais frequentemente utilizada na prática, mas note que o padrão ECMAScript permite que cada `case` seja seguido por uma expressão arbitrária.

A instrução `switch` avalia primeiro a expressão que vem após a palavra-chave `switch` e depois avalia as expressões `case`, na ordem em que aparecem, até encontrar um valor que coincida[1]. O case coincidente é determinado usando-se o operador de identidade `===` e não o operador de igualdade `==`, de modo que as expressões devem coincidir sem qualquer conversão de tipo.

Como nem todas as expressões `case` são avaliadas sempre que a instrução `switch` é executada, você deve evitar o uso de expressões `case` que contenham efeitos colaterais, como chamadas de função ou atribuições. O caminho mais seguro é simplesmente limitar suas expressões `case` às expressões constantes.

Conforme explicado anteriormente, se nenhuma das expressões `case` corresponde à expressão `switch`, a instrução `switch` começa a executar seu corpo na instrução rotulada como `default:`. Se não há rótulo `default:`, a instrução `switch` pula seu corpo completamente. Note que, nos exemplos anteriores, o rótulo `default:` aparece no final do corpo de `switch`, após todos os rótulos `case`. Esse é um lugar lógico e comum para ele, mas pode aparecer em qualquer lugar dentro do corpo da instrução.

[1] O fato de as expressões `case` serem avaliadas em tempo de execução torna a instrução `switch` de JavaScript muito diferente (e menos eficiente) da instrução `switch` de C, C++ e Java. Nessas linguagens, as expressões `case` devem ser constantes definidas em tempo de compilação e devem ser do mesmo tipo, sendo que as instruções `switch` frequentemente podem ser compiladas em *tabelas de salto* altamente eficientes.

5.5 Laços

Para entendermos as instruções condicionais, imaginamos o interpretador JavaScript seguindo um caminho de ramificação em seu código-fonte. As *instruções de laço* são àquelas que desviam esse caminho para si mesmas a fim de repetir partes de seu código. JavaScript tem quatro instruções de laço: `while`, `do/while`, `for` e `for/in`. As subseções a seguir explicam cada uma delas, uma por vez. Um uso comum para laços é na iteração pelos elementos de um array. A Seção 7.6 discute esse tipo de laço em detalhes e aborda métodos de laço especiais definidos pela classe Array.

5.5.1 while

Assim como a instrução `if` é a condicional básica de JavaScript, a instrução `while` é o laço básico da linguagem. Ela tem a seguinte sintaxe:

```
while (expressão)
    instrução
```

Para executar uma instrução `while`, o interpretador primeiramente avalia a *`expressão`*. Se o valor da expressão é falso, o interpretador pula a *`instrução`* que serve de corpo do laço e vai para a instrução seguinte no programa. Se, por outro lado, a *`expressão`* é verdadeira, o interpretador executa a *`instrução`* e repete, pulando de volta para o início do laço e avaliando a *`expressão`* novamente. Outra maneira de dizer isso é que o interpretador executa a *`instrução`* repetidamente *enquanto* a *`expressão`* é verdadeira. Note que é possível criar um laço infinito com a sintaxe `while(true)`.

Em geral, você não quer que JavaScript execute exatamente a mesma operação repetidamente. Em quase todo laço, uma ou mais variáveis mudam a cada *iteração*. Como as variáveis mudam, as ações realizadas pela execução da *`instrução`* podem diferir a cada passagem pelo laço. Além disso, se a variável (ou variáveis) que muda está envolvida na *`expressão`*, o valor da expressão pode ser diferente a cada passagem pelo laço. Isso é importante; caso contrário, uma expressão que começasse verdadeira nunca mudaria e o laço nunca terminaria! Aqui está um exemplo de laço `while` que imprime os números de 0 a 9:

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

Como você pode ver, a variável `count` começa em 0 e é incrementada cada vez que o corpo do laço é executado. Quando o laço tiver executado 10 vezes, a expressão se torna `false` (isto é, a variável `count` deixa de ser menor do que 10), a instrução `while` termina e o interpretador pode passar para a próxima instrução do programa. Muitos laços têm uma variável contadora como `count`. Os nomes de variável `i`, `j` e `k` são comumente utilizados como contadores de laço, embora você deva usar nomes mais descritivos se isso tornar seu código mais fácil de entender.

5.5.2 do/while

O laço `do/while` é como um laço `while`, exceto que a expressão do laço é testada no final e não no início do laço. Isso significa que o corpo do laço sempre é executado pelo menos uma vez. A sintaxe é:

```
do
    instrução
while (expressão);
```

O laço `do/while` é menos comumente usado do que seu primo `while` – na prática, é um tanto incomum ter certeza de que se quer executar um laço pelo menos uma vez. Aqui está um exemplo de laço `do/while`:

```
function printArray(a) {
    var len = a.length, i = 0;
    if (len == 0)
        console.log("Empty Array");
    else {
        do {
            console.log(a[i]);
        } while (++i < len);
    }
}
```

Existem duas diferenças sintáticas entre o laço `do/while` e o laço `while` normal. Primeiramente, o laço `do` exige a palavra-chave `do` (para marcar o início do laço) e a palavra-chave `while` (para marcar o fim e introduzir a condição do laço). Além disso, o laço `do` sempre deve ser terminado com um ponto e vírgula. O laço `while` não precisa de ponto e vírgula se o corpo do laço estiver colocado entre chaves.

5.5.3 for

A instrução `for` fornece uma construção de laço frequentemente mais conveniente do que a instrução `while`. A instrução `for` simplifica os laços que seguem um padrão comum. A maioria dos laços tem uma variável contadora de algum tipo. Essa variável é inicializada antes que o laço comece e é testada antes de cada iteração do laço. Por fim, a variável contadora é incrementada ou atualizada de algum modo no final do corpo do laço, imediatamente antes que a variável seja novamente testada. Nesse tipo de laço, a inicialização, o teste e a atualização são as três manipulações fundamentais de uma variável de laço. A instrução `for` codifica cada uma dessas três manipulações como uma expressão e torna essas expressões uma parte explícita da sintaxe do laço:

```
for(inicialização ; teste ; incremento)
    instrução
```

inicialização, *teste* e *incremento* são três expressões (separadas com pontos e vírgulas) que são responsáveis por inicializar, testar e incrementar a variável de laço. Colocar todas elas na primeira linha do laço facilita entender o que um laço `for` está fazendo e evita erros, como esquecer de inicializar ou incrementar a variável de laço.

O modo mais simples de explicar o funcionamento de um laço `for` é mostrando o laço `while` equivalente[2]:

```
inicialização;
  while(teste) {
        instrução
        incremento;
  }
```

Em outras palavras, a expressão *`inicialização`* é avaliada uma vez, antes que o laço comece. Para ser útil, essa expressão deve ter efeitos colaterais (normalmente uma atribuição). JavaScript também permite que *`inicialização`* seja uma instrução de declaração de variável `var`, de modo que é possível declarar e inicializar um contador de laço ao mesmo tempo. A expressão *`teste`* é avaliada antes de cada iteração e controla se o corpo do laço é executado. Se *`teste`* é avaliada como um valor verdadeiro, a *`instrução`* que é o corpo do laço é executada. Por fim, a expressão *`incremento`* é avaliada. Novamente, para ser útil ela deve ser uma expressão com efeitos colaterais. De modo geral, ou ela é uma expressão de atribuição, ou ela utiliza os operadores `++` ou `--`.

Podemos imprimir os números de 0 a 9 com um laço `for`, como segue. Compare isso com o laço `while` equivalente mostrado na seção anterior:

```
for(var count = 0; count < 10; count++)
    console.log(count);
```

É claro que os laços podem se tornar muito mais complexos do que esse exemplo simples e, às vezes, diversas variáveis são alteradas em cada iteração do laço. Essa situação é o único lugar em que o operador vírgula é comumente usado em JavaScript; ele oferece uma maneira de combinar várias expressões de inicialização e incremento em uma única expressão conveniente para uso em um laço `for`:

```
var i,j;
for(i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

Em todos os nossos exemplos de laço até aqui, a variável de laço era numérica. Isso é muito comum, mas não necessário. O código a seguir usa um laço `for` para percorrer uma estrutura de dados tipo lista encadeada e retornar o último objeto da lista (isto é, o primeiro objeto que não tem uma propriedade `next`):

```
function tail(o) {                          // Retorna a cauda da lista encadeada o
    for(; o.next; o = o.next) /* vazio */ ;  // Percorre enquanto o.next é verdadeiro
    return o;
}
```

Note que o código anterior não tem qualquer expressão *`inicialização`*. Qualquer uma das três expressões pode ser omitida de um laço `for`, mas os dois pontos e vírgulas são obrigatórios. Se você omite a expressão *`teste`*, o loop se repete para sempre e `for(;;)` é outra maneira de escrever um laço infinito, assim como `while(true)`.

[2] Quando considerarmos a instrução `continue`, na Seção 5.6.3, vamos ver que esse laço `while` não é um equivalente exato do laço `for`.

5.5.4 for/in

A instrução `for/in` utiliza a palavra-chave `for`, mas é um tipo de laço completamente diferente do laço `for` normal. Um laço `for/in` é como segue:

```
for (variável in objeto)
    instrução
```

`variável` normalmente nomeia uma variável, mas pode ser qualquer expressão que seja avaliada como lvalue (Seção 4.7.3) ou uma instrução `var` que declare uma única variável – deve ser algo apropriado para o lado esquerdo de uma expressão de atribuição. *`objeto`* é uma expressão avaliada como um objeto. Como sempre, *`instrução`* é a instrução ou bloco de instruções que serve como corpo do laço.

É fácil usar um laço `for` normal para iterar pelos elementos de um array:

```
for(var i = 0; i < a.length; i++)    // Atribui índices do array à variável i
    console.log(a[i]);               // Imprime o valor de cada elemento do array
```

O laço `for/in` torna fácil fazer o mesmo para as propriedades de um objeto:

```
for(var p in o)                      // Atribui nomes de propriedade de o à variável p
    console.log(o[p]);               // Imprime o valor de cada propriedade
```

Para executar uma instrução `for/in`, o interpretador JavaScript primeiramente avalia a expressão *`objeto`*. Se for avaliada como `null` ou `undefined`, o interpretador pula o laço e passa para a instrução seguinte[3]. Se a expressão é avaliada como um valor primitivo, esse valor é convertido em seu objeto empacotador equivalente (Seção 3.6). Caso contrário, a expressão já é um objeto. Agora o interpretador executa o corpo do laço, uma vez para cada propriedade enumerável do objeto. Contudo, antes de cada iteração, o interpretador avalia a expressão *`variável`* e atribui o nome da propriedade (um valor de string) a ela.

Note que a *`variável`* no laço `for/in` pode ser uma expressão arbitrária, desde que seja avaliada como algo adequado ao lado esquerdo de uma atribuição. Essa expressão é avaliada em cada passagem pelo laço, ou seja, ela pode ser avaliada de forma diferente a cada vez. Por exemplo, é possível usar código como o seguinte para copiar os nomes de todas as propriedades de objeto em um array:

```
var o = {x:1, y:2, z:3};
var a = [], i = 0;
for(a[i++] in o) /* vazio */;
```

Os arrays em JavaScript são simplesmente um tipo de objeto especializado e os índices de array são propriedades de objeto que podem ser enumeradas com um laço `for/in`. Por exemplo, colocar a linha a seguir no código anterior enumera os índices 0, 1 e 2 do array:

```
for(i in a) console.log(i);
```

O laço `for/in` não enumera todas as propriedades de um objeto, mas somente as que são *enumeráveis* (consulte a Seção 6.7). Os vários métodos internos definidos por JavaScript básica não são enume-

[3] As implementações ECMAScript 3 podem, em vez disso, lançar um TypeError nesse caso.

ráveis. Todos os objetos têm um método `toString()`, por exemplo, mas o laço `for/in` não enumera essa propriedade `toString`. Além dos métodos internos, muitas outras propriedades dos objetos internos não são enumeráveis. Contudo, todas as propriedades e métodos definidos pelo seu código são enumeráveis. (Mas, em ECMAScript 5, é possível torná-los não enumeráveis usando as técnicas explicadas na Seção 6.7.) As propriedades herdadas definidas pelo usuário (consulte a Seção 6.2.2) também são enumeradas pelo laço `for/in`.

Se o corpo de um laço `for/in` exclui uma propriedade que ainda não foi enumerada, essa propriedade não vai ser enumerada. Se o corpo do laço define novas propriedades no objeto, essas propriedades geralmente não vão ser enumeradas. (No entanto, algumas implementações podem enumerar propriedades herdadas, adicionadas depois que o laço começa.)

5.5.4.1 Ordem de enumeração de propriedades

A especificação ECMAScript não define a ordem na qual o laço `for/in` enumera as propriedades de um objeto. Na prática, contudo, as implementações de JavaScript de todos os principais fornecedores de navegador enumeram as propriedades de objetos simples de acordo como foram definidas, com as propriedades mais antigas enumeradas primeiro. Se um objeto foi criado como objeto literal, sua ordem de enumeração é a mesma das propriedades que aparecem no literal. Existem sites e bibliotecas na Web que contam com essa ordem de enumeração e é improvável que os fornecedores de navegador a alterem.

O parágrafo anterior especifica uma ordem de enumeração de propriedade que serve indistintamente para objetos "simples". A ordem de enumeração se torna dependente da implementação (e não serve indistintamente) se:

- o objeto herda propriedades enumeráveis;
- o objeto tem propriedades que são índices inteiros de array;
- `delete` foi usado para excluir propriedades existentes do objeto; ou
- `Object.defineProperty()` (Seção 6.7) ou métodos semelhantes foram usados para alterar atributos da propriedade do objeto.

Normalmente (mas não em todas as implementações), as propriedades herdadas (consulte a Seção 6.2.2) são enumeradas depois de todas as propriedades "próprias" não herdadas de um objeto, mas também são enumeradas na ordem em que foram definidas. Se um objeto herda propriedades de mais de um "protótipo" (consulte a Seção 6.1.3) – isto é, se ele tem mais de um objeto em seu "encadeamento de protótipos" –, então as propriedades de cada objeto protótipo do encadeamento são enumeradas na ordem de criação, antes da enumeração das propriedades do objeto seguinte. Algumas implementações (mas não todas) enumeram propriedades de array na ordem numérica, em vez de usar a ordem de criação, mas revertem a ordem de criação se o array também receber outras propriedades não numéricas ou se o array for esparso (isto é, se estão faltando alguns índices do array).

5.6 Saltos

Outra categoria de instruções de JavaScript são as *instruções de salto*. Conforme o nome lembra, elas fazem o interpretador JavaScript saltar para um novo local no código-fonte. A instrução `break` faz o interpretador saltar para o final de um laço ou para outra instrução. `continue` faz o interpretador pular o restante do corpo de um laço e voltar ao início de um laço para começar uma nova iteração. JavaScript permite que as instruções sejam nomeadas (ou *rotuladas*), sendo que `break` e `continue` podem identificar o laço de destino ou outro rótulo de instrução.

A instrução `return` faz o interpretador saltar de uma chamada de função de volta para o código que a chamou e também fornece o valor para a chamada. A instrução `throw` provoca (ou "lança") uma exceção e foi projetada para trabalhar com a instrução `try/catch/finally`, a qual estabelece um bloco de código de tratamento de exceção. Esse é um tipo complicado de instrução de salto: quando uma exceção é lançada, o interpretador pula para a rotina de tratamento de exceção circundante mais próxima, a qual pode estar na mesma função ou acima na pilha de chamada, em uma função invocadora.

Os detalhes de cada uma dessas instruções de salto estão nas seções a seguir.

5.6.1 Instruções rotuladas

Qualquer instrução pode ser *rotulada* por ser precedida por um identificador e dois-pontos:

identificador: *instrução*

Rotulando uma instrução, você dá a ela um nome que pode ser usado para se referir a ela em qualquer parte de seu programa. É possível rotular qualquer instrução, embora só seja útil rotular instruções que tenham corpos, como laços e condicionais. Dando um nome a um laço, você pode usar instruções `break` e `continue` dentro do corpo do laço para sair dele ou para pular diretamente para o seu início, a fim de começar a próxima iteração. `break` e `continue` são as únicas instruções em JavaScript que utilizam rótulos; elas são abordadas posteriormente neste capítulo. Aqui está um exemplo de laço `while` rotulado e de uma instrução `continue` que utiliza o rótulo.

```
mainloop: while(token != null) {
    // Código omitido...
    continue mainloop;          // Pula para a próxima iteração do laço nomeado
    // Mais código omitido...
}
```

O *identificador* utilizado para rotular uma instrução pode ser qualquer identificador JavaScript válido, que não seja uma palavra reservada. O espaço de nomes para rótulos é diferente do espaço de nomes para variáveis e funções; portanto, pode-se usar o mesmo identificador como rótulo de instrução e como nome de variável ou função. Os rótulos de instrução são definidos somente dentro da instrução na qual são aplicados (e dentro de suas subinstruções, evidentemente). Uma instrução pode não ter o mesmo rótulo de uma instrução que a contém, mas duas instruções podem ter o mesmo rótulo, desde que nenhuma delas esteja aninhada dentro da outra. As próprias instruções rotuladas podem ser rotuladas. Efetivamente, isso significa que qualquer instrução pode ter vários rótulos.

5.6.2 break

A instrução `break`, utilizada sozinha, faz com que o laço ou instrução `switch` circundante mais interna seja abandonada imediatamente. Sua sintaxe é simples:

```
break;
```

Como ela é usada para sair de um laço ou `switch` para sair, essa forma da instrução `break` é válida apenas dentro de uma dessas instruções.

Já vimos exemplos da instrução `break` dentro de uma instrução `switch`. Em laços, ela é normalmente utilizada para sair prematuramente, quando, por qualquer motivo, não há mais qualquer necessidade de completar o laço. Quando um laço tem condições de término complexas, frequentemente é mais fácil implementar algumas dessas condições com instruções `break` do que tentar expressar todas elas em uma única expressão de laço. O código a seguir procura um valor específico nos elementos de um array. O laço termina normalmente ao chegar no fim do array; ele termina com uma instrução `break` se encontra o que está procurando no array:

```
for(var i = 0; i < a.length; i++) {
    if (a[i] == target) break;
}
```

JavaScript também permite que a palavra-chave `break` seja seguida por um rótulo de instrução (apenas o identificador, sem os dois-pontos):

```
break nomerótulo;
```

Quando a instrução `break` é usada com um rótulo, ela pula para o final (ou termina) da instrução circundante que tem o rótulo especificado. Se não houver qualquer instrução circundante com o rótulo especificado, é um erro de sintaxe usar `break` dessa forma. Nessa forma da instrução `break`, a instrução nomeada não precisa ser um laço ou `switch`: `break` pode "sair de" qualquer instrução circundante. Essa instrução pode até ser um bloco de instruções agrupadas dentro de chaves, com o único objetivo de nomear o bloco com um rótulo.

Não é permitido um caractere de nova linha entre a palavra-chave `break` e *`nomerótulo`*. Isso é resultado da inserção automática de pontos e vírgulas omitidos de JavaScript: se um terminador de linha é colocado entre a palavra-chave `break` e o rótulo que se segue, JavaScript presume que se quis usar a forma simples, não rotulada, da instrução e trata o terminador de linha como um ponto e vírgula. (Consulte a Seção 2.5.)

A forma rotulada da instrução `break` é necessária quando se quer sair de uma instrução que não é o laço ou uma instrução switch circundante mais próxima. O código a seguir demonstra isso:

```
var matrix = getData();          // Obtém um array 2D de números de algum lugar
// Agora soma todos os números da matriz.
var sum = 0, success = false;
// Começa com uma instrução rotulada da qual podemos sair se ocorrerem erros
compute_sum: if (matrix) {
    for(var x = 0; x < matrix.length; x++) {
        var row = matrix[x];
        if (!row) break compute_sum;
```

```
            for(var y = 0; y < row.length; y++) {
                var cell = row[y];
                if (isNaN(cell)) break compute_sum;
                sum += cell;
            }
        }
        success = true;
    }
    // As instruções break pulam para cá. Se chegamos aqui com success == false,
    // então algo deu errado com a matriz que recebemos.
    // Caso contrário, sum contém a soma de todas as células da matriz.
```

Por fim, note que uma instrução `break`, com ou sem rótulo, não pode transferir o controle para além dos limites da função. Não se pode rotular uma instrução de definição de função, por exemplo, e depois usar esse rótulo dentro da função.

5.6.3 continue

A instrução `continue` é semelhante à instrução `break`. No entanto, em vez de sair de um laço, `continue` reinicia um laço na próxima iteração. A sintaxe da instrução `continue` é tão simples quanto a da instrução `break`:

```
continue;
```

A instrução `continue` também pode ser usada com um rótulo:

```
continue nomerótulo;
```

A instrução `continue`, tanto em sua forma rotulada como na não rotulada, só pode ser usada dentro do corpo de um laço. Utilizá-la em qualquer outro lugar causa erro de sintaxe.

Quando a instrução `continue` é executada, a iteração atual do laço circundante é terminada e a próxima iteração começa. Isso significa coisas diferentes para diferentes tipos de laços:

- Em um laço `while`, a *`expressão`* especificada no início do laço é testada novamente e, se for `true`, o corpo do laço é executado desde o início.
- Em um laço `do/while`, a execução pula para o final do laço, onde a condição de laço é novamente testada, antes de recomeçar o laço desde o início.
- Em um laço `for`, a expressão de *`incremento`* é avaliada e a expressão de *`teste`* é novamente testada para determinar se deve ser feita outra iteração.
- Em um laço `for/in`, o laço começa novamente com o próximo nome de propriedade sendo atribuído à variável especificada.

Note a diferença no comportamento da instrução `continue` nos laços `while` e `for`: um laço `while` retorna diretamente para sua condição, mas um laço `for` primeiramente avalia sua expressão de *`incremento`* e depois retorna para sua condição. Anteriormente, consideramos o comportamento do laço `for` em termos de um laço `while` "equivalente". Contudo, como a instrução `continue` se comporta diferentemente para esses dois laços, não é possível simular perfeitamente um laço `for` com um laço `while` sozinho.

O exemplo a seguir mostra uma instrução `continue` não rotulada sendo usada para pular o restante da iteração atual de um laço quando ocorre um erro:

```
for(i = 0; i < data.length; i++) {
    if (!data[i]) continue;         // Não pode prosseguir com dados indefinidos
    total += data[i];
}
```

Assim como a instrução `break`, a instrução `continue` pode ser usada em sua forma rotulada dentro de laços aninhados, quando o laço a ser reiniciado não é o laço imediatamente circundante. Além disso, assim como na instrução `break`, não são permitidas quebras de linha entre a instrução `continue` e seu *`nomerótulo`*.

5.6.4 return

Lembre-se de que as chamadas de função são expressões e de que todas as expressões têm valores. Uma instrução `return` dentro de uma função especifica o valor das chamadas dessa função. Aqui está a sintaxe da instrução `return`:

```
return expressão;
```

A instrução `return` só pode aparecer dentro do corpo de uma função. É erro de sintaxe ela aparecer em qualquer outro lugar. Quando a instrução `return` é executada, a função que a contém retorna o valor de *`expressão`* para sua chamadora. Por exemplo:

```
function square(x) { return x*x; }      // Uma função que tem instrução return
square(2)                               // Esta chamada é avaliada como 4
```

Sem uma instrução `return`, uma chamada de função simplesmente executa cada uma das instruções do corpo da função até chegar ao fim da função e, então, retorna para sua chamadora. Nesse caso, a expressão de invocação é avaliada como `undefined`. A instrução `return` aparece frequentemente como a última instrução de uma função, mas não precisa ser a última: uma função retorna para sua chamadora quando uma instrução `return` é executada, mesmo que ainda restem outras instruções no corpo da função.

A instrução `return` também pode ser usada sem uma *`expressão`*, para fazer a função retornar `undefined` para sua chamadora. Por exemplo:

```
function display_objeto(o) {
    // Retorna imediatamente se o argumento for null ou undefined.
    if (!o) return;
    // O restante da função fica aqui...
}
```

Por causa da inserção automática de ponto e vírgula em JavaScript (Seção 2.5), não se pode incluir uma quebra de linha entre a palavra-chave `return` e a expressão que a segue.

5.6.5 throw

Uma *exceção* é um sinal indicando que ocorreu algum tipo de condição excepcional ou erro. *Disparar uma exceção* é sinalizar tal erro ou condição excepcional. *Capturar* uma exceção é tratar dela – execu-

tar as ações necessárias ou apropriadas para se recuperar da exceção. Em JavaScript, as exceções são lançadas quando ocorre um erro em tempo de execução e quando o programa lança uma explicitamente, usando a instrução `throw`. As exceções são capturadas com a instrução `try/catch/finally`, a qual está descrita na próxima seção.

A instrução `throw` tem a seguinte sintaxe:

```
throw expressão;
```

`expressão` pode ser avaliada com um valor de qualquer tipo. Pode-se lançar um número representando um código de erro ou uma string contendo uma mensagem de erro legível para seres humanos. A classe Error e suas subclasses são usadas quando o próprio interpretador JavaScript lança um erro, e você também pode usá-las. Um objeto Error tem uma propriedade `name` que especifica o tipo de erro e uma propriedade `message` que contém a string passada para a função construtora (consulte a classe Error na seção de referência). Aqui está um exemplo de função que lança um objeto Error quando chamada com um argumento inválido:

```
function factorial(x) {
    // Se o argumento de entrada é inválido, dispara uma exceção!
    if (x < 0) throw new Error("x must not be negative");
    // Caso contrário, calcula um valor e retorna normalmente
    for(var f = 1; x > 1; f *= x, x--) /* vazio */ ;
    return f;
}
```

Quando uma exceção é disparada, o interpretador JavaScript interrompe imediatamente a execução normal do programa e pula para a rotina de tratamento de exceção mais próxima. As rotinas de tratamento de exceção são escritas usando a cláusula `catch` da instrução `try/catch/finally`, que está descrita na próxima seção. Se o bloco de código no qual a exceção foi lançada não tem uma cláusula `catch` associada, o interpretador verifica o próximo bloco de código circundante mais alto para ver se ele tem uma rotina de tratamento de exceção associada. Isso continua até uma rotina de tratamento ser encontrada. Se uma exceção é lançada em uma função que não contém uma instrução `try/catch/finally` para tratar dela, a exceção se propaga para o código que chamou a função. Desse modo, as exceções se propagam pela estrutura léxica de métodos de JavaScript e para cima na pilha de chamadas. Se nenhuma rotina de tratamento de exceção é encontrada, a exceção é tratada como erro e o usuário é informado.

5.6.6 try/catch/finally

A instrução `try/catch/finally` é o mecanismo de tratamento de exceção de JavaScript. A cláusula `try` dessa instrução simplesmente define o bloco de código cujas exceções devem ser tratadas. O bloco `try` é seguido de uma cláusula `catch`, a qual é um bloco de instruções que são chamadas quando ocorre uma exceção em qualquer lugar dentro do bloco `try`. A cláusula `catch` é seguida por um bloco `finally` contendo o código de limpeza que é garantidamente executado, independente do que aconteça no bloco `try`. Os blocos `catch` e `finally` são opcionais, mas um bloco `try` deve estar acompanhado de pelo menos um desses blocos. Os blocos `try`, `catch` e `finally` começam e terminam com chaves. Essas chaves são uma parte obrigatória da sintaxe e não podem ser omitidas, mesmo que uma cláusula contenha apenas uma instrução.

O código a seguir ilustra a sintaxe e o objetivo da instrução try/catch/finally:

```
try {
  // Normalmente, este código é executado do início ao fim do bloco
  // sem problemas. Mas às vezes pode disparar uma exceção
  // diretamente, com uma instrução throw, ou indiretamente, pela
  // chamada de um método que lança uma exceção.
}
catch (e) {
  // As instruções deste bloco são executadas se, e somente se, o bloco
  // try dispara uma exceção. Essas instruções podem usar a variável local
  // e se referir ao objeto Error ou a outro valor que foi lançado.
  // Este bloco pode tratar da exceção de algum modo, pode ignorá-la
  // não fazendo nada ou pode lançar a exceção novamente com throw.
}
finally {
  // Este bloco contém instruções que são sempre executadas, independente
  // do que aconteça no bloco try. Elas são executadas se o bloco
  // try terminar:
  //    1) normalmente, após chegar ao final do bloco
  //    2) por causa de uma instrução break, continue ou return
  //    3) com uma exceção que é tratada por uma cláusula catch anterior
  //    4) com uma exceção não capturada que ainda está se propagando
}
```

Note que a palavra-chave catch é seguida por um identificador entre parênteses. Esse identificador é como um parâmetro de função. Quando uma exceção é capturada, o valor associado à exceção (um objeto Error, por exemplo) é atribuído a esse parâmetro. Ao contrário das variáveis normais, o identificador associado a uma cláusula catch tem escopo de bloco – ele é definido apenas dentro do bloco catch.

Aqui está um exemplo realista da instrução try/catch. Ele usa o método factorial() definido na seção anterior e os métodos JavaScript do lado do cliente prompt() e alert() para entrada e saída:

```
try {
    // Pede para o usuário inserir um número
    var n = Number(prompt("Please enter a positive integer", ""));
    // Calcula o fatorial do número, supondo que a entrada seja válida
    var f = factorial(n);
    // Mostra o resultado
    alert(n + "! = " + f);
}
catch (ex) {          // Se a entrada do usuário não foi válida, terminamos aqui
    alert(ex);        // Informa ao usuário qual é o erro
}
```

Esse exemplo é uma instrução try/catch sem qualquer cláusula finally. Embora finally não seja usada tão frequentemente quanto catch, ela pode ser útil. Contudo, seu comportamento exige mais explicações. É garantido que a cláusula finally é executada se qualquer parte do bloco try é executada, independente de como o código do bloco try termina. Ela é geralmente usada para fazer a limpeza após o código na cláusula try.

No caso normal, o interpretador JavaScript chega ao final do bloco `try` e então passa para o bloco `finally`, o qual faz toda limpeza necessária. Se o interpretador sai do bloco `try` por causa de uma instrução `return`, `continue` ou `break`, o bloco `finally` é executado antes que o interpretador pule para seu novo destino.

Se ocorre uma exceção no bloco `try` e existe um bloco `catch` associado para tratar da exceção, o interpretador primeiramente executa o bloco `catch` e depois o bloco `finally`. Se não há qualquer bloco `catch` local para tratar da exceção, o interpretador primeiramente executa o bloco `finally` e depois pula para a cláusula `catch` circundante mais próxima.

Se o próprio bloco `finally` causa um salto com uma instrução `return`, `continue`, `break` ou `throw`, ou chamando um método que lança uma exceção, o interpretador abandona o salto que estava pendente e realiza o novo salto. Por exemplo, se uma cláusula `finally` lança uma exceção, essa exceção substitui qualquer outra que estava no processo de ser lançada. Se uma cláusula `finally` executa uma instrução `return`, o método retorna normalmente, mesmo que uma exceção tenha sido lançada e ainda não tratada.

`try` e `finally` podem ser usadas juntas, sem uma cláusula `catch`. Nesse caso, o bloco `finally` é simplesmente código de limpeza que garantidamente é executado, independente do que aconteça no bloco `try`. Lembre-se de que não podemos simular completamente um laço `for` com um laço `while`, pois a instrução `continue` se comporta diferentemente para os dois laços. Se adicionamos uma instrução `try/finally`, podemos escrever um loop `while` que funciona como um laço `for` e que trata instruções `continue` corretamente:

```
// Simula o corpo de for( inicialização ; teste ; incremento );
inicialização ;
while( teste ) {
    try { corpo ; }
    finally { incremento ; }
}
```

Note, entretanto, que um *`corpo`* que contém uma instrução `break` se comporta de modo ligeiramente diferente (causando um incremento extra antes de sair) no laço `while` e no laço `for`; portanto, mesmo com a cláusula `finally`, não é possível simular completamente o laço `for` com `while`.

5.7 Instruções diversas

Esta seção descreve as três instruções restantes de JavaScript – `with`, `debugger` e `use strict`.

5.7.1 with

Na Seção 3.10.3, discutimos o encadeamento de escopo – uma lista de objetos que são pesquisados, em ordem, para realizar a solução de nomes de variável. A instrução `with` é usada para ampliar o encadeamento de escopo temporariamente. Ela tem a seguinte sintaxe:

```
with (objeto)
instrução
```

Essa instrução adiciona *objeto* na frente do encadeamento de escopo, executa *instrução* e, então, restaura o encadeamento de escopo ao seu estado original.

A instrução `with` é proibida no modo restrito (consulte a Seção 5.7.3) e deve ser considerada desaprovada no modo não restrito: evite usá-la, quando possível. Um código JavaScript que utiliza `with` é difícil de otimizar e é provável que seja executado mais lentamente do que um código equivalente escrito sem a instrução `with`.

O uso comum da instrução `with` é para facilitar o trabalho com hierarquias de objeto profundamente aninhadas. Em JavaScript do lado do cliente, por exemplo, talvez seja necessário digitar expressões como a seguinte para acessar elementos de um formulário HTML:

```
document.forms[0].address.value
```

Caso precise escrever expressões como essa várias vezes, você pode usar a instrução `with` para adicionar o objeto formulário no encadeamento de escopo:

```
with(document.forms[0]) {
    // Acessa elementos do formulário diretamente aqui. Por exemplo:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

Isso reduz o volume de digitação necessária: não é mais preciso prefixar cada nome de propriedade do formulário com `document.forms[0]`. Esse objeto faz parte do encadeamento de escopo temporariamente e é pesquisado automaticamente quando JavaScript precisa solucionar um identificador, como `address`. É claro que é muito simples evitar a instrução `with` e escrever o código anterior como segue:

```
var f = document.forms[0];
f.name.value = "";
f.address.value = "";
f.email.value = "";
```

Lembre-se de que o encadeamento de escopo é usado somente ao se pesquisar identificadores e não ao se criar outros novos. Considere este código:

```
with(o) x = 1;
```

Se o objeto `o` tem uma propriedade `x`, então esse código atribui o valor `1` a essa propriedade. Mas se `x` não está definida em `o`, esse código é o mesmo que `x = 1` sem a instrução `with`. Ele atribui a uma variável local ou global chamada `x` ou cria uma nova propriedade do objeto global. Uma instrução `with` fornece um atalho para ler propriedades de `o`, mas não para criar novas propriedades de `o`.

5.7.2 debugger

A instrução `debugger` normalmente não faz nada. No entanto, se um programa depurador estiver disponível e em execução, então uma implementação pode (mas não é obrigada a) executar algum tipo de ação de depuração. Na prática, essa instrução atua como um ponto de interrupção: a execução do

código JavaScript para e você pode usar o depurador para imprimir valores de variáveis, examinar a pilha de chamada, etc. Suponha, por exemplo, que você esteja obtendo uma exceção em sua função `f()` porque ela está sendo chamada com um argumento indefinido e você não consegue descobrir de onde essa chamada está vindo. Para ajudar na depuração desse problema, você poderia alterar `f()` de modo que começasse como segue:

```
function f(o) {
  if (o === undefined) debugger;     // Linha temporária para propósitos de depuração
  ...                                // O restante da função fica aqui.
}
```

Agora, quando `f()` for chamada sem argumentos, a execução vai parar e você poderá usar o depurador para inspecionar a pilha de chamada e descobrir de onde está vindo essa chamada incorreta.

`debugger` foi adicionada formalmente na linguagem por ECMAScript 5, mas tem sido implementada pelos principais fornecedores de navegador há bastante tempo. Note que não é suficiente ter um depurador disponível: a instrução `debugger` não vai iniciar o depurador para você. No entanto, se um depurador já estiver em execução, essa instrução vai causar um ponto de interrupção. Se você usa a extensão de depuração Firebug para Firefox, por exemplo, deve ter o Firebug habilitado para a página Web que deseja depurar para que a instrução `debugger` funcione.

5.7.3 "use strict"

`"use strict"` é uma *diretiva* introduzida em ECMAScript 5. As diretivas não são instruções (mas são parecidas o suficiente para que `"use strict"` seja documentada aqui). Existem duas diferenças importantes entre a diretiva `"use strict"` e as instruções normais:

- Ela não inclui qualquer palavra-chave da linguagem: a diretiva é apenas uma instrução de expressão que consiste em uma string literal especial (entre aspas simples ou duplas). Os interpretadores JavaScript que não implementam ECMAScript 5 vão ver simplesmente uma instrução de expressão sem efeitos colaterais e não farão nada. É esperado que as futuras versões do padrão ECMAScript apresentem `use` como uma verdadeira palavra-chave, permitindo que as aspas sejam eliminadas.
- Ela só pode aparecer no início de um script ou no início do corpo de uma função, antes que qualquer instrução real tenha aparecido. Contudo, não precisa ser o primeiro item no script ou na função: uma diretiva `"use strict"` pode ser seguida ou precedida por outras instruções de expressão de string literal, sendo que as implementações de JavaScript podem interpretar essas outras strings literais como diretivas definidas pela implementação. As instruções de expressão de string literal que vêm depois da primeira instrução normal em um script ou em uma função são apenas instruções de expressão normais; elas não podem ser interpretadas como diretivas e não têm efeito algum.

O objetivo de uma diretiva `"use strict"` é indicar que o código seguinte (no script ou função) é *código restrito*. O código de nível superior (não função) de um script é código restrito se o script tem uma diretiva `"use strict"`. O corpo de uma função é código restrito se está definido dentro de código restrito ou se tem uma diretiva `"use strict"`. Um código passado para o método `eval()` é código restrito se `eval()` é chamado a partir de código restrito ou se a string de código inclui uma diretiva `"use strict"`.

Um código restrito é executado no *modo restrito*. O modo restrito de ECMAScript 5 é um subconjunto restrito da linguagem que corrige algumas deficiências importantes e fornece verificação de erro mais forte e mais segurança. As diferenças entre o modo restrito e o modo não restrito são as seguintes (as três primeiras são especialmente importantes):

- A instrução `with` não é permitida no modo restrito.
- No modo restrito, todas as variáveis devem ser declaradas: um ReferenceError é lançado se você atribui um valor a um identificador que não é uma variável, função, parâmetro de função, parâmetro de cláusula `catch` ou propriedade declarada do objeto global. (No modo não restrito, isso declara uma variável global implicitamente, pela adição de uma nova propriedade no objeto global.)
- No modo restrito, as funções chamadas como funções (e não como métodos) têm o valor de `this` igual a `undefined`. (No modo não restrito, as funções chamadas como funções são sempre passadas para o objeto global como seu valor de `this`.) Essa diferença pode ser usada para determinar se uma implementação suporta o modo restrito:

  ```
  var hasStrictMode = (function() { "use strict"; return this===undefined}());
  ```

  Além disso, no modo restrito, quando uma função é chamada com `call()` ou `apply()`, o valor de `this` é exatamente o valor passado como primeiro argumento para `call()` ou `apply()`. (No modo não restrito, valores `null` e `undefined` são substituídos pelo objeto global e valores que não são objeto são convertidos em objetos.)
- No modo restrito, as atribuições para propriedades não graváveis e tentativas de criar novas propriedades em objetos não extensíveis lançam um TypeError. (No modo não restrito, essas tentativas falham silenciosamente.)
- No modo restrito, um código passado para `eval()` não pode declarar variáveis nem definir funções no escopo do chamador, como acontece no modo não restrito. Em vez disso, as definições de variável e de função têm um novo escopo criado para `eval()`. Esse escopo é descartado quando `eval()` retorna.
- No modo restrito, o objeto `arguments` (Seção 8.3.2) em uma função contém uma cópia estática dos valores passados para a função. No modo não restrito, o objeto `arguments` tem comportamento "mágico", no qual os elementos do array e os parâmetros de função nomeados se referem ambos ao mesmo valor.
- No modo restrito, um SyntaxError é lançada se o operador `delete` é seguido por um identificador não qualificado, como uma variável, função ou parâmetro de função. (No modo não restrito, tal expressão `delete` não faz nada e é avaliada como `false`.)
- No modo restrito, uma tentativa de excluir uma propriedade que não pode ser configurada lança um TypeError. (No modo não restrito, a tentativa falha e a expressão `delete` é avaliada como `false`.)
- No modo restrito, é erro de sintaxe um objeto literal definir duas ou mais propriedades com o mesmo nome. (No modo não restrito, não ocorre erro.)
- No modo restrito, é erro de sintaxe uma declaração de função ter dois ou mais parâmetros com o mesmo nome. (No modo não restrito, não ocorre erro.)

- No modo restrito, literais inteiros em octal (começando com um 0 que não é seguido por um x) não são permitidas. (No modo não restrito, algumas implementações permitem literais em octal.)
- No modo restrito, os identificadores `eval` e `arguments` são tratados como palavras-chave e não é permitido alterar seus valores. Você pode atribuir um valor a esses identificadores, declará-los como variáveis, utilizá-los como nomes de função, utilizá-los como nomes de parâmetro de função ou utilizá-los como o identificador de um bloco `catch`.
- No modo restrito, a capacidade de examinar a pilha de chamada é restrita. `arguments.caller` e `arguments.callee` lançam ambos um TypeError dentro de uma função de modo restrito. As funções de modo restrito também têm propriedades `caller` e `arguments` que lançam um TypeError quando lidas. (Algumas implementações definem essas propriedades não padronizadas em funções não restritas.)

5.8 Resumo das instruções JavaScript

Este capítulo apresentou cada uma das instruções da linguagem JavaScript. A Tabela 5-1 as resume, listando a sintaxe e o objetivo de cada uma.

Tabela 5-1 Sintaxe das instruções JavaScript

Instrução	Sintaxe	Objetivo
`break`	`break [`*`rótulo`*`];`	Sai do laço ou `switch` mais interno ou da instrução circundante nomeada
`case`	`case` *`expressão`*`:`	Rotula uma instrução dentro de um `switch`
`continue`	`continue [`*`rótulo`*`];`	Começa a próxima iteração do laço mais interno ou do laço nomeado
`debugger`	`debugger;`	Ponto de interrupção de depurador
`default`	`default:`	Rotula a instrução padrão dentro de um `switch`
`do/while`	`do` *`instrução`* `while (`*`expressão`*`);`	Uma alternativa para o laço `while`
`empty`	`;`	Não faz nada
`for`	`for(`*`inic`*`;` *`teste`*`;` *`incr`*`)` *`instrução`*	Um laço fácil de usar
`for/in`	`for (`*`var`* `in` *`objeto`*`)` *`instrução`*	Enumera as propriedades de *objeto*
`function`	`function` *`nome`*`([`*`parâm`*`[,...]]) {` *`corpo`* `}`	Declara uma função chamada *nome*
`if/else`	`if (`*`expr`*`)` *`instrução1`* `[else` *`instrução2`*`]`	Executa *instrução1* ou *instrução2*
`label`	*`rótulo`*`:` *`instrução`*	Dá à *instrução* o nome *rótulo*
`return`	`return [`*`expressão`*`];`	Retorna um valor de uma função

Tabela 5-1 Sintaxe das instruções JavaScript *(Continuação)*

Instrução	Sintaxe	Objetivo
switch	switch (*expressão*) { *instruções* }	Ramificação de múltiplos caminhos para rótulos case ou default:
throw	throw *expressão*;	Lança uma exceção
try	try { *instruções* }	Trata exceções
	[catch { *instruções de rotina de tratamento* }]	
	[finally { *instruções de limpeza* }]	
use strict	"use strict";	Aplica restrições do modo restrito em um script ou função
var	var *nome* [= *expr*] [,...];	Declara e inicializa uma ou mais variáveis
while	while (*expressão*) *instrução*	Uma construção de laço básica
with	with (*objeto*) *instrução*	Amplia o encadeamento de escopo (proibida no modo restrito)

Capítulo 6

Objetos

O tipo fundamental de dados de JavaScript é o *objeto*. Um objeto é um valor composto: ele agrega diversos valores (valores primitivos ou outros objetos) e permite armazenar e recuperar esses valores pelo nome. Um objeto é um conjunto não ordenado de *propriedades*, cada uma das quais tendo um nome e um valor. Os nomes de propriedade são strings; portanto, podemos dizer que os objetos mapeiam strings em valores. Esse mapeamento de string em valor recebe vários nomes: você provavelmente já conhece a estrutura de dados fundamental pelo nome "hash", "tabela de hash", "dicionário" ou "array associativo". Contudo, um objeto é mais do que um simples mapeamento de strings para valores. Além de manter seu próprio conjunto de propriedades, um objeto JavaScript também herda as propriedades de outro objeto, conhecido como seu "protótipo". Os métodos de um objeto normalmente são propriedades herdadas e essa "herança de protótipos" é um recurso importante de JavaScript.

Os objetos JavaScript são dinâmicos – normalmente propriedades podem ser adicionadas e excluídas –, mas podem ser usados para simular os objetos e as "estruturas" estáticas das linguagens estaticamente tipadas. Também podem ser usados (ignorando-se a parte referente ao valor do mapeamento de string para valor) para representar conjuntos de strings.

Qualquer valor em JavaScript que não seja uma string, um número, `true`, `false`, `null` ou `undefined`, é um objeto. E mesmo que strings, números e valores booleanos não sejam objetos, eles se comportam como objetos imutáveis (consulte a Seção 3.6).

Lembre-se, da Seção 3.7, de que os objetos são *mutáveis* e são manipulados por referência e não por valor. Se a variável `x` se refere a um objeto e o código `var y = x;` é executado, a variável `y` contém uma referência para o mesmo objeto e não uma cópia desse objeto. Qualquer modificação feita no objeto por meio da variável `y` também é visível por meio da variável `x`.

As coisas mais comuns feitas com objetos são: criá-los e configurar, consultar, excluir, testar e enumerar suas propriedades. Essas operações fundamentais estão descritas nas seções de abertura deste capítulo. As seções seguintes abordam temas mais avançados, muitos dos quais são específicos de ECMAScript 5.

Uma *propriedade* tem um nome e um valor. Um nome de propriedade pode ser qualquer string, incluindo a string vazia, mas nenhum objeto pode ter duas propriedades com o mesmo nome. O valor pode ser qualquer valor de JavaScript ou (em ECMAScript 5) uma função "getter" ou "setter"