Unearthing the Excellence in JavaScript

JavaScript: The Good Parts

O'REILLY® | YAHOO! PRESS

Douglas Crockford

JavaScript: The Good Parts

Capítulo 1. Good Parts

Cuando yo era un joven programador, aprendía sobre cada característica de los idiomas que estaba usando, e intentaría usar todas esas características cuando escribiera. Supongo que era una forma de mostrarse, y supongo que funcionó porque yo era el tipo al que acudías si querías saber cómo usar una característica en particular.

Eventualmente me di cuenta de que algunas de esas características eran más problemas de lo que valían. Algunos de ellos estaban mal especificados, y por lo tanto eran más propensos a causar problemas de portabilidad. Algunos resultaron en código que era difícil de leer o modificar. Algunos me indujeron a escribir de una manera que era demasiado complicada y propensa a errores. Y algunas de esas características eran errores de diseño. A veces, los diseñadores de lenguaje cometen errores.

La mayoría de los lenguajes de programación contienen partes buenas y partes defectuosas. Descubrí que podría ser un mejor programador utilizando sólo las partes buenas y evitar las partes malas. Después de todo, ¿cómo se puede construir algo bueno de las partes malas?

Rara vez es posible que los comités de normalización eliminen las imperfecciones de un idioma porque hacerlo podría causar la ruptura de todos los malos programas que dependen de esas partes malas. Por lo general son impotentes para hacer cualquier cosa, excepto más características en la parte superior de la pila de imperfecciones existentes. Y las nuevas características no siempre interactúan armoniosamente, produciendo así más partes malas.

Pero usted tiene el poder de definir su propio subconjunto. Usted puede escribir mejores programas confiando exclusivamente en las partes buenas.

Afortunadamente, JavaScript tiene algunas partes extraordinariamente buenas. En JavaScript, hay un lenguaje hermoso, elegante, altamente expresivo, que está enterrado bajo una pila humeante de buenas intenciones y errores. La mejor naturaleza de JavaScript es tan ocultada que durante muchos años la opinión predominante de JavaScript fue que era un juguete antiestético e incompetente. Mi intención aquí es exponer la bondad en JavaScript, un excelente lenguaje de programación dinámico. JavaScript es un bloque de mármol, y me despojo de las características que no son hermosas hasta que la verdadera naturaleza del lenguaje se revela. Creo que el elegante subconjunto que he tallado es muy superior al lenguaje en su conjunto, siendo más confiable, legible y mantenible.

Este libro no intentará describir completamente el idioma. En su lugar, se centrará en las partes buenas con advertencias ocasionales para evitar lo malo. El subconjunto que se describirá aquí se puede utilizar para construir programas confiables y legibles, pequeños y grandes. Al centrarnos sólo en las partes buenas, podemos reducir el tiempo de aprendizaje, aumentar la robustez y ahorrar algunos árboles.

Tal vez el mayor beneficio de estudiar las partes buenas es que se puede evitar la necesidad de desaprender las partes malas. Desaprender malos patrones es muy difícil. Es una tarea dolorosa que la mayoría de nosotros enfrentamos con extrema renuencia. A veces, los idiomas están subdivididos para que funcionen mejor para los estudiantes. Pero en este caso, estoy subdividiendo JavaScript para que funcione mejor para los profesionales.

¿Por qué JavaScript?

JavaScript es un idioma importante porque es el idioma del navegador web. Su asociación con el navegador lo convierte en uno de los lenguajes de programación más populares del mundo. Al mismo tiempo, es uno de los lenguajes de programación más despreciados del mundo. El API del navegador, el DOM (Document Object Model) es bastante terrible, y JavaScript es injustamente

culpado. Con DOM sería doloroso trabajar con él en cualquier lengua. El DOM está mal especificado e implementado de manera inconsistente. Este libro sólo toca muy ligeramente en los DOM. Creo que escribir un buen libro de piezas sobre el DOM sería muy difícil.

JavaScript es más despreciado porque no es algún otro idioma. Si eres bueno en algún otro idioma y tienes que programar en un entorno que sólo es compatible con JavaScript, entonces estás obligado a usar JavaScript, y eso es molesto. La mayoría de las personas en esa situación ni siquiera se molestan en aprender JavaScript primero, y luego se sorprenden cuando JavaScript resulta tener diferencias significativas con el otro idioma que preferirían usar, y que esas diferencias importan.

La cosa asombrosa sobre el Javascript es que es posible conseguir el trabajo hecho con él sin saber mucho sobre el lenguaje, o incluso saber mucho sobre la programación. Es un lenguaje con enorme poder expresivo. Es aún mejor cuando sabes lo que estás haciendo. La programación es un negocio difícil. Nunca debe emprenderse en la ignorancia.

Analizando JavaScript

JavaScript se basa en algunas muy buenas ideas y algunas muy malas.

Las ideas muy buenas incluyen funciones, mecanografía suelta, objetos dinámicos, y una notación expresiva del objeto literal. Las malas ideas incluyen un modelo de programación basado en variables globales.

Las funciones de JavaScript son objetos de primera clase con (principalmente) el ámbito léxico. JavaScript es el primer lenguaje lambda que va a la corriente principal. En el fondo, JavaScript tiene más en común con Lisp y Scheme que con Java. Es Lisp en la ropa de C. Esto hace que JavaScript sea un lenguaje notablemente potente.

La moda en la mayoría de los lenguajes de programación hoy en día exige una fuerte mecanografía. La teoría es que el mecanografiar fuerte permite que un compilador detecte una clase grande de errores en tiempo de la compilación. Cuanto antes podamos detectar y reparar errores, menos nos cuestan. JavaScript es un lenguaje suelto, por lo que los compiladores JavaScript no pueden detectar errores de tipo. Esto puede ser alarmante para las personas que están llegando a JavaScript de los idiomas fuertemente mecanografiados. Pero resulta que la tipificación fuerte no elimina la necesidad de una cuidadosa prueba. Y he encontrado en mi trabajo que los tipos de errores que la comprobación de tipo fuerte encuentra no son los errores que me preocupan. Por otro lado, considero que la mecanografía suelta es liberadora. No necesito formar jerarquías de clases complejas. Y nunca tengo que lanzar o luchar con el sistema de tipo para obtener el comportamiento que quiero.

JavaScript tiene una notación literal muy potente. Los objetos se pueden crear simplemente enumerando sus componentes. Esta notación fue la inspiración para JSON, el popular formato de intercambio de datos. (Habrá más información sobre JSON en el Apéndice E.)

Una característica controvertida en JavaScript es la herencia **prototypal**. JavaScript tiene un sistema de objetos sin clases en el que los objetos heredan propiedades directamente de otros objetos. Esto es realmente poderoso, pero no es familiar para los programadores entrenados clásicamente. Si intenta aplicar patrones de diseño clásicos directamente a JavaScript, se sentirá frustrado. Pero si aprende a trabajar con la naturaleza prototípica de JavaScript, sus esfuerzos serán recompensados.

JavaScript es muy difamado por su elección de ideas clave. En su mayor parte, sin embargo, esas opciones eran buenas, si fuera inusual. Pero había una opción que era particularmente mala: JavaScript depende de las variables globales para la vinculación. Todas las variables de nivel superior de todas las unidades de compilación se lanzan juntas en un espacio de nombres común denominado objeto global. Esto es malo porque las variables globales son malas, y en JavaScript

son fundamentales. Afortunadamente, como veremos, JavaScript también nos da las herramientas para mitigar este problema.

En algunos casos, no podemos ignorar las partes malas. Hay algunas partes inevitables horribles, que serán llamados a medida que ocurren. También se resumirán en el Apéndice A. Pero tendremos éxito en evitar la mayoría de las partes malas de este libro, resumiendo gran parte de lo que quedó fuera en el Apéndice B. Si quieres saber más sobre las partes malas y cómo usarlas Mal, consulte cualquier otro libro JavaScript.

El estándar que define JavaScript (aka JScript) es la tercera edición de The ECMAScript Programming Language, que está disponible en http://www.ecmainternational.org/publications/files/ecma-st/ECMA-262.pdf. El lenguaje descrito en este libro es un subconjunto adecuado de ECMAScript. Este libro no describe el idioma entero porque deja fuera las partes malas. El tratamiento aquí no es exhaustivo. Evita los casos de borde. Tú también deberías. Hay peligro y miseria en los bordes.

El Apéndice C describe una herramienta de programación llamada JSLint, un analizador de JavaScript que puede analizar un programa JavaScript e informar sobre las partes defectuosas que contiene. JSLint proporciona un grado de rigor que generalmente carece de desarrollo de JavaScript. Puede darle confianza de que sus programas contienen solamente las partes buenas.

JavaScript es un lenguaje de muchos contrastes. Contiene muchos errores y bordes afilados, por lo que podría preguntarse: "¿Por qué debo usar JavaScript?" Hay dos respuestas. La primera es que no tienes elección. La Web se ha convertido en una plataforma importante para el desarrollo de aplicaciones, y JavaScript es el único idioma que se encuentra en todos los navegadores. Es lamentable que Java falló en ese entorno; Si no lo hubiera hecho, podría haber una opción para las personas que desean un lenguaje clásico fuertemente mecanografiado. Pero Java falló y JavaScript está floreciendo, por lo que hay evidencia de que JavaScript hizo algo bien.

La otra respuesta es que, a pesar de sus deficiencias, JavaScript es realmente bueno. Es ligero y expresivo. Y una vez que te quedes al tanto, la programación funcional es muy divertida.

Pero para usar bien el lenguaje, debes estar bien informado sobre sus limitaciones. Voy a golpear a los que con cierta brutalidad. No dejes que eso te desanime. Las partes buenas son lo suficientemente buenas para compensar las partes malas.

Una tierra de prueba simple
Si tienes un navegador web y cualquier editor de texto, tienes todo lo que necesitas para ejecutar programas JavaScript. En primer lugar, crear un archivo HTML con un nombre como program.html:

```
<html><body><pre><script src="program.js">
</script></pre></body></html>
```

A continuación, cree un archivo en el mismo directorio con un nombre como program.js:

```
document.writeln('Hello, world!');
```

A continuación, abra su archivo HTML en su navegador para ver el resultado. A lo largo del libro, se utiliza un método de método para definir nuevos métodos. Esta es su definición:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

Se explicará en el capítulo 4.

Capítulo 2. Gramática

Este capítulo introduce la gramática de las partes buenas de JavaScript, presentando una visión general rápida de cómo está estructurado el lenguaje. Representaremos la gramática con diagramas de ferrocarril.

Las reglas para interpretar estos diagramas son simples:

- Comience en el borde izquierdo y siga las pistas hasta el borde derecho.
- A medida que vaya, encontrará literales en óvalos, y reglas o descripciones en rectángulos.
- Cualquier secuencia que se pueda realizar siguiendo las pistas es legal.
- Cualquier secuencia que no pueda realizarse siguiendo las pistas no es legal.
- Los diagramas de ferrocarril con una barra en cada extremo permiten que se inserte espacio en blanco entre cualquier par de fichas. Los diagramas de ferrocarril con dos barras en cada extremo no lo hacen.

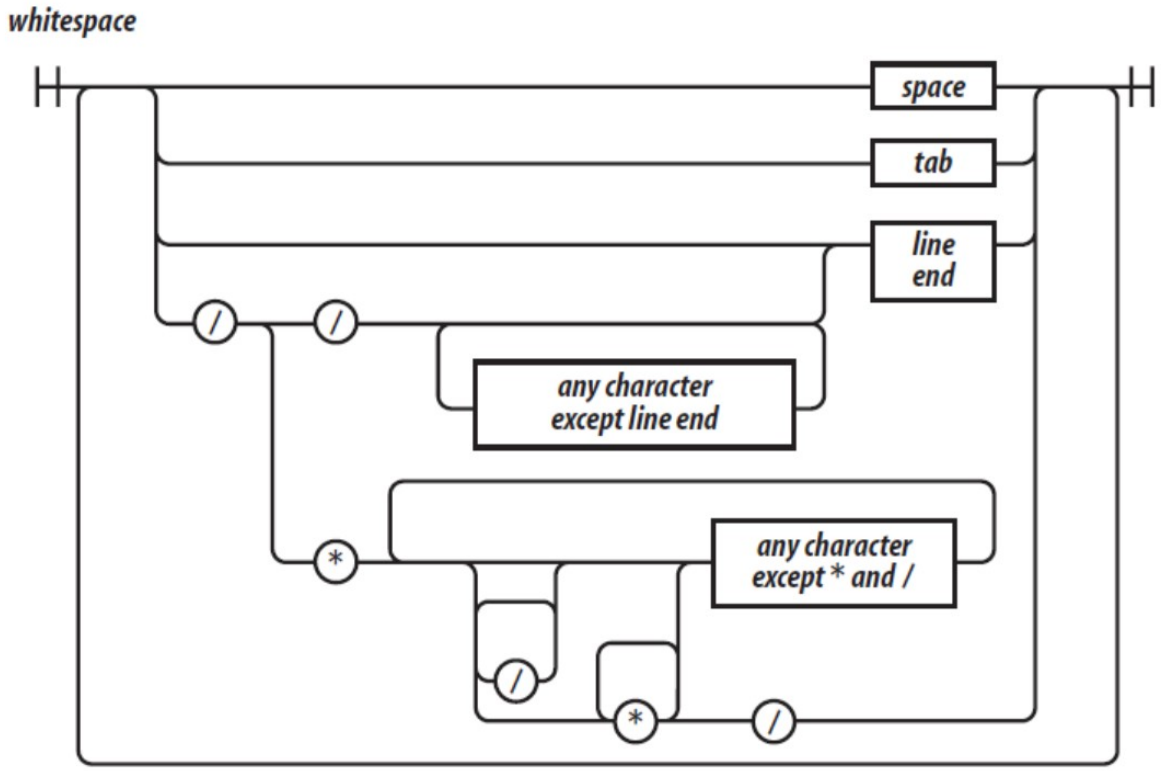
La gramática de las partes buenas presentadas en este capítulo es significativamente más simple que la gramática de todo el lenguaje.

Espacio en blanco

Los espacios en blanco pueden tomar la forma de caracteres de formato o comentarios. Los espacios en blanco suelen ser insignificantes, pero ocasionalmente es necesario usar espacios en blanco para separar secuencias de caracteres que de otro modo se combinarían en un solo token. Por ejemplo, en:

```
var that = this;
```

El espacio entre var y that no se puede quitar, pero los otros espacios se pueden quitar.



JavaScript ofrece dos formas de comentarios, bloques de comentarios formados con / * * / y los comentarios de finalización de la línea comenzando por //. Los comentarios deben ser usados liberalmente para mejorar la legibilidad de sus programas. Tenga cuidado de que los comentarios siempre describan con precisión el código. Comentarios obsoletos son peores que ningún comentario.

La forma / * * / de los comentarios de bloque procedía de un lenguaje llamado PL/I. PL/I escogió esos pares extraños como los símbolos para los comentarios porque era improbable que ocurriera en los programas de ese idioma, excepto tal vez en literales de cuerdas. En JavaScript, esos pares también pueden aparecer en literales de expresiones regulares, por lo que los comentarios de bloque no son seguros para comentar bloques de código. Por ejemplo:

```
/*
    var rm_a = /a*/.match(s);
*/
```

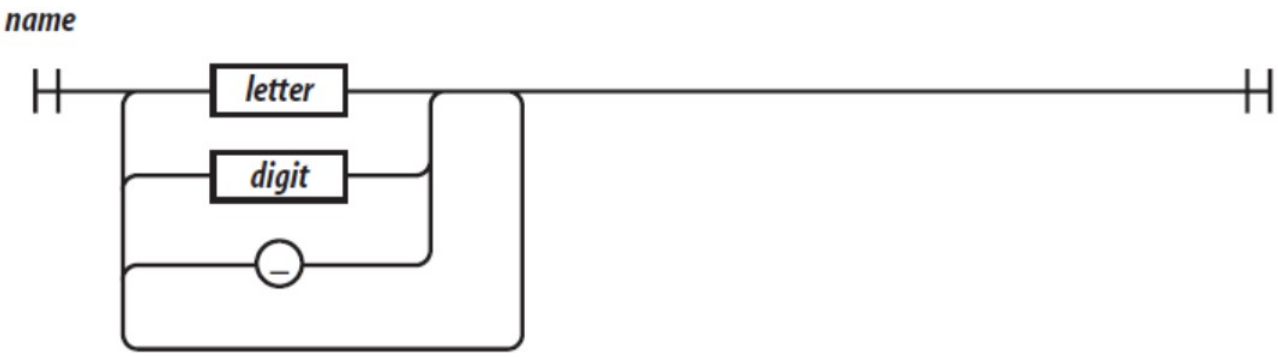
Causa un error de sintaxis. Por lo tanto, se recomienda evitar / * * / comentarios y // se deben usar los comentarios en su lugar. En este libro, // se utilizará exclusivamente.

Nombres

Un nombre es una letra opcionalmente seguida de una o más letras, dígitos o barras inferiores. Un nombre no puede ser una de estas palabras reservadas:

```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
```

```
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
```

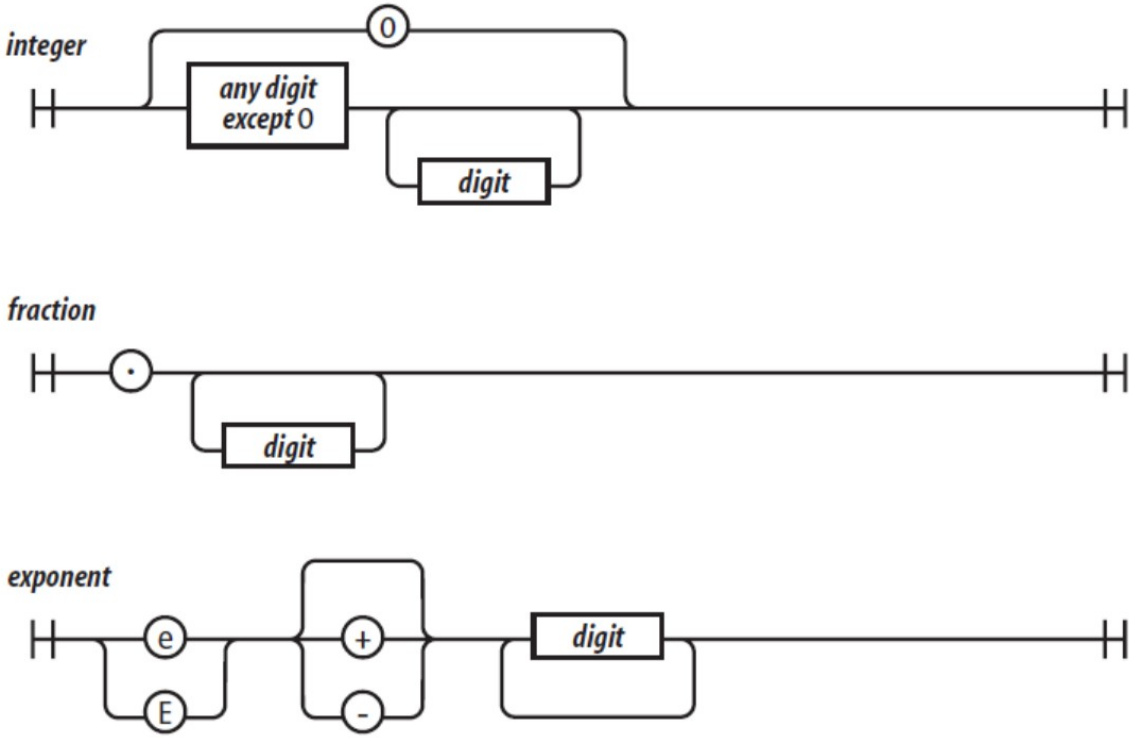


La mayoría de las palabras reservadas en esta lista no se utilizan en el idioma. La lista no incluye algunas palabras que deberían haber sido reservadas pero que no fueron, como **undefined**, **NaN** e **Infinity**. No se permite nombrar una variable o un parámetro con una palabra reservada. Peor aún, no se permite usar una palabra reservada como el nombre de una propiedad de objeto en un literal de objeto o después de un punto en un refinamiento.

Los nombres se utilizan para declaraciones, variables, parámetros, nombres de propiedad, operadores y etiquetas.

Números

JavaScript tiene un solo tipo de número. Internamente, se representa como punto flotante de 64 bits, igual que el doble de Java. A diferencia de la mayoría de los otros lenguajes de programación, no hay ningún tipo entero separado, por lo que 1 y 1,0 son el mismo valor. Esta es una conveniencia significativa porque los problemas de desbordamiento en enteros cortos se evitan por completo, y todo lo que necesita saber acerca de un número es que es un número. Se evita una clase grande de errores de tipo numérico.



Si un literal de número tiene una parte de exponente, entonces el valor del literal se calcula multiplicando la parte antes de e por 10 elevada a la potencia de la parte después de la e. Así que 100 y 1e2 son el mismo número.

Los números negativos pueden formarse usando el operador - prefix.

El valor NaN es un valor numérico que es el resultado de una operación que no puede producir un resultado normal. NaN no es igual a ningún valor, incluyendo a sí mismo. Puede detectar NaN con la función isNaN (número).

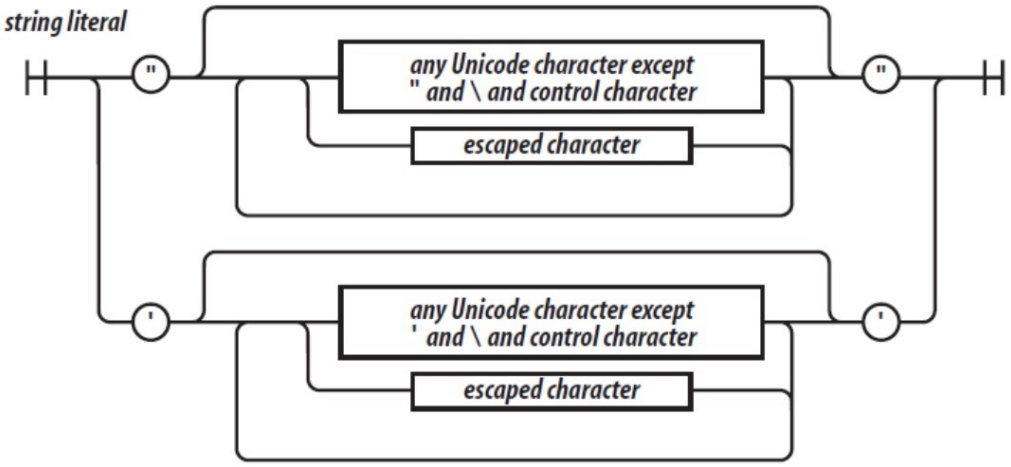
El valor **Infinity** representa todos los valores mayores que 1.79769313486231570e + 308.

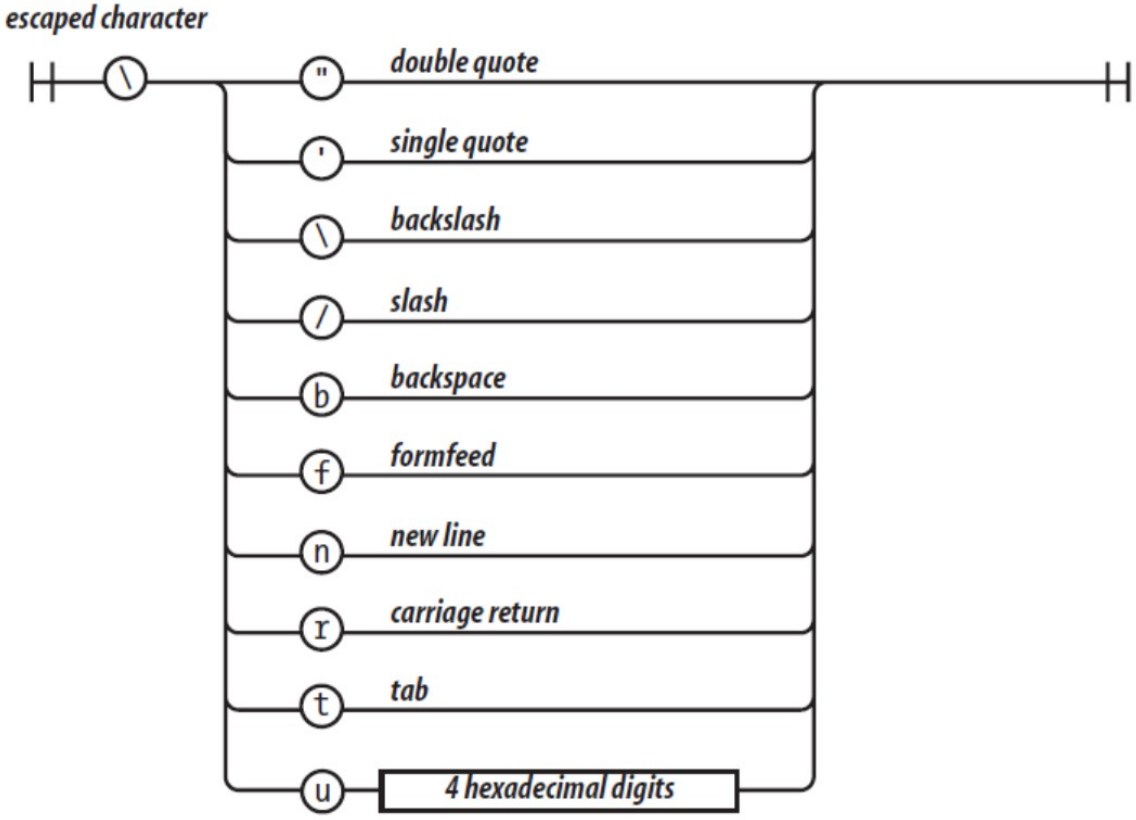
Los números tienen métodos (véase el capítulo 8). JavaScript tiene un objeto Math que contiene un conjunto de métodos que actúan sobre los números. Por ejemplo, el método Math.floor (número) se puede utilizar para convertir un número en un entero.

String

Un literal de cadena puede ser envuelto en comillas simples o comillas dobles. Puede contener cero o más caracteres. La \ (barra invertida) es el carácter de escape. JavaScript se creó en un momento en que Unicode era un juego de caracteres de 16 bits, por lo que todos los caracteres en JavaScript tienen 16 bits de ancho.

JavaScript no tiene un tipo de carácter. Para representar un carácter, haga una cadena con sólo un carácter.





Las secuencias de escape permiten insertar caracteres en cadenas que normalmente no están permitidas, como barras invertidas, comillas y caracteres de control. La convención \u permite especificar numéricamente los puntos del código de caracteres.

```
"A" === "\u0041"
```

Las cadenas tienen una propiedad length. Por ejemplo, "siete" .length es 5.

Las cadenas son inmutables. Una vez que se hace, una cadena nunca se puede cambiar. Pero es fácil hacer una nueva cadena concatenando otras cadenas junto con el operador (+).

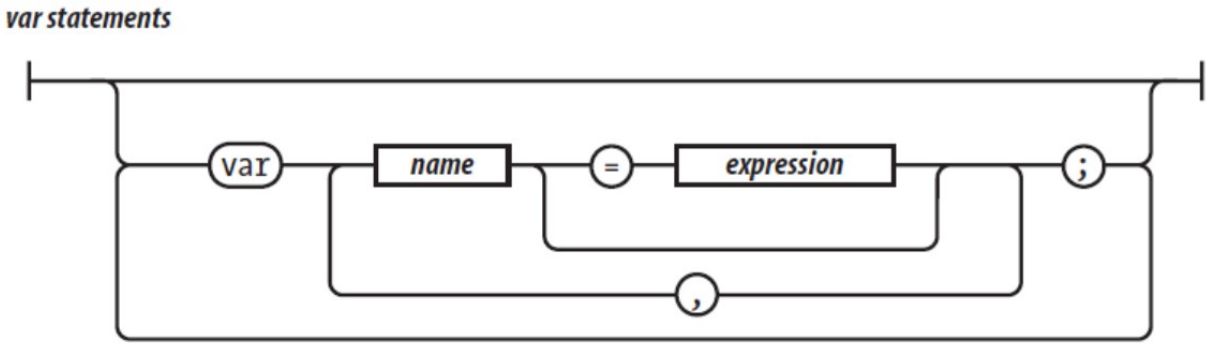
Dos cadenas que contienen exactamente los mismos caracteres en el mismo orden se consideran la misma cadena. Asi que:

```
'c' + 'a' + 't' === 'cat'    -> true
```

Las cadenas tienen métodos (véase el capítulo 8):

```
'cat'.toUpperCase( ) === 'CAT'
```

Declaraciones



Una unidad de compilación contiene un conjunto de sentencias ejecutables. En los navegadores web, cada etiqueta <script> proporciona una unidad de compilación compilada e inmediatamente ejecutada. Al carecer de un vinculador, JavaScript los arroja todos juntos en un espacio de nombres global común. Hay más en las variables globales en el Apéndice A.

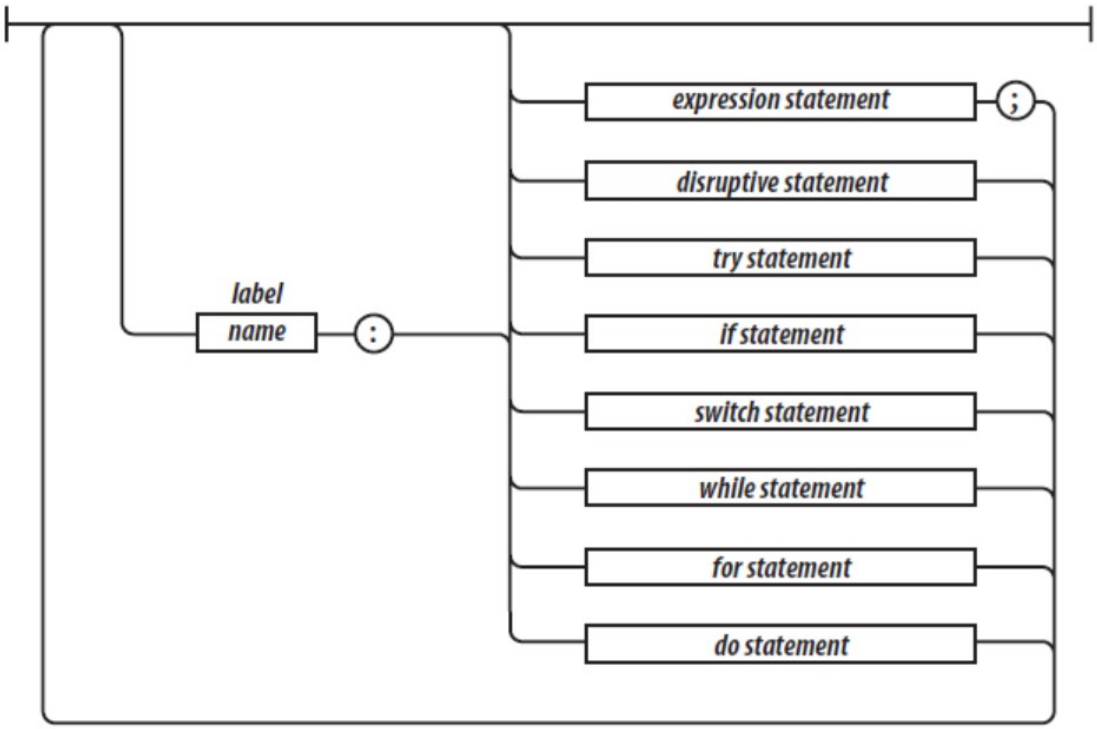
Cuando se utiliza dentro de una función, la instrucción var define las variables privadas de la función.

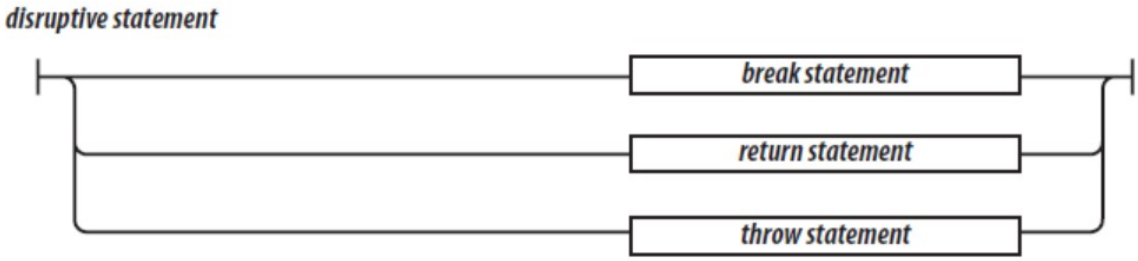
Las sentencias **switch**, **while**, **for** y **do** están permitidas tener un prefijo de etiqueta opcional que interactúa con la sentencia break.

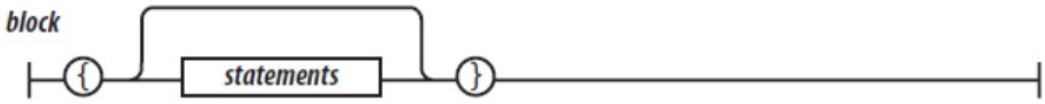
Las declaraciones tienden a ser ejecutadas en orden de arriba a abajo. La secuencia de ejecución puede ser alterada por las sentencias condicionales (**if** y **switch**), por las sentencias looping (**while**, **for** y **do**), por las sentencias disruptivas (**break**, **return** y **throw**) y por invocación de función.

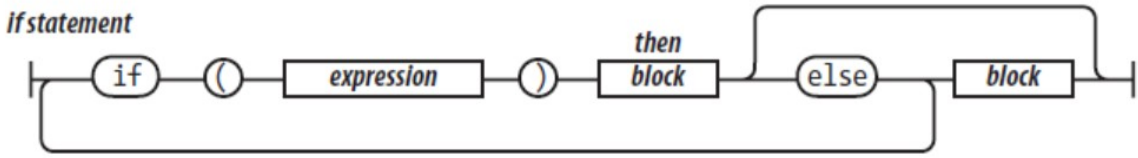
Un bloque es un conjunto de declaraciones envueltas en llaves. A diferencia de muchos otros lenguajes, los bloques en JavaScript no crean un nuevo ámbito, por lo que las variables deben definirse en la parte superior de la función, no en bloques.

La instrucción if cambia el flujo del programa basado en el valor de la expresión. El bloque entonces se ejecuta si la expresión es verídica; De lo contrario, se toma la derivación opcional.





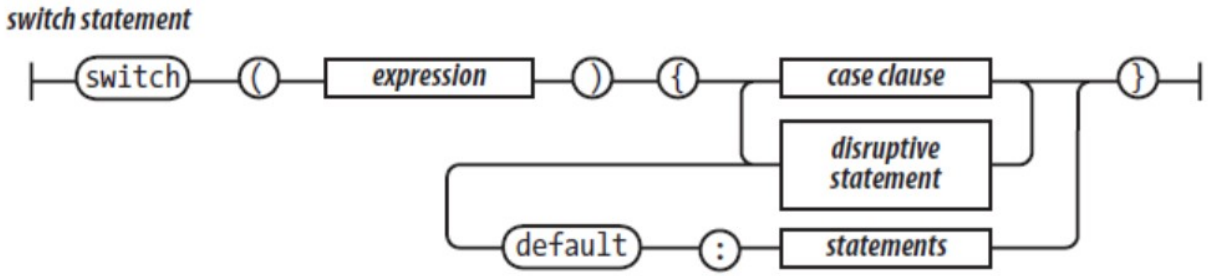




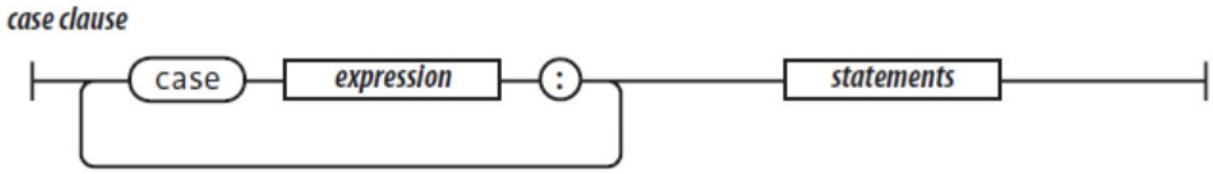
Estos son los valores falsos:

- false
- null
- undefined
- String vacío “”
- El número 0
- El número NaN

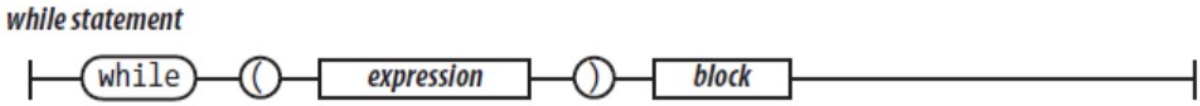
Todos los demás valores son verdaderos, incluyendo true, la cadena 'false', y todos los objetos.



La instrucción **switch** realiza una ramificación multipunto. Compara la expresión de igualdad con todos los casos especificados. La expresión puede producir un número o una cadena. Cuando se encuentra una coincidencia exacta, se ejecutan las sentencias de la cláusula de coincidencia del caso. Si no hay coincidencia, se ejecutan las instrucciones opcionales predeterminadas.



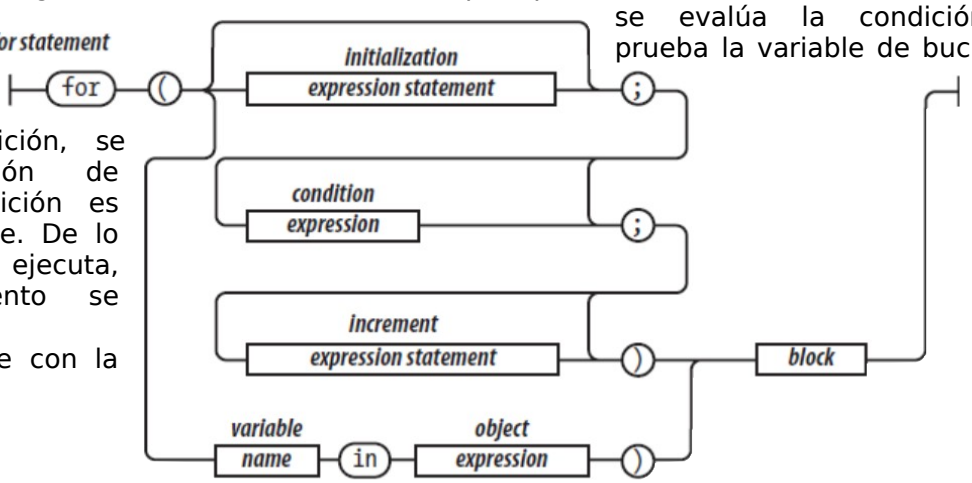
Una cláusula case contiene una o más expresiones de caso. Las expresiones de caso no necesitan ser constantes. La declaración que sigue a una cláusula debe ser una declaración disruptiva para evitar caer en el siguiente caso. La instrucción break se puede utilizar para salir de un switch.



La instrucción **while** realiza un bucle simple. Si la expresión es falsa, entonces el bucle se romperá. Mientras que la expresión es verdad, el bloque será ejecutado.

La instrucción **for** es una instrucción de bucle más complicada. Viene en dos formas.

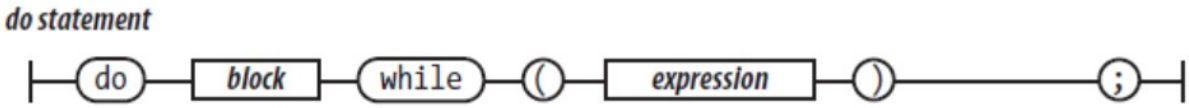
La forma convencional es controlada por tres cláusulas opcionales: la inicialización, la condición y el incremento. En primer lugar, se realiza la inicialización, que típicamente inicializa la variable de bucle. A continuación, se evalúa la condición. Normalmente, esto prueba la variable de bucle con respecto a un criterio de finalización.
Si se omite la condición, se asume una condición de verdadero. Si la condición es false, el bucle se rompe. De lo contrario, el bloque se ejecuta, entonces el incremento se ejecuta, y
luego el bucle se repite con la condición.



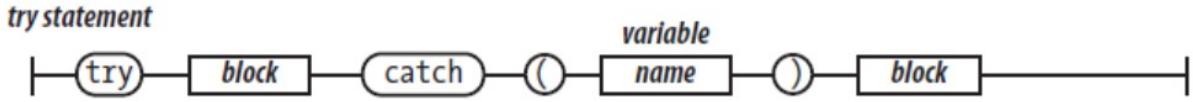
El otro formulario (llamado en) enumera los nombres de propiedad (o claves) de un objeto. En cada iteración, se asigna otra variable de nombre de propiedad del objeto a la variable.

Por lo general, es necesario probar **object.hasOwnProperty (variable)** para determinar si el nombre de la propiedad es realmente un miembro del objeto o se encontró en su lugar en la cadena de prototipos.

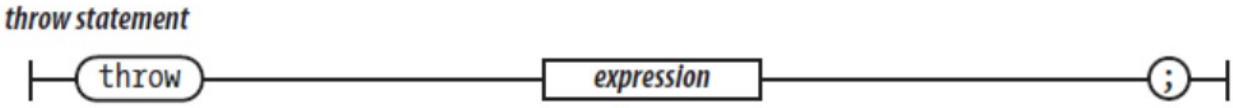
```
for (myvar in obj) {
    if (obj.hasOwnProperty(myvar)) {
        ...
    }
}
```



La instrucción do es como la sentencia while excepto que la expresión se prueba después de ejecutar el bloque en vez de antes. Esto significa que el bloque siempre se ejecutará al menos una vez.

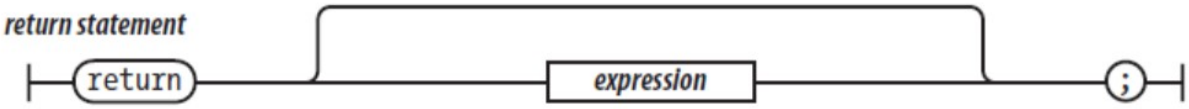


La sentencia try ejecuta un bloque y captura las excepciones que fueron lanzadas por el bloque. La cláusula catch define una nueva variable que recibirá el objeto de excepción.



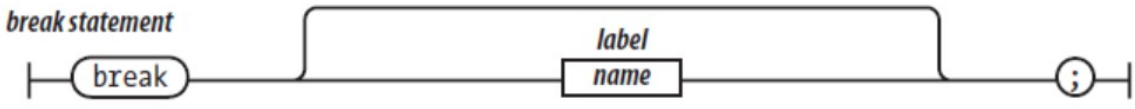
La sentencia throw genera una excepción. Si la sentencia throw está en un bloque try, entonces control va a la cláusula catch. De lo contrario, la función de invocación es abandonada, y el control va a la cláusula catch del try en la función de llamada.

La expresión suele ser un objeto literal que contiene una propiedad de nombre y una propiedad de mensaje. El receptor de la excepción puede usar esa información para determinar qué hacer.

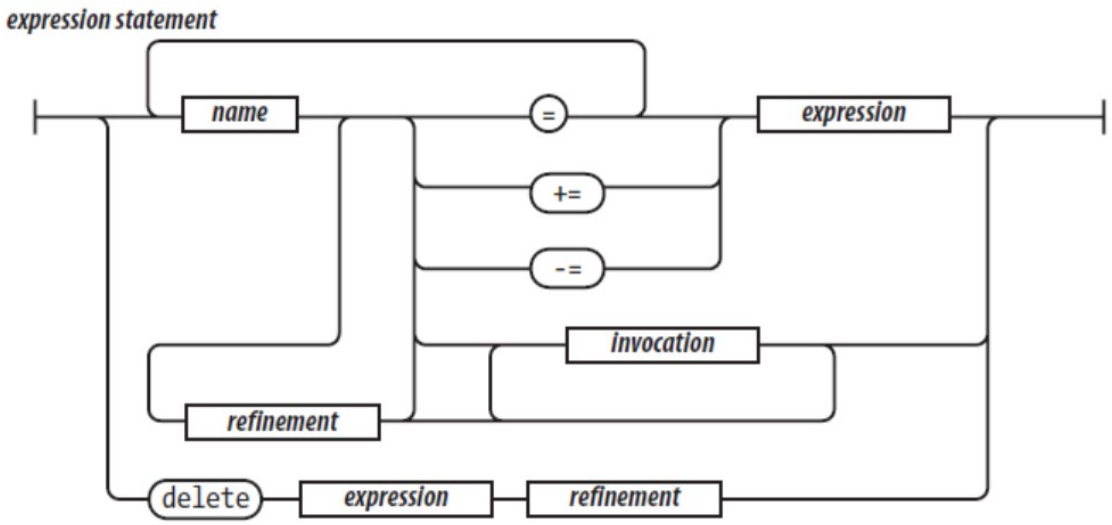


La instrucción **return** provoca la devolución anticipada de una función. También puede especificar el valor a devolver. Si no se especifica una expresión de retorno, el valor de retorno será undefined.

JavaScript no permite un final de línea entre el retorno y la expresión.

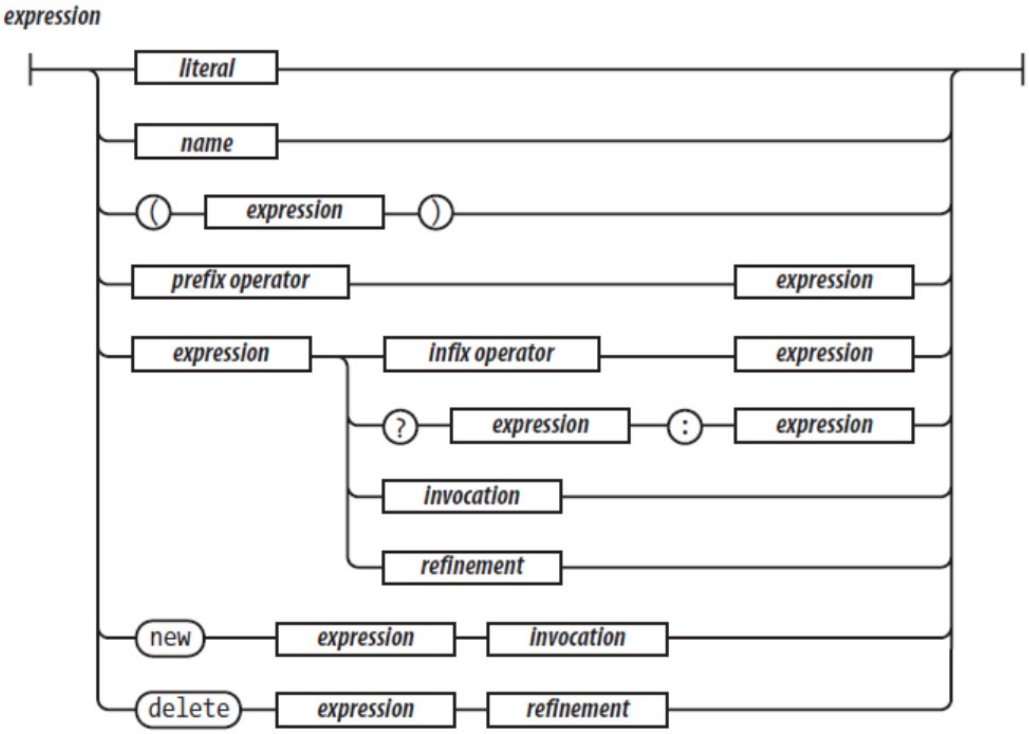


La instrucción break produce la salida de una instrucción loop o una instrucción switch. Opcionalmente puede tener una etiqueta que causará una salida de la declaración etiquetada. JavaScript no permite un final de línea entre el break y la etiqueta.



Una sentencia de expresión puede asignar valores a una o más variables o miembros, invocar un método, eliminar una propiedad de un objeto. El operador = se utiliza para la asignación. No lo confunda con el operador de igualdad ===. El operador + = puede agregar o concatenar.

Expressions



Las expresiones más simples son un valor literal (como una cadena o un número), una variable, un valor incorporado (verdadero, falso, nulo, indefinido, NaN o Infinito), una expresión de invocación precedida de new, una expresión de refinamiento precedida Por delete, una expresión envuelta entre paréntesis, una expresión precedida por un operador prefijo o una expresión seguida por:

- Un operador infix y otra expresión
- Los ? Ternario seguido por otra expresión, luego por:, y luego por otra expresión
- Una invocación
- Un refinamiento

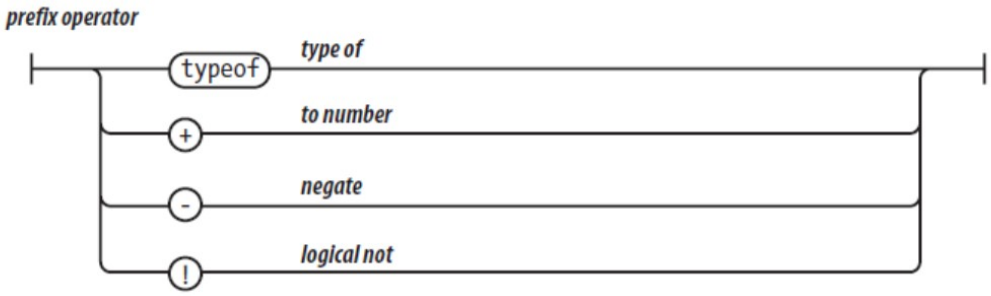
Los ? Operador ternario toma tres operandos. Si el primer operando es verdadero, produce el valor del segundo operando. Pero si el primer operando es falso, produce el valor del tercer operando.

Los operadores que se encuentran en la parte superior de la lista de precedentes del operador en la Tabla 2-1 tienen mayor precedencia. Se atan los más apretados. Los operadores en la parte inferior tienen la precedencia más baja. Los paréntesis se pueden usar para alterar la precedencia normal, así que:

```
2 + 3 * 5 === 17
(2 + 3) * 5 === 25
```

Table 2-1. Operator precedence

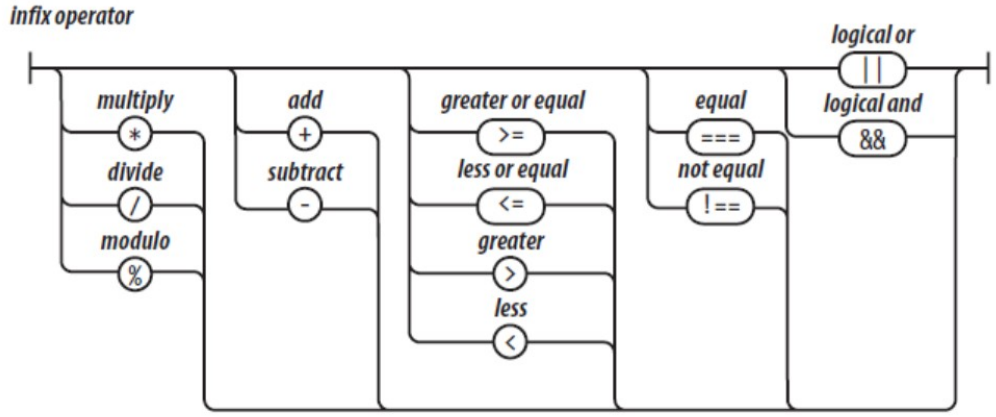
`. [] ()`	Refinement and invocation
`delete new typeof + - !`	Unary operators
`* / %`	Multiplication, division, modulo
`+ -`	Addition/concatenation, subtraction
`>= <= > <`	Inequality
`=== !==`	Equality
`&&`	Logical and
`||`	Logical or
`?:`	Ternary



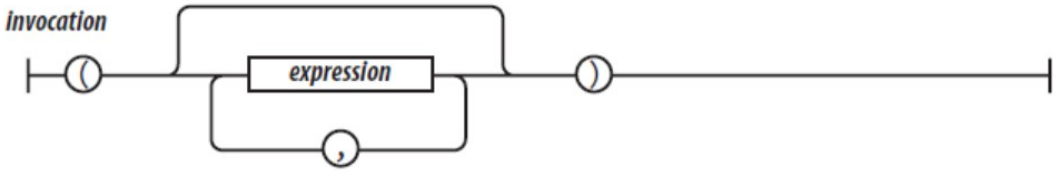
Los valores producidos por typeof son **'number'**, **'string'**, **'boolean'**, **'undefined'**, **'function'** y **'object'**. Si el operando es un **array** o **null**, entonces el resultado es **'object'**, lo cual es incorrecto. Habrá más sobre typeof en el Capítulo 6 y Apéndice A.

Si el operando de! Es verdad, produce falsa. De lo contrario, produce verdadero. El operador + agrega o concatena. Si desea que se agregue, asegúrese de que ambos operandos sean números.

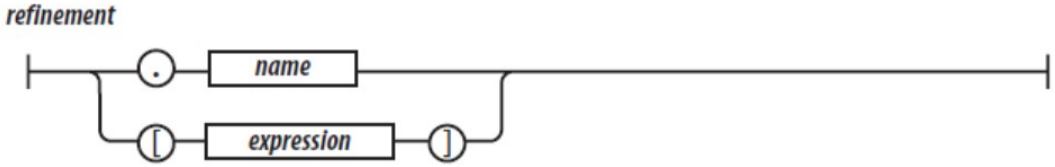
El operador / puede producir un resultado sin integrar incluso si ambos operandos son enteros. El operador && produce el valor de su primer operando si el primer operando es false. De lo contrario, produce el valor del segundo operando.



El || Operador produce el valor de su primer operando si el primer operando es verídico. De lo contrario, produce el valor del segundo operando.



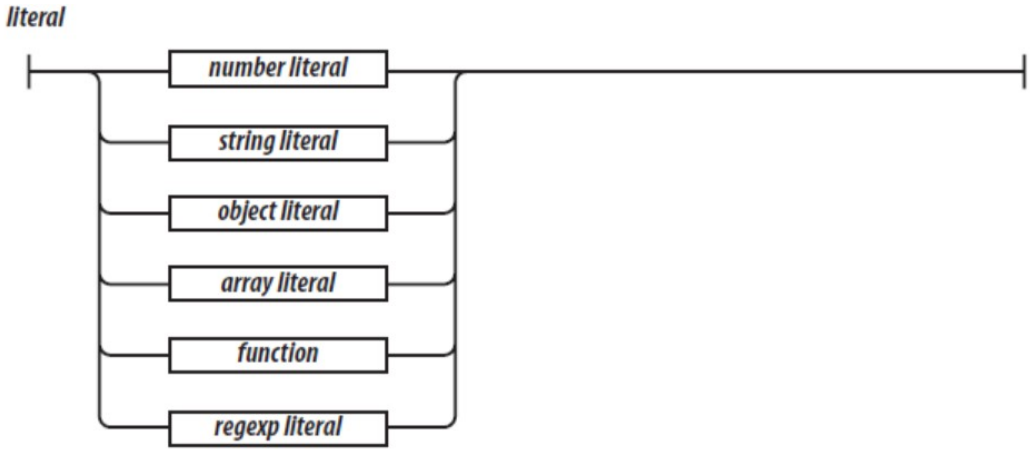
La invocación causa la ejecución de un valor de función. El operador de invocación es un par de paréntesis que siguen el valor de la función. Los paréntesis pueden contener argumentos que se entregarán a la función. Habrá mucho más acerca de las funciones en el Capítulo 4.

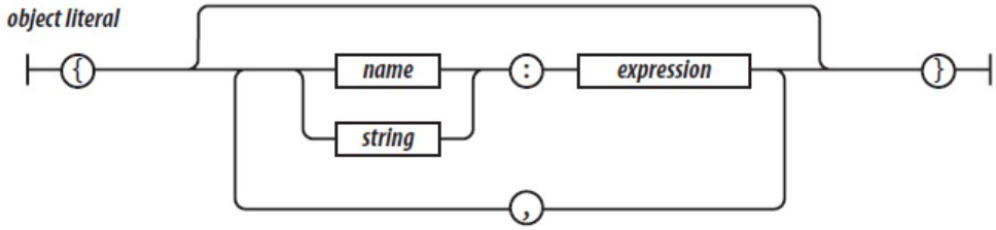


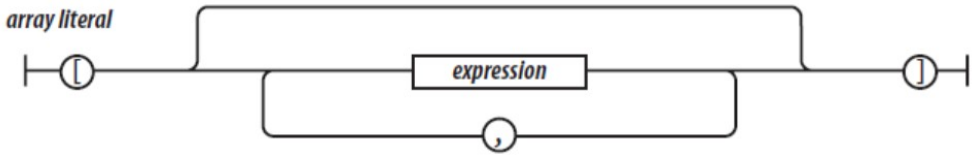
Un refinamiento se utiliza para especificar una propiedad o elemento de un objeto o array. Esto se describirá en detalle en el próximo capítulo.

Literales

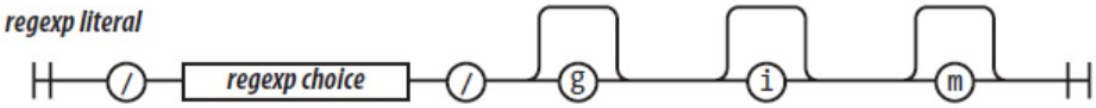
Los literales de objetos son una notación conveniente para especificar nuevos objetos. Los nombres de las propiedades se pueden especificar como nombres o como cadenas. Los nombres se tratan como nombres literales, no como nombres de variable, por lo que los nombres de las propiedades del objeto deben ser conocidos en tiempo de compilación. Los valores de las propiedades son expresiones. Habrá más acerca de los literales de objetos en el próximo capítulo.





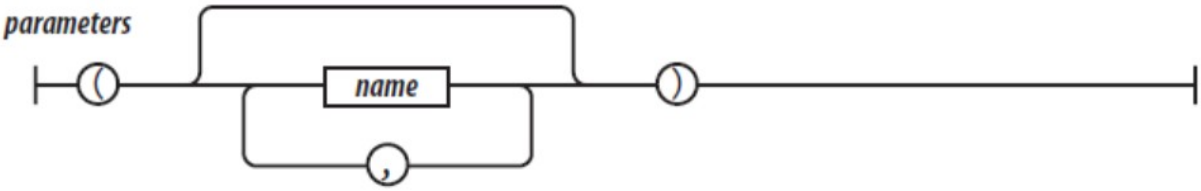


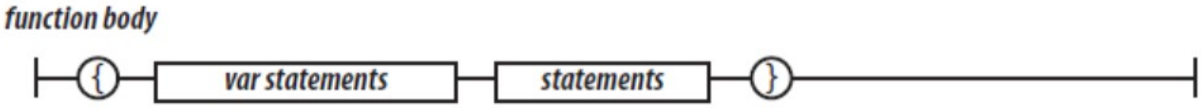
Los literales de arrays son una notación conveniente para especificar nuevos arrays. Habrá más acerca de los literales de array en el Capítulo 6.



Habrá más sobre las expresiones regulares en el Capítulo 7.

Funciones





Una función literal define un valor de función. Puede tener un nombre opcional que puede usar para llamarse recursivamente. Puede especificar una lista de parámetros que actuarán como variables inicializadas por los argumentos de invocación. El cuerpo de la función incluye definiciones de variables y declaraciones. Habrá más información sobre las funciones en el Capítulo 4.

Capítulo 3. Object

Los tipos simples de JavaScript son números, cadenas, booleanos (true y false), null y undefined. Todos los demás valores son objetos. Los números, las cadenas y los booleanos son similares a los objetos, ya que tienen métodos, pero son inmutables. Los objetos en JavaScript son mutable keyed collection. En JavaScript, los arrays son objetos, las funciones son objetos, las expresiones regulares son objetos y, por supuesto, los objetos son objetos.

Un objeto es un contenedor de propiedades, donde una propiedad tiene un nombre y un valor. Un nombre de propiedad puede ser cualquier cadena, incluyendo la cadena vacía. Un valor de propiedad puede ser cualquier valor de JavaScript excepto undefined.

Los objetos en JavaScript no tienen clase. No hay ninguna restricción en los nombres de las nuevas propiedades o en los valores de las propiedades. Los objetos son útiles para recopilar y organizar datos. Los objetos pueden contener otros objetos, por lo que pueden representar fácilmente estructuras de árbol o de gráfico.

JavaScript incluye una característica de enlace de prototipo que permite a un objeto heredar las propiedades de otro. Cuando se utiliza bien, esto puede reducir el tiempo de inicialización de objetos y el consumo de memoria.

Literales de objetos

Los literales de objetos proporcionan una notación muy conveniente para crear nuevos valores de objeto. Un objeto literal es un par de llaves que rodean cero o más pares nombre / valor. Un literal de objeto puede aparecer en cualquier lugar donde pueda aparecer una expresión:

```
var empty_object = {};

var stooge = {
    "first-name": "Jerome",
    "last-name": "Howard"
};
```

El nombre de una propiedad puede ser cualquier cadena, incluyendo la cadena vacía. Las comillas alrededor del nombre de una propiedad en un literal de objeto son opcionales si el nombre sería un nombre JavaScript legal y no una palabra reservada. Así que las citas se requieren alrededor de "firstname", pero son opcionales alrededor de first_name. Las comas se utilizan para separar los pares.

El valor de una propiedad se puede obtener de cualquier expresión, incluyendo otro literal de objeto. Los objetos pueden anidar:

```
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

Recuperación

Los valores se pueden recuperar de un objeto envolviendo una expresión de cadena en un sufijo []. Si la expresión string es una constante, y si es un nombre JavaScript legal y no una palabra reservada, entonces el archivo. Notación se puede utilizar en su lugar. Los . Se prefiere la notación porque es más compacta y se lee mejor:

```
stooge["first-name"]     // "Joe"
flight.departure.IATA    // "SYD"
```

El valor **undefined** se produce si se intenta recuperar un miembro inexistente:

```
stooge["middle-name"]    // undefined
flight.status            // undefined
stooge["FIRST-NAME"]     // undefined
```

El || Operador se puede utilizar para rellenar los valores por defecto:

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

Si intenta recuperar valores **undefined**, lanzará una excepción TypeError. Esto puede protegerse con el operador &&:

```
flight.equipment                              // undefined
flight.equipment.model                        // throw "TypeError"
flight.equipment && flight.equipment.model    // undefined
```

Update

Un valor en un objeto se puede actualizar mediante asignación. Si el nombre de propiedad ya existe en el objeto, se reemplazará el valor de la propiedad:

```
stooge['first-name'] = 'Jerome';
```

Si el objeto no tiene ya ese nombre de propiedad, el objeto se aumenta:

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```

Referencia

Los objetos se pasan por referencia. Nunca se copian:

```
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
    // nick is 'Curly' because x and stooge
    // are references to the same object
```

```
var a = {}, b = {}, c = {};
    // a, b, and c each refer to a
    // different empty object
a = b = c = {};
    // a, b, and c all refer to
    // the same empty object
```

Prototype

Cada objeto está vinculado a un objeto prototipo del que puede heredar propiedades. Todos los objetos creados a partir de literales de objeto están vinculados a Object.prototype, un objeto que viene de serie con JavaScript.

Cuando crea un nuevo objeto, puede seleccionar el objeto que debe ser su prototipo. El mecanismo que proporciona JavaScript para hacer esto es desordenado y complejo, pero puede simplificarse significativamente. Vamos a agregar un método **create** a la función Object. El método **create** crea un nuevo objeto que utiliza un objeto antiguo como su prototipo. Habrá mucho más acerca de las funciones en el próximo capítulo.

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
var another_stooge = Object.create(stooge);
```

El enlace del prototype no tiene ningún efecto en la actualización. Cuando realizamos cambios en un objeto, no se toca el prototipo del objeto:

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

El enlace de Prototype sólo se utiliza en la recuperación. Si tratamos de recuperar un valor de propiedad de un objeto y si el objeto carece del nombre de propiedad, JavaScript intenta recuperar el valor de la propiedad del objeto Prototype. Y si ese objeto carece de la propiedad, entonces va a su Prototype, y así sucesivamente hasta que el proceso finalmente termina con Object.prototype. Si la propiedad deseada no existe en ninguna parte de la cadena de prototipos, el resultado es el valor indefinido. Esto se llama delegación.

La relación de Prototype es una relación dinámica. Si agregamos una nueva propiedad a un Prototype, esa propiedad será inmediatamente visible en todos los objetos que se basan en ese prototipo:

```
stooge.profession = 'actor';
another_stooge.profession    // 'actor'
```

Veremos más sobre la cadena de prototipos en el Capítulo 6.

Reflexión

Es fácil inspeccionar un objeto para determinar qué propiedades tiene intentando recuperar las propiedades y examinar los valores obtenidos. El operador typeof puede ser muy útil para determinar el tipo de propiedad:

```
typeof flight.number      // 'number'
typeof flight.status      // 'string'
typeof flight.arrival     // 'object'
typeof flight.manifest    // 'undefined'
```

Se debe tener cuidado porque cualquier propiedad en la cadena de prototipos puede producir un valor:

```
typeof flight.toString    // 'function'
typeof flight.constructor // 'function'
```

Existen dos enfoques para tratar estas propiedades no deseadas. El primero es hacer que su programa busque y rechace los valores de las funciones. Generalmente, cuando estás reflexionando, te interesan los datos, por lo que debes tener en cuenta que algunos valores podrían ser funciones.

El otro enfoque es utilizar el método **hasOwnProperty**, que devuelve true si el objeto tiene una propiedad particular. El método **hasOwnProperty** no mira la cadena de prototype:

```
flight.hasOwnProperty('number')         // true
flight.hasOwnProperty('constructor')    // false
```

Enumeración

La sentencia **for in** puede realizar bucle sobre todos los nombres de propiedad de un objeto. La enumeración incluirá todas las propiedades, incluidas las funciones y las propiedades del prototipo en las que quizás no esté interesado, por lo que es necesario filtrar los valores que no desea. Los filtros más comunes son el método hasOwnProperty y el uso de typeof para excluir funciones:

```
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

No hay garantía en el orden de los nombres, así que prepárate para que los nombres aparezcan en cualquier orden. Si desea asegurarse de que las propiedades aparecen en un orden en particular, lo mejor es evitar la sentencia **for** en su totalidad y, en su lugar, crear una array que contenga los nombres de las propiedades en el orden correcto:

```
var i;
var properties = [
    'first-name',
    'middle-name',
    'last-name',
    'profession'
];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' +
            another_stooge[properties[i]]);
}
```

first-name: MAriana
middle-name: Lester
last-name: Howard
profession: undefined

Usando para el lugar de para en, pudimos conseguir las propiedades que buscaban sin preocuparnos por lo que podía ser dragado de la cadena de prototipos, y los conseguimos en el orden correcto.

Borrar - delete

El operador **delete** se puede utilizar para quitar una propiedad de un objeto. Eliminará una propiedad del objeto si la tiene. No tocará ninguno de los objetos del enlace del prototipo.

Eliminar una propiedad de un objeto puede permitir que una propiedad del enlace de prototipo brille a través de:

```
another_stooge.nickname    // 'Moe'

// Remove nickname from another_stooge, revealing
// the nickname of the prototype.
  delete another_stooge.nickname;

  another_stooge.nickname    // 'Curly'
```

Variables Globales

JavaScript hace que sea fácil definir variables globales que pueden contener todos los activos de su aplicación. Desafortunadamente, las variables globales debilitan la resiliencia de los programas y deben ser evitadas.

Una forma de minimizar el uso de variables globales es crear una única variable global para su aplicación:

```
var MYAPP = {};
```

Esa variable entonces se convierte en el contenedor para su aplicación:

```
MYAPP.stooge = {
    "first-name": "Joe",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

Capítulo 4. Funciones

Lo mejor de JavaScript es su implementación de funciones. Se hizo casi todo bien. Pero, como usted debe esperar con JavaScript, no lo hizo todo bien.

Una función incluye un conjunto de sentencias. Las funciones son la unidad modular fundamental de JavaScript. Se utilizan para la reutilización del código, la ocultación de la información y la composición. Las funciones se utilizan para especificar el comportamiento de los objetos. Generalmente, el arte de la programación es el factoraje de un conjunto de requisitos en un conjunto de funciones y estructuras de datos.

Function Objects

Las funciones en JavaScript son objetos. Los objetos son colecciones de pares nombre / valor que tienen un vínculo oculto con un objeto prototipo. Los objetos producidos a partir de literales de objeto están vinculados a Object.prototype. Los objetos de función están vinculados a Function.prototype (que está vinculado a Object.prototype). Cada función también se crea con dos propiedades ocultas adicionales: el contexto de la función y el código que implementa el comportamiento de la función.

Cada objeto de función también se crea con una propiedad de prototype. Su valor es un objeto con una propiedad constructor cuyo valor es la función. Esto es distinto del enlace oculto a Function.prototype. El significado de esta complicada construcción se revelará en el próximo capítulo.

Dado que las funciones son objetos, pueden utilizarse como cualquier otro valor. Las funciones pueden almacenarse en variables, objetos y arrays. Las funciones se pueden pasar como argumentos a las funciones, y las funciones se pueden volver de las funciones. Además, puesto que las funciones son objetos, las funciones pueden tener métodos.

Lo que es especial acerca de las funciones es que pueden ser invocadas.

Función Literal

Los objetos de función se crean con literales de función:

```
// Create a variable called add and store a function
// in it that adds two numbers.

var add = function (a, b) {
    return a + b;
};
```

Una función literal tiene cuatro partes. La primera parte es la función de palabra reservada.

La segunda parte opcional es el nombre de la función. La función puede usar su nombre para llamarse recursivamente. El nombre también puede ser utilizado por depuradores y herramientas de desarrollo para identificar la función. Si una función no recibe un nombre, como se muestra en el ejemplo anterior, se dice que es anónima.

La tercera parte es el conjunto de parámetros de la función, envueltos entre paréntesis. Dentro de los paréntesis está un conjunto de cero o más nombres de parámetros, separados por comas. Estos nombres se definirán como variables en la función. A diferencia de las variables ordinarias, en lugar de inicializarse a indefinido, se inicializarán a los argumentos suministrados cuando se invoca la función.

La cuarta parte es un conjunto de declaraciones envueltas en llaves. Estas declaraciones son el cuerpo de la función. Se ejecutan cuando se invoca la función.

Invocación

Invocar una función suspende la ejecución de la función actual, pasando el control y los parámetros a la nueva función. Además de los parámetros declarados, cada función recibe dos parámetros adicionales: **this** y **arguments**. Este parámetro es muy importante en la programación orientada a objetos, y su valor está determinado por el patrón de invocación. Existen cuatro patrones de invocación en JavaScript: el patrón de invocación de **método**, el patrón de invocación de **función**, el patrón de invocación de **constructor** y el patrón de invocación de **aplicación**. Los patrones difieren en cómo esto se inicializa.

El operador de invocación es un par de paréntesis que siguen cualquier expresión que produce un valor de función. Los paréntesis pueden contener cero o más expresiones, separadas por comas. Cada expresión produce un valor de argumento. Cada uno de los valores de los argumentos se asignará a los nombres de los parámetros de la función. No hay ningún error de tiempo de ejecución cuando el número de argumentos y el número de parámetros no coinciden. Si hay demasiados valores de argumento, los valores de argumento extra se ignorarán. Si hay pocos valores de argumento, el valor **indefinido** se sustituirá por los valores faltantes. No hay ninguna comprobación de tipos en los valores de los argumentos: cualquier tipo de valor se puede pasar a cualquier parámetro.

El patrón de invocación del método

Cuando una función se almacena como una propiedad de un objeto, lo llamamos un método. Cuando se invoca un método, éste está enlazado a ese objeto. Si una expresión de invocación contiene un refinamiento (es decir, una expresión de punto o una expresión de subíndice), se invoca como un método:

```
var myObject = {
    value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};

myObject.increment( );
document.writeln(myObject.value);    // 1

myObject.increment(2);
document.writeln(myObject.value);    // 3
```

Un método puede usar esto para acceder al objeto para que pueda recuperar valores del objeto o modificar el objeto. La vinculación de esto con el objeto ocurre en el momento de la invocación. Esta vinculación muy tardía hace que las funciones que utilizan este altamente reutilizable. Métodos que obtienen su contexto de objeto de esto se llaman métodos públicos.

El patrón de invocación de la función

Cuando una función no es propiedad de un objeto, se invoca como una función:

```
var sum = add(3, 4); // sum is 7
```

Cuando se invoca una función con este patrón, ésta está enlazada al objeto global. Esto fue un error en el diseño del lenguaje. Si el lenguaje hubiera sido diseñado correctamente, cuando se invocara la función interna, esto seguiría vinculado a esta variable de la función externa. Una consecuencia de este error es que un método no puede emplear una función interna para ayudarle a hacer su trabajo porque la función interna no comparte el acceso del método al objeto como su

esto está atado al valor incorrecto. Afortunadamente, hay una solución fácil. Si el método define una variable y asigna el valor de ésta, la función interna tendrá acceso a ésta a través de esa variable. Por convención, el nombre de esa variable es que:

```
// Augment myObject with a double method.

myObject.double = function ( ) {
    var that = this;    // Workaround.

    var helper = function ( ) {
        that.value = add(that.value, that.value);
    };

    helper( );    // Invoke helper as a function.
};

// Invoke double as a method.

myObject.double( );
document.writeln(myObject.getValue( ));    // 6
```

El patrón de invocación del constructor

JavaScript es un lenguaje prototypal de herencia. Esto significa que los objetos pueden heredar propiedades directamente de otros objetos. El idioma está libre de clases.

Esto es una salida radical de la moda actual. La mayoría de los idiomas de hoy son clásicos. La herencia prototípica es poderosamente expresiva, pero no se entiende ampliamente. El propio JavaScript no confía en su naturaleza prototípica, por lo que ofrece una sintaxis de creación de objetos que recuerda a las lenguas clásicas. Pocos programadores clásicos encontraron que la herencia prototípica era aceptable, y la sintaxis de inspiración clásica oscurece la verdadera naturaleza prototípica del lenguaje. Es lo peor de ambos mundos.

Si se invoca una función con el nuevo prefijo, se creará un nuevo objeto con un vínculo oculto con el valor del miembro prototipo de la función, y éste se enlazará a ese nuevo objeto.

El nuevo prefijo también cambia el comportamiento de la instrucción return. Veremos más sobre eso después.

```
// Create a constructor function called Quo.
// It makes an object with a status property.

var Quo = function (string) {
    this.status = string;
};

// Give all instances of Quo a public method
// called get_status.

Quo.prototype.get_status = function ( ) {
    return this.status;
};

// Make an instance of Quo.

var myQuo = new Quo("confused");

document.writeln(myQuo.get_status( ));  // confused
```

Las funciones que se pretenden utilizar con el nuevo prefijo se llaman constructores. Por convención, se mantienen en variables con un nombre mayúscula. Si un constructor se llama sin el nuevo prefijo, cosas muy malas pueden suceder sin una advertencia de tiempo de compilación o de tiempo de ejecución, por lo que la convención de capitalización es realmente importante.

No se recomienda el uso de este estilo de funciones de constructor. Veremos mejores alternativas en el próximo capítulo.

El patrón de invocación Pattern

Debido a que JavaScript es un lenguaje funcional orientado a objetos, las funciones pueden tener métodos.

El método **apply** nos permite construir un array de argumentos para invocar una función. También nos permite elegir el valor de esto. El método apply toma dos parámetros. El primero es el valor que debe estar vinculado a esto. La segunda es un array de parámetros.