Table of Contents

Javascript Avanzado en Español

Esta es una traducción de la serie de libros de You Don't Know JS (book series), la cual es una serie de 6 libros que navegan profundamente en los mecanismos básicos y avanzados del lenguaje JavaScript. La primera edición de la serie está ahora completa.

Porque realizo este trabajo?

Mi nombre es Daniel Morales, este es mi perfil de GitHub, y soy un apasionado de Ruby y Ruby On Rails y de Javascript. Inicié la traducción de estos libros para comprender mejor el funcionamiento de Javascript, por tanto era un deseo tener la documentación traducida para mi, pero de paso pensé "Nadie ha realizado una traducción **COMPLETA** al español, porqué no hacerlo?". La publico para que todos los programadores de habla hispana puedan aprovecharse de este material.

Si deseas hacer alguna corrección o aporte (ya que seguro podría tener errores de traducción o interpretación), puedes hacer un fork al repo principal y solicitar un pull-request, apreciaré dicha ayuda!

Quien es el Autor detrás de la Serie de Libros?

Como mencioné anteriormente, no soy el creador de la serie de libros, solo el traductor al español. El creador es Kyle Simpson, cuyo trabajo esta bajo licencia Creative Commons Attribution-NonCommercial-NoDerivs 4.0 Unported License.

¿Por Donde Empezar?

Aunque ésta es una serie de libros de Javascript Avanzado, tambien puede empezarlos con un nivel básico.

- Si usted tiene nivel básico en Programación y/o en Javascript = Vaya al Libro # 1 - Up & Going
- Si usted tiene un nivel básico/medio/avanzado en Programación y/o en Javascrip = Vaya al Libro # 2 - Scope & Closures

I- Up & Going

Prefacio

¿Cuál fue la última cosa nueva que aprendiste?

Tal vez fue una lengua extranjera, como italiano o alemán. O tal vez un editor de gráficos, como Photoshop. O una técnica de cocción o carpintería o una rutina de ejercicios. Quiero que recuerdes esa sensación cuando finalmente lo conseguiste: el momento que alumbró tu bombilla. Cuando las cosas pasaban de borrosas a cristalinas, cuando lo dominabas veías o entendías la diferencia entre sustantivos masculinos y femeninos en francés. ¿Como se sintió? Bastante asombroso, ¿verdad?

Ahora quiero que vayas un poco más lejos en tu memoria antes de que aprendieras una nueva habilidad. ¿Cómo se sintió eso? Probablemente un poco intimidante y tal vez un poco frustrante, ¿verdad? En un momento, no sabíamos las cosas que sabemos ahora y eso está bien; Todos empezamos en alguna parte. Aprender material nuevo es una aventura emocionante, especialmente si usted está buscando aprender el tema de manera eficiente.

Enseño muchas clases de codificación a principiantes. Los estudiantes que toman mis clases a menudo han tratado de aprender por sí mismos temas como HTML o JavaScript por medio de la lectura de blogs o copiar y pegar código, pero no han sido capaces de dominar realmente el material que les permita codificar el resultado deseado. Y debido a que no comprenden realmente los entresijos de ciertos temas de codificación, no pueden escribir código de gran alcance o depurar su propio trabajo, ya que realmente no entienden lo que está sucediendo.

Siempre creo en enseñar mis clases de la manera correcta, es decir, enseño estándares web, marcado semántico, código bien comentado y otras buenas prácticas. Cubro el tema de una manera minuciosa para explicar los comos y los porqués, sin sacar el código para copiar y pegar. Cuando te esfuerzas en comprender tu código, creas mejor trabajo y te vuelves mejor en lo que haces. El código ya no es solo tu trabajo, es tu oficio. Esta es la razón por la que amo Up & Going. Kyle nos lleva a una inmersión profunda a través de la sintaxis y la terminología para dar una gran introducción a JavaScript sin cortes. Este libro no navega sobre la superficie del lenguaje, sino que realmente nos permite entender los conceptos que estaremos escribiendo.

Debido a que no es suficiente poder duplicar fragmentos jQuery en su sitio web, de la misma manera no es suficiente aprender a abrir, cerrar y guardar un documento en Photoshop. Claro, una vez que aprenda algunos conceptos básicos sobre el programa

podría crear y compartir un diseño que se hace. Pero sin conocer legítimamente las herramientas y lo que hay detrás de ellas, ¿cómo puedo definir una cuadrícula, crear un sistema de tipo legible u optimizar los gráficos para el uso de la web. Lo mismo ocurre con JavaScript. Sin saber cómo funcionan los bucles, o cómo definir variables, o qué alcance tiene, no estaremos escribiendo el mejor código que podamos. No queremos conformarnos con nada menor a ésto - esto es, después de todo, nuestro oficio.

Cuanto más se exponen a JavaScript, más claro se vuelve. Palabras como closures, objetos y métodos pueden parecer fuera de alcance para usted ahora, pero este libro ayudará a entender estos términos con claridad. Quiero que mantenga esos dos sentimientos en mente antes y después de que aprenda algo al comenzar este libro. Puede parecer desalentador, pero has elegido este libro porque estás empezando un viaje impresionante para perfeccionar tu conocimiento. Up & Going es el comienzo de nuestro camino hacia la comprensión de la programación. Disfrute de los momentos bombilla!

Jenn Lukas

Jennlukas.com, @jennlukas

Consultora de front-end

0- Prefacio

Estoy seguro de que se dio cuenta, pero "JS" en el título de la serie de libros no es una abreviatura de palabras utilizadas para maldecir sobre JavaScript, aunque maldecir a las peculiaridades del lenguaje es algo con lo que probablemente todos se pueden identificar.

Desde los primeros días de la web, JavaScript ha sido una tecnología fundamental que ha impulsado la experiencia interactiva en torno al contenido que consumimos. Mientras que el apuntador parpadeante y los molestos avisos emergentes podrían ser donde comenzó JavaScript, casi dos décadas después, la tecnología y la capacidad de JavaScript ha crecido en muchos ámbitos de magnitud, y pocos dudan de su importancia en el corazón de la plataforma de software más ampliamente disponible: La web.

Pero como lenguaje, ha sido perpetuamente un blanco para mucha crítica, debido en parte a su "herencia", pero más aún debido a su filosofía de diseño. Incluso el nombre evoca, como Brendan Eich una vez lo dijo, "el hermano bebe estúpido" comparado junto a su hermano mayor más maduro "Java". Pero el nombre es simplemente un accidente de la política y el marketing. Los dos idiomas son muy diferentes en muchos aspectos importantes. "JavaScript" está relacionado con "Java" como "Carnival" es a "Car".

Debido a que JavaScript toma conceptos y sintaxis idiomáticos de varios idiomas, incluyendo orgullosas raíces procedimentales de estilo C, así como raíces funcionales sutiles, menos obvias de estilo Scheme/Lisp, es sumamente accesible a una amplia audiencia de desarrolladores, incluso aquellos con poca o sin experiencia en programación. El "Hello World" de JavaScript es tan simple que el idioma se hace atractivo y es fácil de ponerse cómodo con la comprensión temprana.

Mientras que JavaScript es quizás uno de los idiomas más fáciles de poner en funcionamiento, sus excentricidades hacen que el dominio sólido del lenguaje sea una ocurrencia mucho menos común que en muchos otros idiomas. Cuando se necesita un conocimiento bastante profundo de un lenguaje como C o C++ para escribir un programa a gran escala, la producción a gran escala de JavaScript puede, y con frecuencia lo es, apenas se raya con la superficie de lo que el lenguaje puede realmente hacer.

Los conceptos sofisticados, que están profundamente arraigados en el lenguaje, tienden a aparecer en formas aparentemente simplistas, como pasar las funciones como devoluciones de llamada, lo que anima al desarrollador de JavaScript a utilizar el lenguaje tal como está y ha no preocuparse demasiado por lo que está pasando bajo la capucha.

Es simultáneamente un lenguaje sencillo y fácil de usar que tiene un amplio atractivo y una colección compleja y matizada de mecánica del lenguaje que sin un estudio cuidadoso escapará a la verdadera comprensión, incluso de los más experimentados desarrolladores de JavaScript.

Ahí radica la paradoja de JavaScript, el talón de Aquiles de la lengua, el desafío que estamos abordando actualmente. Debido a que JavaScript se puede utilizar sin comprender, la comprensión del lenguaje a menudo nunca se logra.

Misión

Si en cada punto que encuentres una sorpresa o frustración en JavaScript, tu respuesta es añadirlo a la lista negra, como algunos están acostumbrados a hacer, pronto serás relegado a una concha hueca de la riqueza de JavaScript.

Si bien este subconjunto ha sido conocido como "The Good Parts", le pediría a usted, querido lector, que lo considere "The Easy Parts", "The Safe Parts" o incluso "The Incomplete Parts".

Esta serie de libros de Javascript ofrece un desafío contrario: aprenda y entienda profundamente todo Javascript, incluso y especialmente "las piezas resistentes (The Tough Parts)".

Aquí, nos dirigimos a los desarrolladores de JS que tienen en la cabeza la mentalidad de aprender "lo suficiente" para continuar, sin nunca obligarse a aprender exactamente cómo y por qué el lenguaje se comporta de la manera que lo hace. Además, evitamos el consejo común de retirarse cuando el camino se vuelve áspero.

No estoy contento, ni debes estar, en parar una vez que algo funciona correctamente, y no saber realmente por qué. Le desafío a viajar por ese "camino costoso" y abrazar todo lo que JavaScript es y puede hacer. Con ese conocimiento, ninguna técnica, ningún framework, ningún acrónimo popular de moda, estará más allá de su comprensión.

Estos libros toman partes específicas del lenguaje que son las comúnmente mal entendidas o no comprendidas, y se sumerge muy profundamente y exhaustivamente en ellas. Usted debe termina la lectura con una firme confianza en su comprensión, no sólo de lo teórico, sino lo práctico "lo que necesita saber".

El JavaScript que usted conoce ahora mismo es probablemente la partes dada a usted por otros que han sido "quemados" por una comprensión incompleta. Ese JavaScript es sólo una sombra del verdadero lenguaje. Realmente no sabes JavaScript, pero si crees en esta serie, lo harás. Sigue leyendo, amigos. JavaScript te espera.

Resumen

JavaScript es impresionante. Es fácil de aprender en parte, y mucho más difícil de aprender por completo (o incluso lo suficiente). Cuando los desarrolladores encuentran confusión, suelen culpar al idioma en lugar de su falta de comprensión. Estos libros apuntan a arreglar eso, inspirando una apreciación fuerte del lenguaje para que usted pueda entenderlo ahora, y deba, profundamente entenderlo.

Nota: Muchos de los ejemplos en este libro asumen los entornos de motor de JavaScript modernos (y futuros), como ES6. Es posible que algunos códigos no funcionen como se describe si se ejecutan en motores anteriores (pre-ES6).

1- En la programación

Bienvenido a la serie You Do not Know JS (YDKJS).

Up & Going es una introducción a varios conceptos básicos de programación - por supuesto nos inclinamos hacia JavaScript (a menudo abreviado JS) específicamente - y cómo acercarse y entender el resto de los títulos de esta serie. Especialmente si usted acaba de entrar en la programación y/o JavaScript, este libro explorará brevemente lo que necesita para levantarse y seguir.

Este libro comienza explicando los principios básicos de la programación a un nivel muy alto. Está pensado principalmente si usted está comenzando YDKJS con poca o ninguna experiencia de programación, y está mirando a estos libros para ayudarle a comenzar a lo largo de un camino a la comprensión de la programación a través del lente de JavaScript.

El capítulo 1 debe ser abordado como un resumen rápido de las cosas que usted querrá aprender más y practicar para entrar en la programación. También hay muchos otros fantásticos recursos de introducción a la programación que pueden ayudarle a profundizar en estos temas y le animo a aprender de ellos además de este capítulo.

Una vez que se sienta cómodo con los conceptos básicos generales de programación, el Capítulo 2 le ayudará a familiarizarse con el sabor de la programación de JavaScript. El Capítulo 2 presenta lo que es JavaScript, pero de nuevo, no es una guía completa - ¡eso se lo dejamos al resto de los libros de YDKJS!

Si ya está bastante cómodo con JavaScript, primero eche un vistazo al Capítulo 3 como un breve vistazo de lo que puede esperar de YDKJS, luego salte siga adelante!

1.1 Código

Empecemos desde el principio.

Un programa, a menudo denominado código fuente o código, es un conjunto de instrucciones especiales para indicar al equipo qué tareas realizar. Normalmente, el código se guarda en un archivo de texto, aunque con JavaScript también se puede escribir código directamente en una consola de desarrollador en un navegador, que veremos en breve.

Las reglas para el formato válido y las combinaciones de instrucciones se llaman un lenguaje informático, a veces se refiere como su sintaxis, lo mismo que el inglés te dice cómo deletrear palabras y cómo crear oraciones válidas usando palabras y signos de puntuación.

Declaraciones

En un lenguaje de computadora, un grupo de palabras, números y operadores que realiza una tarea específica es una sentencia/declaración. En JavaScript, una declaración podría verse como sigue:

```
a = b * 2;
```

Los caracteres `a` y `b` se llaman variables (vea "Variables"), que son como simples cajas en las que puede almacenar cualquiera de sus cosas. En los programas, las variables contienen valores (como el número 42) que debe usar el programa. Piense en ellos como marcadores de posición simbólicos para los valores mismos.

Por el contrario, el 2 es sólo un valor en sí mismo, llamado un valor literal, porque está solo sin ser almacenado en una variable.

Los caracteres `=` y `*` son operadores (ver "Operadores") - realizan acciones con valores y variables tales como asignación y multiplicación matemática.

La mayoría de las declaraciones en JavaScript terminan con un punto y coma (`;`) al final.

La afirmación `a = b * 2` ; Le dice a la computadora, aproximadamente, que obtenga el valor actual almacenado en la variable `b` , multiplique ese valor por 2, y luego guarde el resultado en otra variable a la que llamamos `a` .

Los programas son sólo colecciones de muchas declaraciones de este tipo, que en conjunto describen todos los pasos que se requieren para realizar el propósito de su programa.

Expresiones

Las declaraciones se componen de una o más expresiones. Una expresión es cualquier referencia a una variable o valor, o un conjunto de variables y valores combinados con operadores.

Por ejemplo:

```
a = b * 2;
```

Esta declaración tiene cuatro expresiones en ella:

2 es una expresión de valor literal

b es una expresión variable, lo que significa recuperar su valor actual

b * 2 es una expresión aritmética, que significa hacer la multiplicación

a = b * 2 es una expresión de asignación, lo que significa asignar el resultado de la expresión b * 2 a la variable a (sobre las asignaciones posteriores)

Una expresión general que se mantiene sola también se denomina declaración de expresión, como la siguiente:

```
b * 2;
```

Esta declaración de una expresión no es muy común o útil, ya que en general no tendría ningún efecto en el funcionamiento del programa: recuperaría el valor de b y lo multiplicaría por 2, pero luego no haría nada con ese resultado.

Una sentencia de expresión más común es una instrucción de expresión de llamada (véase "Funciones"), ya que la sentencia completa es la expresión de llamada de la función en sí:

```
alert( a );
```

Ejecutar un programa

¿De qué manera esas colecciones de declaraciones de programación le dicen a la computadora qué hacer? El programa necesita ser ejecutado, también conocido como ejecutar el programa.

Declaraciones como `a = b * 2` son útiles para los desarrolladores al leer y escribir, pero en realidad no están en una forma en que la computadora pueda entender directamente. Así que una utilidad especial en la computadora (ya sea un intérprete o un compilador) se utiliza

para traducir el código que escribe en comandos que una computadora puede entender.

Para algunos lenguajes de computadora, esta traducción de los comandos se hace típicamente de arriba a abajo, línea por línea, cada vez que se ejecuta el programa, se llama generalmente interpretar el código.

Para otros idiomas, la traducción se hace antes de tiempo, llamada compilación del código, por lo que cuando el programa se ejecuta más tarde, lo que está en ejecución es en realidad las instrucciones de la computadora ya compilada listo para ejecutarse.

Normalmente, se afirma que JavaScript se interpreta, porque el código fuente de JavaScript se procesa cada vez que se ejecuta. Pero eso no es del todo exacto. El motor de JavaScript en realidad compila el programa sobre la marcha y luego ejecuta inmediatamente el código compilado.

Nota: Para obtener más información sobre la compilación de JavaScript, consulte los dos primeros capítulos del título Scope & Closures de esta serie.

1.2 Inténtalo tú mismo

Este capítulo va a introducir cada concepto de programación con simples fragmentos de código, todos escritos en JavaScript (obviamente!).

No se puede enfatizar lo suficiente: mientras que usted va a través de este capítulo - y usted necesitará darle el tiempo para repasarlo varias veces - usted debe practicar cada uno de estos conceptos escribiendo el código usted mismo. La forma más sencilla de hacerlo es abrir la consola de herramientas de desarrollador en el navegador más cercano (Firefox, Chrome, IE, etc.).

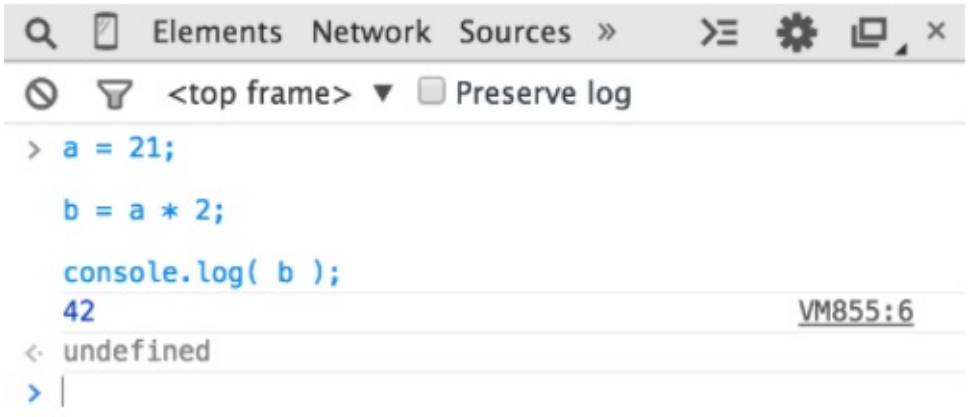
Sugerencia: Normalmente, puede iniciar la consola del programador con un acceso directo de teclado o desde un elemento de menú. Para obtener información más detallada acerca del inicio y el uso de la consola en su navegador favorito, consulte "Dominar de la consola de herramientas de desarrollo" (http://blog.teamtreehouse.com/mastering-developer-tools-console). Para escribir varias líneas en la consola a la vez, use <shift> + <enter> para pasar a la nueva línea siguiente. Una vez que pulse <enter> por sí mismo, la consola ejecutará todo lo que acaba de escribir.

Vamos a familiarizarnos con el proceso de ejecución de código en la consola. En primer lugar, sugiero abrir una pestaña vacía en su navegador. Prefiero hacer esto escribiendo: en blanco en la barra de direcciones. A continuación, asegúrese de que su consola de desarrollo está abierta, como acabamos de mencionar.

Ahora, escriba este código y vea cómo se ejecuta:

```
a = 21;

b = a * 2;

console.log( b );
```

Escribir el código anterior en la consola en Chrome debería producir algo como lo siguiente:



Vamos, prueba. La mejor manera de aprender la programación es empezar a codificar!

Salida

En el fragmento de código anterior, utilizamos console.log (..). En pocas palabras, veamos de qué se trata esa línea de código.

Usted puede haber adivinado, pero eso es exactamente cómo imprimimos el texto (aka salida al usuario) en la consola del navegador. Hay dos características de esa afirmación que debemos explicar.

En primer lugar, la parte log (b) se denomina llamada de función (véase "Funciones"). Lo que pasa es que estamos entregando la variable b a esa función, que le pide tomar el valor de b e imprimirlo en la consola.

En segundo lugar, la parte console es una referencia de objeto donde se encuentra la función log (..). Cubrimos los objetos y sus propiedades con más detalle en el Capítulo 2.

Otra forma de crear resultados que puede ver es ejecutar una instrucción de alert(..). Por ejemplo:

```
alert( b );
```

Si ejecuta eso, notará que en lugar de imprimir la salida a la consola, muestra un cuadro emergente "Aceptar" con el contenido de la variable b. Sin embargo, usar console.log (...) generalmente hará que el aprendizaje sobre la codificación y ejecución de sus programas en la consola sea más fácil que usar la alert (..), ya que puede generar muchos valores a la vez sin interrumpir la interfaz del navegador.

Para este libro, usaremos console.log (..) para la salida.

Entrada

Mientras estamos discutiendo la salida, también puede preguntarse sobre la entrada (es decir, recibir información del usuario).

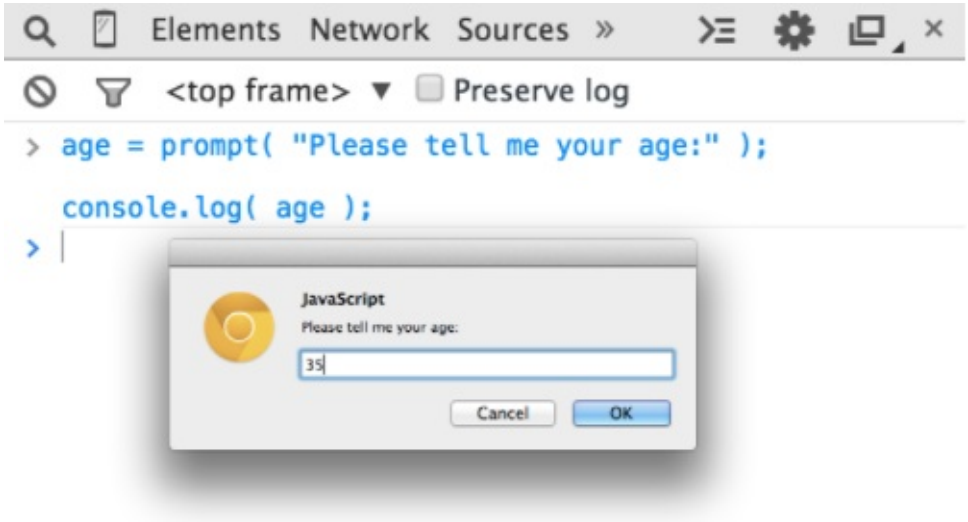
La forma más común que ocurre es que la página HTML muestre elementos de formulario (como cuadros de texto) en un usuario al que puedan escribir y, a continuación, utilizar JS para leer esos valores en las variables del programa.

Pero hay una manera más fácil de obtener información para fines sencillos de aprendizaje y demostración, como la que estará haciendo a lo largo de este libro. Utilice la función de prompt (..):

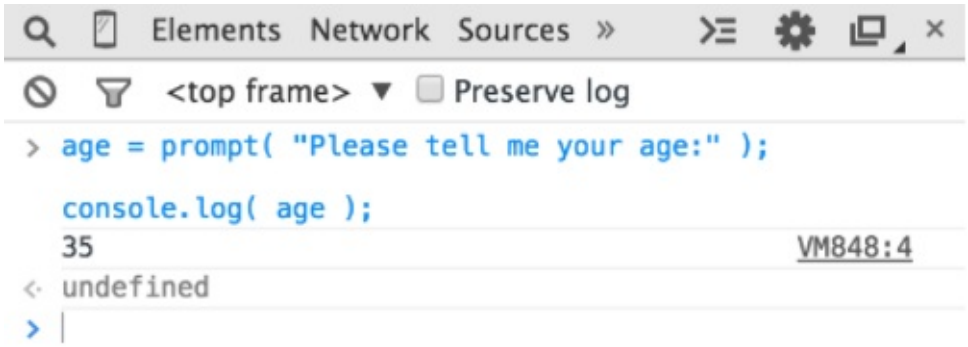
```
age = prompt( "Please tell me your age:" );

console.log( age );
```

Como ya habrás adivinado, el mensaje que pasas al prompt (..) - en este caso, "Por favor, dime tu edad:" - se imprime en el popup.

Esto debería ser similar a lo siguiente:



Una vez que envíe el texto de entrada haciendo clic en "Aceptar", observará que el valor que escribió se almacena en la variable de edad, que luego se emite con console.log (..):



Para mantener las cosas simples mientras estamos aprendiendo conceptos básicos de programación, los ejemplos en este libro no requerirán entrada. Pero ahora que has visto cómo usar el prompt (..), si quieres desafiarte puedes intentar usar la entrada en tus exploraciones de los ejemplos.

1.3 Operadores

Los operadores son la forma en que realizamos acciones sobre variables y valores. Ya hemos visto dos operadores de JavaScript, el `=` y el `*`.

El operador `*` realiza la multiplicación matemática. Lo suficientemente simple, ¿no?

El operador `=` igual se utiliza para la asignación - primero calculamos el valor en el lado derecho (valor fuente) de la = y luego ponerlo en la variable que especificamos en el lado izquierdo (variable objetivo).

Advertencia: Esto puede parecer un orden inverso extraño para especificar asignación. En lugar de a = 42, algunos podrían preferir voltear el orden para que el valor de origen esté a la izquierda y la variable de destino a la derecha, como 42 -> a (JavaScript no válido). Desafortunadamente, la forma ordenada a = 42, y variaciones similares, es bastante frecuente en los lenguajes de programación modernos. Si se siente antinatural, sólo pase algún tiempo ensayando y el orden en su mente lo acostumbrará a ella.

Considere:

```
a = 2;
b = a + 1;
```

Aquí, asignamos el valor 2 a la variable a. A continuación, obtenemos el valor de la variable a (todavía 2), agregamos 1 a ella resultando en el valor 3, luego almacenamos ese valor en la variable b.

Si bien técnicamente no es un operador, necesitará la palabra clave `var` en cada programa, ya que es la forma principal de declarar (aka crear) variables (consulte "Variables").

Siempre debe declarar la variable por nombre antes de usarla. Pero sólo es necesario declarar una variable una vez para cada ámbito (ver "Alcance"); Se puede utilizar tantas veces como sea necesario. Por ejemplo:

```
var a = 20;

a = a + 1;
a = a * 2;

console.log( a );    // 42
```

Éstos son algunos de los operadores más comunes en JavaScript:

- Asignación: = como en a = 2.
- Matemáticas: + (adición), - (sustracción), * (multiplicación) y / (división), como en a * 3.
- Asignación Compuesta : + =, - =, * =, y / = son operadores compuestos que combinan una operación matemática con asignación, como en a + = 2 (igual que a = a + 2).
- Incremento / Decremento: ++ (incremento), - (decremento), como en a ++ (similar a a = a + 1).
- Acceso a la propiedad del objeto:. Como en console.log (). Los objetos son valores que contienen otros valores en determinadas ubicaciones denominadas propiedades. `obj.a` significa un valor de objeto llamado obj con una propiedad del nombre a. Las propiedades se pueden acceder alternativamente como obj ["a"]. Consulte el Capítulo 2.
- Igualdad: == (suelto-igual), === (estricto-igual),! = (Suelto no-igual),! == (estricto no igual), como en a == b.
- Consulte "Valores y Tipos" y el Capítulo 2.
- Comparación: <(menor que),> (mayor que), <= (menor que o suelto-igual),> = (mayor que o suelto-igual), como en a <= b.
- Consulte "Valores y Tipos" y el Capítulo 2.
- Lógico: && (y), || (O), como en a || b que selecciona a o b. Estos operadores se usan para expresar condicionales compuestos (ver "Condicionales"), como si a o b es verdadero.

Nota: Para obtener más detalles y cobertura de los operadores que no se mencionan aquí, consulte las "Expresiones y operadores" de Mozilla Developer Network (MDN) (https://developer.mozilla.org/en-US/docs/Web/JavaScript / Guía / Expresiones_y_Operadores).

1.4 Valores y Tipos

Si le preguntas a un empleado en una tienda de teléfono cuánto cuesta un teléfono determinado, y dicen "noventa y nueve, noventa y nueve" (es decir, $ 99.99), te están dando una cifra numérica real que representa lo que vas a necesitar pagar (más impuestos) para comprarlo. Si usted quiere comprar dos de esos teléfonos, puede hacer fácilmente las matemáticas mentales para duplicar ese valor para obtener $ 199.98 por su costo base.

Si ese mismo empleado coge otro teléfono similar, pero dice que es "gratis", no le están dando un número, sino otro tipo de representación de su costo esperado ($ 0.00) - la palabra "free ."

Cuando más tarde pregunte si el teléfono incluye un cargador, esa respuesta sólo podría haber sido "sí" o "no".

De manera muy similar, cuando expresa valores en un programa, elige diferentes representaciones para esos valores en función de lo que planea hacer con ellos.

Estas diferentes representaciones de los valores se denominan tipos en terminología de programación. JavaScript tiene tipos incorporados para cada uno de estos llamados valores primitivos:

- Cuando necesitas hacer matemáticas, quieres un número.
- Cuando necesite imprimir un valor en la pantalla, necesita una cadena (uno o más caracteres, palabras, oraciones).
- Cuando necesite tomar una decisión en su programa, necesita un booleano (verdadero o falso).

Los valores que se incluyen directamente en el código fuente se llaman literales. Los literales de cadenas están rodeados por comillas dobles "..." o comillas simples ('...') - la única diferencia es la preferencia estilística. El número y los literales booleanos se presentan como es (es decir, 42, true, etc.).

Considere:

```
"I am a string";
'I am also a string';

42;

true;
false;
```

Más allá de los tipos de valores string / number / boolean, es común que los lenguajes de programación proporcionen arrays, objetos, funciones y más. Cubriremos mucho más sobre valores y tipos a lo largo de este capítulo y el siguiente.

Conversión entre tipos

Si tienes un número pero necesitas imprimirlo en la pantalla, necesitas convertir el valor en una cadena, y en JavaScript esta conversión se llama "coerción". Del mismo modo, si alguien introduce una serie de caracteres numéricos en un formulario en una página de comercio electrónico, es una cadena, pero si necesita utilizar ese valor para realizar operaciones matemáticas, debe coaccionarla a un número.

JavaScript proporciona varias facilidades diferentes para forzar la coerción entre tipos. Por ejemplo:

```
var a = "42";
var b = Number( a );

console.log( a );    // "42"
console.log( b );    // 42
```

El uso de Number (..) (una función incorporada) como se muestra es una coerción explícita de cualquier otro tipo al tipo de número. Eso debería ser bastante sencillo.

Pero un tema polémico es lo que sucede cuando se intenta comparar dos valores que ya no son del mismo tipo, lo que requeriría coerción implícita.

Al comparar la cadena "99.99" con el número 99.99, la mayoría de la gente estaría de acuerdo en que son equivalentes. Pero no son exactamente iguales, ¿verdad? Es el mismo valor en dos representaciones diferentes, dos tipos diferentes. Se podría decir que son "vagamente iguales", ¿no?

Para ayudarte en estas situaciones comunes, JavaScript a veces se activa y obliga implícitamente a los valores a los tipos coincidentes.

Así que si usas el operador == loose igual para hacer la comparación "99.99" == 99.99, JavaScript convertirá el lado izquierdo "99.99" a su número equivalente 99.99. La comparación entonces se convierte en 99.99 == 99.99, que es por supuesto verdad.

Si bien está diseñado para ayudarlo, la coerción implícita puede crear confusión si no se ha tomado el tiempo para aprender las reglas que rigen su comportamiento. La mayoría de los desarrolladores de JS nunca lo tienen, por lo que la sensación común es que la coerción implícita es confusa y daña los programas con errores inesperados, por lo que debe evitarse. Incluso a veces se llama un defecto en el diseño del lenguaje.

Sin embargo, la coerción implícita es un mecanismo que se puede aprender, y además debe ser aprendido por cualquiera que desee tomar la programación de JavaScript en serio. ¡No sólo no confunde una vez que aprende las reglas, él puede realmente hacer sus programas mejores! El esfuerzo vale la pena.

Nota: Para obtener más información sobre la coerción, consulte el Capítulo 2 de este título y el Capítulo 4 del título de Tipos y Gramática de esta serie.

1.5 Comentarios del Código

El empleado de la tienda telefónica puede anotar algunas notas sobre las características de un teléfono recién lanzado o sobre los nuevos planes que su empresa ofrece. Estas notas son sólo para el empleado - no son para los clientes leer. Sin embargo, estas notas ayudan al empleado a hacer su trabajo mejor documentando de los comos y los porqués de lo que ella debe decir a los clientes.

Una de las lecciones más importantes que puede aprender sobre la escritura de código es que no es sólo para la computadora. El código es mucho más, si no más, para el desarrollador tanto como lo es para el compilador.

Su computadora sólo se preocupa por el código de máquina, una serie de 0s y 1s binarios, que viene de la compilación. Hay un número casi infinito de programas que podría escribir que producen la misma serie de 0s y 1s. Las opciones que hagas sobre cómo escribir tu programa son importantes, no sólo para ti, sino para los demás miembros del equipo e incluso para tu futuro yo.

Usted debe esforzarse no sólo en escribir programas que funcionen correctamente, sino programas que tienen sentido cuando se examinan. Puede recorrer un largo camino en ese esfuerzo eligiendo buenos nombres para sus variables (ver "Variables") y funciones (ver "Funciones").

Pero otra parte importante es los comentarios del código. Éstos son pedacitos de texto en su programa que se insertan puramente para explicar cosas a un ser humano. El intérprete / compilador siempre ignorará estos comentarios.

Hay un montón de opiniones sobre lo que debe ser un buen código comentado; No podemos definir reglas universales absolutas. Pero algunas observaciones y directrices son muy útiles:

- El código sin comentarios es subóptimo.
- Demasiados comentarios (uno por línea, por ejemplo) es probablemente un signo de código mal escrito.
- Los comentarios deben explicar por qué, no qué hacen. Opcionalmente pueden explicar cómo si eso es particularmente confuso.

En JavaScript, hay dos tipos de comentarios posibles: un comentario de una sola línea y un comentario de varias líneas.

Considere:

```
// This is a single-line comment

/* But this is
       a multiline
             comment.
                      */
```

El // comentario de una sola línea es apropiado si vas a poner un comentario justo encima de una sola sentencia, o incluso al final de una línea. Todo en la línea después de // se trata como el comentario (por lo tanto es ignorado por el compilador), todo el camino hasta el final de la línea. No hay ninguna restricción a lo que puede aparecer dentro de un comentario de una sola línea.

Considere:

```
var a = 42;        // 42 is the meaning of life
```

El comentario de / * .. * / multiline es apropiado si tienes varias líneas de explicación para hacer en tu comentario.

Aquí está un uso común de comentarios multilínea:

```
/* The following value is used because
   it has been shown that it answers
   every question in the universe. */
var a = 42;
```

También puede aparecer en cualquier parte de una línea, incluso en el centro de una línea, porque el * / lo termina. Por ejemplo:

```
var a = /* arbitrary value */ 42;

console.log( a );    // 42
```

Lo único que no puede aparecer dentro de un comentario de varias líneas es un * /, porque eso sería interpretado para finalizar el comentario.

Usted definitivamente querrá comenzar su aprendizaje de programación comenzando con el hábito de comentar el código. A lo largo del resto de este capítulo, verás que utilizo los comentarios para explicar las cosas, así que haz lo mismo en tu propia práctica. Confía en mí, todo el mundo que lea su código se lo agradecerá!

1.6 Variables

La mayoría de los programas útiles necesitan rastrear un valor a medida que cambia a lo largo del programa, pasando por diferentes operaciones según lo solicitado por las tareas previstas de su programa.

La forma más sencilla de hacerlo en su programa es asignar un valor a un contenedor simbólico, llamado variable, llamado así porque el valor en este contenedor puede variar con el tiempo según sea necesario.

En algunos lenguajes de programación, se declara una variable (contenedor) para contener un tipo específico de valor, como número o cadena. La tipificación estática, también conocida como ejecución de tipo, se cita típicamente como un beneficio para la corrección del programa al evitar conversiones de valor no deseadas.

Otros idiomas enfatizan tipos para valores en lugar de variables. La tipificación débil, también conocida como escritura dinámica, permite que una variable contenga cualquier tipo de valor en cualquier momento. Se suele citar como un beneficio para la flexibilidad del programa al permitir que una única variable represente un valor sin importar la forma de tipo que el valor pueda tomar en un momento dado en el flujo lógico del programa.

JavaScript utiliza el último enfoque, el tipo dinámico, lo que significa que las variables pueden contener valores de cualquier tipo sin ningún tipo de aplicación.

Como se mencionó anteriormente, declaramos una variable usando la instrucción `var` - note que no hay otra información de tipo en la declaración. Considere este sencillo programa:

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );        // 199.98

// convert `amount` to a string, and
// add "$" on the beginning
amount = "$" + String( amount );

console.log( amount );        // "$199.98"
```

La variable `amount` comienza teniendo el número 99.99, y luego tiene el resultado del número de la cantidad * 2, que es 199.98.

El primer comando console.log (..) tiene que coaccionar implícitamente ese valor numérico a una cadena para imprimirlo.

Entonces el valor de la sentencia `= "$" + String (amount)` coacciona explícitamente el valor 199.98 a una cadena y agrega un carácter "$" al principio. En este punto, `amount` ahora tiene el valor de cadena "$ 199.98", por lo que la segunda instrucción console.log (..) no necesita hacer ninguna coerción para imprimirlo.

Los desarrolladores de JavaScript notarán la flexibilidad de usar la variable `amount` para cada uno de los valores de 99.99, 199.98 y "$ 199.98". Los entusiastas de la tipificación estática preferirían una variable independiente como `amountStr` para mantener la representación final de "$ 199.98" del valor, porque es un tipo diferente.

De cualquier manera, notará que `amount` tiene un valor de ejecución que cambia a lo largo del programa, ilustrando el propósito principal de las variables: administrar el estado del programa.

En otras palabras, `state` es el seguimiento de los cambios a los valores a medida que su programa se ejecuta.

Otro uso común de variables es para centralizar la configuración de valores. Esto se llama más comúnmente constantes, cuando usted declara una variable con un valor y la intención es que el valor no cambie a través del programa.

Usted declara estas constantes, a menudo en la parte superior de un programa, por lo que es conveniente para usted tener un lugar para ir a alterar un valor si es necesario. Por convención, las variables JavaScript como constantes suelen ser mayúsculas, con guiones bajos _ entre varias palabras.

He aquí un ejemplo tonto:

```
var TAX_RATE = 0.08;    // 8% sales tax

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );                // 215.9784
console.log( amount.toFixed( 2 ) );    // "215.98"
```

Nota: Similar a cómo `console.log (..)` es un registro de `function(..)` que se accede como una propiedad de objeto en el valor de la consola, `toFixed(..)` aquí es una función a la que se puede acceder en valores numéricos. Los números de JavaScript no se formatean

automáticamente para dólares - el motor no sabe cuál es su intención y no hay un tipo de moneda. `ToFixed(..)` nos permite especificar cuántos números decimales queremos que el número sea redondeado, y produce la cadena según sea necesario.

La variable `TAX_RATE` es sólo constante por convención - no hay nada especial en este programa que impide que se cambie. Pero si la ciudad eleva la tasa del impuesto a las ventas al 9%, podemos actualizar fácilmente nuestro programa estableciendo el valor `TAX_RATE` asignado a 0,09 en un lugar, en lugar de buscar y encontrar muchas ocurrencias del valor 0.08 puesto a lo largo del programa y actualizandolo en todas ellas .

La versión más reciente de JavaScript en el momento de escribir este documento (comúnmente llamado "ES6") incluye una nueva forma de declarar constantes, utilizando `const` en lugar de `var` :

```
// as of ES6:
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

Las constantes son útiles al igual que las variables con valores sin cambios, excepto que las constantes también evitan el cambio accidental de valor en algún otro lugar después de la configuración inicial. Si intentó asignar un valor diferente a `TAX_RATE` después de esa primera declaración, su programa rechazaría el cambio (y en modo estricto, fallará con un error; consulte "Modo estricto" en el Capítulo 2).

Por cierto, ese tipo de "protección" contra los errores es similar al tipo de tipificación estática, por lo que puede ver por qué los tipos estáticos en otros idiomas pueden ser atractivos!

Nota: Para obtener más información sobre cómo se pueden utilizar diferentes valores en las variables en sus programas, consulte el título Tipos y Gramática de esta serie.

1.7 Bloques

El cliente de la tienda telefónica debe pasar por una serie de pasos para completar la compra mientras compra su nuevo teléfono.

Del mismo modo, en código a menudo es necesario agrupar una serie de declaraciones en conjunto, que a menudo llamamos un bloque. En JavaScript, un bloque se define envolviendo una o más instrucciones dentro de un par de llaves `{..}` . Considere:

```
var amount = 99.99;

// un bloque general
{
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

Este tipo de bloque independiente general `{..}` es válido, pero no es tan comúnmente visto en los programas de JS. Por lo general, los bloques se adjuntan a alguna otra instrucción de control, como una instrucción `if` (consulte "Condicionales") o un bucle (consulte "Loops"). Por ejemplo:

```
var amount = 99.99;

// is amount big enough?
if (amount > 10) {            // <-- block attached to `if`
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

Vamos a explicar las declaraciones `if` en la siguiente sección, pero como puede ver, el bloque `{..}` con sus dos declaraciones se adjunta a `if (amount > 10)` ; Las declaraciones dentro del bloque sólo se procesarán si pasa el condicional.

Nota: A diferencia de la mayoría de otras sentencias como `console.log(amount);` , una instrucción de bloque no necesita un punto y coma (;) para concluirlo.

1.8 Condicionales

"¿Desea agregar los protectores de pantalla adicionales a su compra, por $ 9.99?" El empleado de la tienda telefónica le ha pedido que tome una decisión. Y puede que necesite consultar primero el estado actual de su cartera o cuenta bancaria para responder a esa pregunta. Pero, obviamente, esto es sólo una simple pregunta "sí o no".

Hay bastantes maneras de expresar condicionales (aka decisiones) en nuestros programas.

La más común es la instrucción `if`. Esencialmente, usted está diciendo: "Si esta condición es cierta, haga lo siguiente ...". Por ejemplo:

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

La instrucción `if` requiere una expresión entre los paréntesis `()` que puede tratarse como verdadero o falso. En este programa, hemos proporcionado la expresión `amount < bank_balance`, que de hecho se evaluará a `true` o `false` dependiendo de la cantidad en la variable `bank_balance`.

Incluso puede proporcionar una alternativa si la condición no es verdadera, llamada cláusula `else`. Considere:

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// can we afford the extra purchase?
if ( amount < bank_balance ) {
    console.log( "I'll take the accessory!" );
    amount = amount + ACCESSORY_PRICE;
}
// otherwise:
else {
    console.log( "No, thanks." );
}
```

Aquí, si `amount < bank_balance` es `true`, imprimiremos "¡Tomaré el accesorio!" Y agregue 9.99 a nuestra variable `amount`. De lo contrario, la cláusula `else` dice que simplemente responderemos cortésmente con "No, gracias". Y dejar la cantidad sin cambios.

Como discutimos en "Valores y Tipos" anteriormente, los valores que no son ya de un tipo esperado son a menudo coaccionados a ese tipo. La sentencia `if` espera un booleano, pero si se pasa algo que no sea ya booleano, la coerción se producirá.

JavaScript define una lista de valores específicos que se consideran "falseados" porque cuando se coaccionan a un booleano, se convierten en falsos - estos incluyen valores como `0` y `""`. Cualquier otro valor que no esté en la lista de "falsos" es automáticamente "verdad" - cuando se coacciona a un booleano se convierten en verdaderos. Los valores Truthy incluyen cosas como 99.99 y "free". Vea "Truthy & Falsy" en el Capítulo 2 para más información.

Los condicionales existen en otras formas además del `if`. Por ejemplo, la instrucción `switch` puede usarse como una abreviatura para una serie de instrucciones `if..else` (vea el Capítulo 2). Los bucles (ver "Loops") usan un condicional para determinar si el bucle debe continuar o detenerse.

Nota: Para obtener información más detallada sobre las coerciones que pueden ocurrir implícitamente en las expresiones de prueba de condicionales, consulte el Capítulo 4 del título de Tipos y Gramática de esta serie.

1.9 Bucles

Durante las horas pico, hay una lista de espera para los clientes que necesitan hablar con el empleado de la tienda telefónica. Aunque todavía hay personas en esa lista, sólo tiene que seguir sirviendo al próximo cliente.

Repetir un conjunto de acciones hasta que cierta condición falla - en otras palabras, repetir sólo mientras la condición se mantiene - es el trabajo de los bucles de programación; Los bucles pueden tomar diferentes formas, pero todos satisfacen este comportamiento básico.

Un bucle incluye la condición de prueba, así como un bloque (normalmente como `{..}`). Cada vez que se ejecuta el bloque del bucle, se llama iteración.

Por ejemplo, el bucle `while` y las formas de bucle `do..while` ilustran el concepto de repetir un bloque de sentencias hasta que una condición ya no se evalúa como verdadera:

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
}

// versus:

do {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

La única diferencia práctica entre estos bucles es si el condicional se prueba antes de la primera iteración (`while`) o después de la primera iteración (`do..while`).

De cualquier forma, si las pruebas condicionales son falsas, la siguiente iteración no se ejecutará. Esto significa que si la condición es inicialmente falsa, un bucle while nunca se ejecutará, pero un bucle `do..while` se ejecutará sólo la primera vez.

A veces usted está haciendo un bucle con el propósito de contar un cierto conjunto de números, como de 0 a 9 (diez números). Puede hacerlo estableciendo una variable de iteración de bucle como `i` en el valor `0` e incrementándola en `1` cada iteración.

Advertencia: Por una variedad de razones históricas, los lenguajes de programación casi siempre cuentan las cosas de manera cero, es decir, comenzando con 0 en lugar de 1. Si no está familiarizado con ese modo de pensar, puede ser muy confuso al principio. Tómese su tiempo para practicar el conteo empezando por 0 para sentirse más cómodo con él.

El condicional se prueba en cada iteración, como si hubiera una declaración implícita `if` dentro del bucle.

Podemos usar la instrucción `break` de JavaScript para detener un bucle. También, podemos observar que es extremadamente fácil crear un bucle que de otra manera funcionaría para siempre sin un mecanismo para detenerla.

Vamos a ilustrar esto:

```
var i = 0;

// a `while..true` loop would run forever, right?
while (true) {
    // stop the loop?
    if ((i <= 9) === false) {
        break;
    }

    console.log( i );
    i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

Advertencia: esto no es una forma práctica en la que desearía utilizar sus bucles. Se presenta aquí sólo con fines ilustrativos.

Mientras que un `while` (o `do..while`) puede realizar la tarea manualmente, hay otra forma sintáctica llamada un bucle `for` sólo para ese propósito:

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

Como puede ver, en ambos casos el condicional `i <= 9` es verdadero para las primeras 10 iteraciones (`i` de valores de 0 a 9) de cualquier forma de bucle, pero se convierte en falso una vez que `i` es un valor 10.

El bucle `for` tiene tres cláusulas: la cláusula de inicialización (`var i = 0`), la cláusula de prueba condicional (`i <= 9`) y la cláusula de actualización (`i = i + 1`). Así que si vas a contar con tus iteraciones de bucle, es una forma más compacta ya menudo más fácil de

entender y escribir.

Existen otros formularios de bucle especializados que tienen la intención de iterar sobre valores específicos, como las propiedades de un objeto (véase el capítulo 2), donde la prueba condicional implícita es justamente si se han procesado todas las propiedades. El concepto de "bucle hasta que falla una condición" no importa cuál sea la forma del bucle.

1.10 Funciones

El empleado de la tienda de teléfonos probablemente no lleva consigo una calculadora para calcular los impuestos y la cantidad final de la compra. Esa es una tarea que necesita definir una vez y reutilizar una y otra vez. Las probabilidades son, que la empresa tenga un registro de pago (computadora, tableta, etc.) con esas "funciones" incorporadas.

Del mismo modo, su programa casi seguramente querrá dividir las tareas del código en piezas reutilizables, en lugar de repetir repetidamente repetidamente (juego de palabras!). La forma de hacerlo es definir una función.

Una función es generalmente una sección con el nombre del código que puede ser "llamada" por ese nombre, y el código dentro de ella se ejecutará cada vez que se llama. Considere:

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

Opcionalmente, las funciones pueden tomar argumentos (también conocidos como parámetros), valores que se le "pasan". Además, también pueden devolver un valor.

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );        // "199.98"

amount = formatAmount();
console.log( amount );            // "$99.99"
```

La función `printAmount(..)` toma un parámetro que llamamos `amt`. La función `formatAmount()` devuelve un valor. Por supuesto, también puedes combinar esas dos técnicas en la misma función.

Las funciones se usan a menudo para el código que se planea llamar varias veces, pero también pueden ser útiles sólo para organizar fragmentos de código relacionados en las colecciones con nombre, incluso si sólo planea llamarlas una vez.

Considere:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );        // "107.99"
```

Aunque `calculateFinalPurchaseAmount(..)` sólo se llama una vez, organizar su comportamiento en una función denominada separadamente hace que el código que utiliza su lógica (la declaración `amount = calculateFinal...`) más limpia. Si la función tenía más declaraciones en ella, los beneficios serían aún más pronunciados.

1.11 Scope (Ámbito)

Si le pregunta al empleado de la tienda de teléfonos por un modelo de teléfono que su tienda no tiene, no podrá venderle el teléfono que desea. Sólo tiene acceso a los teléfonos en el inventario de su tienda. Tendrás que buscar otra tienda para ver si puedes encontrar el teléfono que estás buscando.

La programación tiene un término para este concepto: scope (técnicamente llamado ámbito léxico). En JavaScript, cada función obtiene su propio ámbito. El scope es básicamente una colección de variables, así como las reglas de cómo se accede a esas variables por nombre. Sólo el código dentro de esa función puede acceder a las variables de ámbito de la función.

Un nombre de variable tiene que ser único dentro del mismo ámbito - no puede haber dos variables diferentes una al lado del otro. Pero el mismo nombre de variable podría aparecer en diferentes ámbitos.

```
function one() {
    // ésta `a` solo pertenece a la funcion `one()`
    var a = 1;
    console.log( a );
}

function two() {
    // ésta `a` solo pertenece a la funcion `two()`
    var a = 2;
    console.log( a );
}

one();        // 1
two();        // 2
```

También, un scope puede estar anidado dentro de otro scope, apenas como si un payaso en una fiesta del cumpleaños sopla encima de un globo dentro de otro globo. Si un scope está anidado dentro de otro, el código dentro del ámbito más interno puede acceder a las variables desde cualquiera de los ámbitos.

Considere:

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;

        // puede acceder a ambos `a` y `b` aqui
        console.log( a + b );    // 3
    }

    inner();

    // solo puede acceder a `a` aqui
    console.log( a );            // 1
}

outer();
```

Las reglas de ámbito léxico dicen que el código en un ámbito puede acceder a la variable `a` de ese ámbito o de cualquier ámbito fuera de él.

Por lo tanto, el código dentro de la función `inner()` tiene acceso a las variables `a` y `b`, pero el código en `outer()` sólo tiene acceso a `a` - no puede acceder a `b` porque esa variable sólo está dentro de `inner()`.

Recuerde este fragmento de código anterior:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}
```

La constante `TAX_RATE` (variable) es accesible desde dentro de la función `calculateFinalPurchaseAmount(..)`, aunque no la pasamos, debido al alcance léxico.

Nota: Para obtener más información sobre el ámbito léxico, consulte los tres primeros capítulos del título Scope & Closures de esta serie.

1.12 Práctica

No hay absolutamente ningún sustituto para la práctica en el aprendizaje de la programación. Ninguna cantidad de escritura articulada de mi parte hará por si sola de usted un buen programador.

Con esto en mente, intentemos practicar algunos de los conceptos que hemos aprendido aquí en este capítulo. Daré los "requisitos", y lo intentarás primero. Luego consulte la lista de códigos que hay a continuación para ver cómo me llegué a ella.

- Escriba un programa para calcular el precio total de la compra de su teléfono. Usted continuará comprando teléfonos (sugerencia: loop!) Hasta que se quede sin dinero en su cuenta bancaria. También comprará accesorios para cada teléfono, siempre y cuando su cantidad de compra esté por debajo de su umbral de gasto mental.
- Después de haber calculado el importe de su compra, agregue el impuesto y, a continuación, imprima el importe de compra calculado, correctamente formateado.
- Por último, compruebe la cantidad en contra del saldo de su cuenta bancaria para ver si se lo puede permitir o no.
- Debe establecer algunas constantes para la "tasa de impuesto", "precio de teléfono", "precio de accesorio" y "umbral de gasto", así como una variable para su "saldo de cuenta bancaria". "
- Debe definir funciones para calcular el impuesto y para formatear el precio con un "$" y redondear a dos decimales.
- Desafío de bonificación: Trate de incorporar la entrada a este programa, tal vez con el `prompt(..)` cubierto en la "Entrada" anterior. Por ejemplo, puede solicitar al usuario el saldo de su cuenta bancaria. ¡Diviértase y sea creativo!

OK, adelante. Intentalo. ¡No eche un vistazo a mi lista de códigos hasta que lo haya intentado usted mismo!

Nota: Debido a que este es un libro de JavaScript, obviamente voy a resolver el ejercicio de práctica en JavaScript. Pero usted puede hacerlo en otro idioma por ahora si se siente más cómodo.

Aquí está mi solución de JavaScript para este ejercicio:

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// keep buying phones while you still have money
while (amount < bank_balance) {
    // buy a new phone!
    amount = amount + PHONE_PRICE;

    // can we afford the accessory?
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// don't forget to pay the government, too
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// can you actually afford this purchase?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// You can't afford this purchase. :(
```

Nota: La forma más sencilla de ejecutar este programa JavaScript es escribirlo en la consola de desarrollo de su navegador más cercano.

¿Como hiciste? No haría daño intentarlo otra vez ahora que usted ha visto mi código. Y jugar con el cambio de algunas de las constantes para ver cómo se ejecuta el programa con diferentes valores.

1.13 - Revisión

El aprendizaje de programación no tiene que ser un proceso complejo y abrumador. Sólo hay algunos conceptos básicos que necesita retener en la cabeza.

Estos actúan como bloques de construcción. Para construir una torre alta, empiece primero colocando el bloque en la parte superior del bloque en la parte superior del bloque. Lo mismo ocurre con la programación. Éstos son algunos de los elementos fundamentales de la programación:

- Necesita que los operadores realicen acciones sobre valores.
- Necesita valores y tipos para realizar diferentes tipos de acciones como matemáticas en números o salida con cadenas.
- Necesita variables para almacenar datos (también conocido como estado) durante la ejecución de su programa.
- Necesitas condicionales como declaraciones `if` para tomar decisiones.
- Necesitas bucles para repetir las tareas hasta que una condición deja de ser verdad.
- Necesita funciones para organizar su código en trozos lógicos y reutilizables.

Los comentarios de código son una manera eficaz de escribir un código más legible, lo que hace que su programa sea más fácil de entender, mantener y corregir más tarde si hay problemas.

Por último, no descuiden el poder de la práctica. La mejor manera de aprender a escribir código es escribir código.

¡Estoy emocionado de que estés bien en tu camino de aprender a codificar, ahora! Seguid así. No olvide consultar otros recursos de programación para principiantes (libros, blogs, formación en línea, etc.). Este capítulo y este libro son un gran comienzo, pero son sólo una breve introducción.

El siguiente capítulo revisará muchos de los conceptos de este capítulo, pero desde una perspectiva más específica de JavaScript, que resaltará la mayoría de los principales temas que se abordan con más detalle en el resto de la serie.

2- En Javascript

En el capítulo anterior, introduje los componentes básicos de la programación, como variables, bucles, condicionales y funciones. Por supuesto, todo el código mostrado ha sido en JavaScript. Pero en este capítulo, queremos centrarnos específicamente en las cosas que debes saber sobre JavaScript para ponerte en marcha como desarrollador de JS.

Introduciremos bastantes conceptos en este capítulo que no serán explorados completamente hasta los libros posteriores de YDKJS. Puede pensar en este capítulo como una visión general de los temas tratados en detalle en el resto de esta serie.

Especialmente si eres nuevo en JavaScript, deberías pasar un buen rato revisando los conceptos y ejemplos de código aquí varias veces. Cualquier buena base se pone ladrillo por ladrillo, así que no espere que usted lo entienda inmediatamente todo con el primer paso.

Su viaje para aprender profundamente JavaScript comienza aquí.

Nota: Como dije en el capítulo 1, usted debe probar definitivamente todo este código usted mismo mientras lee y trabaja a través de este capítulo. Tenga en cuenta que parte del código aquí asume las capacidades introducidas en la versión más reciente de JavaScript en el momento de escribir este documento (comúnmente conocido como "ES6" para la 6 ª edición de ECMAScript - el nombre oficial de la especificación JS). Si está utilizando un navegador anterior a ES6, es posible que el código no funcione. Debe utilizarse una actualización reciente de un navegador moderno (como Chrome, Firefox o IE).

2.1 Valores y Tipos

Como hemos afirmado en el Capítulo 1, JavaScript ha escrito valores, no las variables escritas. Están disponibles los siguientes tipos (de datos) incorporados:

- `string`
- `number`
- `boolean`
- `null` y `undefined`
- `object`
- `symbol` (nuevo en ES6)

JavaScript proporciona un tipo de operador que puede examinar un valor y decirle qué tipo es:

```
var a;
typeof a;               // "undefined"

a = "hello world";
typeof a;               // "string"

a = 42;
typeof a;               // "number"

a = true;
typeof a;               // "boolean"

a = null;
typeof a;               // "object" -- weird, bug

a = undefined;
typeof a;               // "undefined"

a = { b: "c" };
typeof a;               // "object"
```

El valor de retorno del operador `typeof` es siempre uno de los seis valores (siete de ES6! - el "tipo `symbol`") en `string`. Es decir, `typeof "abc"` devuelve `"string"`, no `string`.

Observe cómo en este fragmento la variable `a` contiene cada tipo diferente de valor, y que a pesar de las apariencias, `typeof a` no está pidiendo el "tipo de `a`", sino más bien para el "tipo del valor actualmente en `a`". Sólo los valores tienen tipos en JavaScript; Las variables son solo contenedores simples para esos valores.

`typeof null` es un caso interesante, porque erróneamente devuelve `"object"`, cuando se espera que devuelva `"null"`.

Advertencia: Este es un error de larga data en JS, pero uno que es probable que nunca va a ser arreglado. Demasiado código en la Web tiene este error y por lo tanto, arreglarlo causaría mucho más errores!

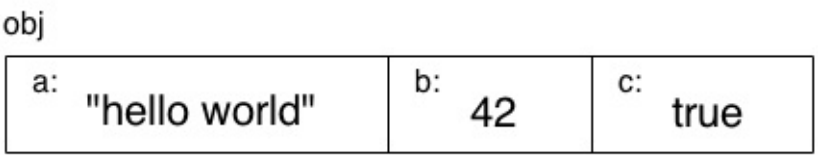
También note `a = undefined`. Definimos explícitamente un valor `undefined`, pero eso no es diferente de una variable que no tiene ningún valor definido, como con la variable `a`; como la línea de la parte superior del fragmento. Una variable puede llegar a este estado de valor "indefinido" de varias maneras diferentes, incluyendo funciones que no devuelven valores y uso del operador `void`.

Objetos

El tipo `object` se refiere a un valor compuesto en el que se pueden establecer propiedades (ubicaciones con nombre) que cada una tiene sus propios valores de cualquier tipo. Este es quizás uno de los tipos de valor más útiles en todo JavaScript.

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;        // "hello world"
obj.b;        // 42
obj.c;        // true

obj["a"];    // "hello world"
obj["b"];    // 42
obj["c"];    // true
```

Puede ser útil pensar visualmente en este valor `obj`:



Se puede acceder a las propiedades con notación de puntos (es decir, `obj.a`) o con notación de paréntesis (es decir, `obj["a"]`). La notación de puntos es más corta y por lo general más fácil de leer, y por lo tanto se prefiere cuando es posible.

La notación de paréntesis es útil si tienes un nombre de propiedad que tiene caracteres especiales en él, como `obj["¡hola mundo!"]` - tales propiedades se refieren a menudo como llaves cuando se accede a través de la notación de paréntesis. La notación `[]` requiere una variable (explicada a continuación) o un `string` literal (que debe ser envuelto en `".."` o `'..'`).

Por supuesto, la notación de paréntesis también es útil si desea acceder a una propiedad/clave pero el nombre se almacena en otra variable, como por ejemplo:

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];            // "hello world"
obj["b"];        // 42
```

Nota: Para obtener más información sobre objetos JavaScript, consulte el título This & Prototype Objects de esta serie, específicamente el Capítulo 3.

Hay un par de otros tipos de valores con los que comúnmente interactúas en los programas JavaScript: `array` y `function` . Pero en lugar de ser tipos incorporados apropiados, estos deberían ser pensados más como subtipos - versiones especializadas del tipo `object` .

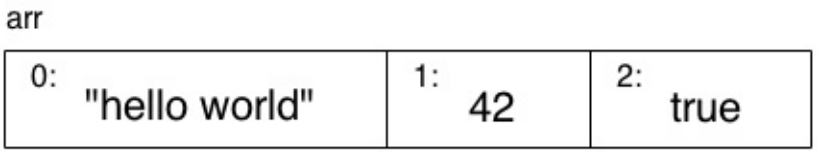
Arrays

Un `array` es un objeto que contiene valores (de cualquier tipo) particularmente no en las propiedades/claves con nombre, sino en posiciones numéricamente indexadas. Por ejemplo:

```
var arr = [
    "hello world",
    42,
    true
];

arr[0];            // "hello world"
arr[1];            // 42
arr[2];            // true
arr.length;        // 3

typeof arr;        // "object"
```

Nota: Los lenguajes que empiezan a contar a cero, como JS, utilizan `0` como el índice del primer elemento de la matriz.

Puede ser útil pensar en un `arr` visual:



Debido a que los arrays son objetos especiales (como `typeof` implica), también pueden tener propiedades, incluyendo la propiedad de longitud actualizada automáticamente.

Teóricamente podría utilizar un array como un objeto normal con sus propias propiedades con nombre, o podría usar un objeto, pero sólo le daría propiedades numéricas (`0` , `1` , etc.) similares a un array. Sin embargo, esto generalmente se considera un uso inadecuado de los tipos respectivos.

El mejor enfoque y más natural es usar arrays para valores numéricamente posicionados y usar objetos para propiedades con nombre.

Funciones

El otro subtipo de objeto que usará en todos sus programas JS es una función:

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;            // "function"
typeof foo();        // "number"
typeof foo.bar;        // "string"
```

De nuevo, las funciones son un subtipo de objetos - `typeof` devuelve `"function"` , lo que implica que una función es un tipo principal - y por lo tanto puede tener propiedades, pero normalmente sólo se utilizan las propiedades del objeto `function` (como `foo.bar`) en limitadas casos.

Nota: Para obtener más información sobre los valores de JS y sus tipos, consulte los dos primeros capítulos del título de Tipos y Gramática de esta serie.

Métodos de tipo incorporado

Los tipos y subtipos incorporados que acabamos de comentar tienen comportamientos predefinidos como propiedades y métodos que son bastante potentes y útiles.

Por ejemplo:

```
var a = "hello world";
var b = 3.14159;

a.length;                 // 11
a.toUpperCase();         // "HELLO WORLD"
b.toFixed(4);             // "3.1416"
```

El "cómo lo hace" detrás de poder llamar `a.toUpperCase()` es más complicado que apenas ese método existente en el valor.

En pocas palabras, hay una forma de envoltorio del objeto `String` (capitalizar `S`), normalmente llamada "nativa", que se empareja con el tipo de cadena primitiva; Es este contenedor de objetos que define el método `toUpperCase()` en su prototipo.

Cuando se utiliza un valor primitivo como `"hello world"` como un objeto haciendo referencia a una propiedad o método (por ejemplo, `a.toUpperCase()` en el fragmento anterior), JS automáticamente "encaja" el valor a su contraparte wrapper (escondido bajo cubierta).

Un valor de `string` puede ser envuelto por un objeto `String` , un `number` puede ser envuelto por un objeto `Number` y un `boolean` puede ser envuelto por un objeto `Boolean` . En la mayoría de las veces, no es necesario preocuparse ni utilizar directamente estas formas de envoltorio de objetos en los valores - se prefieren las formas de valores primitivos en prácticamente todos los casos y JavaScript se encargará del resto por usted.

Nota: Para obtener más información sobre JS nativos y "encajonamiento", consulte el Capítulo 3 del título de Tipos y Gramática de esta serie. Para comprender mejor el prototipo de un objeto, consulte el Capítulo 5 del título this & Object Prototypes de esta serie.

Comparación de valores

Hay dos tipos principales de comparación de valores que necesitará hacer en sus programas JS: igualdad y desigualdad. El resultado de cualquier comparación es un valor estrictamente booleano (verdadero o falso), independientemente de qué tipos de valores se comparan.

Coerción

Hablamos brevemente de la coerción en el Capítulo 1, pero vamos a revisarla aquí.

La coerción viene en dos formas en JavaScript: explícita e implícita. La coerción explícita es simplemente que usted puede ver obviamente en el código que una conversión de un tipo a otro ocurrirá, mientras que la coerción implícita es cuando la conversión del tipo puede suceder más como un efecto secundario no obvio de alguna otra operación.

Usted probablemente ha escuchado sentimientos como "la coerción es maligna" sacado del hecho de que hay claramente lugares donde la coerción puede producir algunos resultados sorprendentes. Tal vez nada evoca más la frustración de los desarrolladores que cuando el lenguaje les sorprende.

La coerción no es mala, ni tiene que ser sorprendente. De hecho, la mayoría de los casos puede construir con la coerción de tipo de forma bastante sensata y comprensible, e incluso se puede utilizar para mejorar la legibilidad de su código. Pero no vamos a ir mucho más lejos en ese debate - El Capítulo 4 del título Tipos y Gramática de esta serie cubre todos los lados.

He aquí un ejemplo de coerción explícita:

```
var a = "42";

var b = Number( a );

a;                // "42"
b;                // 42 -- the number!
```

Y aquí hay un ejemplo de coerción implícita:

```
var a = "42";

var b = a * 1;    // "42" implicitly coerced to 42 here

a;                // "42"
b;                // 42 -- the number!
```

Truthy y Falsy

En el capítulo 1, mencionamos brevemente la naturaleza "verdadera" y "falsa" de los valores: cuando un valor no booleano es coaccionado a un booleano, ¿se convierte en verdadero o falso, respectivamente?

La lista específica de valores falsos en JavaScript es la siguiente:

- `""` (string vacío)
- `0`, `-0`, `NaN` (numero inválido)
- `null`, `undefined`

- `false`

Cualquier valor que no esté en esta lista falsa es " `true` ". Estos son algunos ejemplos:

- `"hello"`
- `42`
- `true`
- `[ ]` , `[ 1, "2", 3 ]` (arrays)
- `{ }` , `{ a: 42 }` (objects)
- `function foo() { .. }` (functions)

Es importante recordar que un valor no booleano sólo sigue esta coerción " `true` " / " `false` " si es realmente coaccionado a un booleano. No es tan difícil confundirse con una situación que parece que coacciona un valor a un booleano cuando no lo es.

Igualdad

Hay cuatro operadores de igualdad: `==` , `===` , `!=` , Y `!==` . Las forma `!` son, por supuesto, las versiones simétricas "no iguales" de sus contrapartes; La no igualdad no debe confundirse con la desigualdad.

La diferencia entre `==` y `===` normalmente se caracteriza por que `==` comprueba la igualdad de valor y `===` comprueba tanto el valor como la igualdad de tipo. Sin embargo, esto es inexacto. La manera correcta de caracterizarlos es que `==` chequea por igualdad de valor con coerción permitida, y `===` comprueba la igualdad de valor sin permitir coerción; `===` se llama a menudo "igualdad estricta" por esta razón.

Considere la coerción implícita que es permitida por la comparación `==` loose-equality y no se permite con la `===` estricta-igualdad:

```
var a = "42";
var b = 42;

a == b;            // true
a === b;        // false
```

En la comparación `a == b` , JS advierte que los tipos no coinciden, por lo que pasa por una serie ordenada de pasos para coaccionar uno o ambos valores a un tipo diferente hasta que los tipos coincidan, donde entonces se puede comprobar una igualdad de valor simple .

Si piensas en ello, hay dos formas posibles de que `a == b` puedan dar `true` a través de coerción. O la comparación podría terminar como `42 == 42` o podría ser `"42" == "42"` . Entonces, ¿cuál es?

La respuesta: `"42"` se convierte en `42`, para hacer la comparación `42 == 42`. En un ejemplo tan simple, en realidad no parece importar de qué manera va ese proceso, ya que el resultado final es el mismo. Hay casos más complejos donde importa no sólo cuál es el resultado final de la comparación, sino cómo llegar allí.

El `a === b` produce `false`, porque la coerción no está permitida, por lo que la comparación de valores sencillos obviamente falla. Muchos desarrolladores sienten que `===` es más predecible, por lo que abogan por usar siempre esa forma y mantenerse alejados de `==`. Creo que esta visión es muy miope. Creo que `==` es una poderosa herramienta que ayuda a su programa, si usted se toma el tiempo para aprender cómo funciona.

No vamos a cubrir todos los detalles básicos de cómo la coerción en comparaciones funciona aquí. Mucho de él es bastante sensible, pero hay algunos casos importantes a tener cuidado. Puede leer la sección 11.9.3 de la especificación ES5 (http://www.ecma-international.org/ecma-262/5.1/) para ver las reglas exactas, y le sorprenderá lo sencillo que es este mecanismo , En comparación con todo el bombo negativo que lo rodea.

Para reducir un montón de detalles a unos pocos takeaways simples, y ayudarle a saber si utilizar `==` o `===` en varias situaciones, aquí están mis reglas simples:

- Si un valor (aka lado) en una comparación podría ser el valor `true` o `false`, evite `==` y utilice `===`.
- Si cualquier valor en una comparación podría ser uno de estos valores específicos (`0`, `""`, o `[]` - matriz vacía), evite `==` y utilice `===`.
- En todos los demás casos, es seguro utilizar `==`. No sólo es seguro, sino que en muchos casos simplifica su código de una manera que mejora la legibilidad.

A lo que estas reglas se reducen es a exigir que usted piense críticamente sobre su código y sobre qué tipos de valores pueden venir a través de las variables que se comparan por igualdad. Si puede estar seguro acerca de los valores, y `==` es seguro, úselo! Si no puede estar seguro acerca de los valores, utilice `===`. Es así de simple.

El `!=` No-igualdad forma pares con `==`, y el `!==` pares de forma con `===`. Todas las reglas y observaciones que acabamos de discutir se mantienen simétricamente para estas comparaciones sin igualdad.

Debe tomar nota especial de las reglas de comparación `==` y `===` si está comparando dos valores no primitivos, como objetos (incluyendo funciones y arrays). Debido a que esos valores se mantienen en realidad por referencia, las comparaciones `==` y `===` simplemente comprobarán si las referencias coinciden, no cualquier cosa acerca de los valores subyacentes.

Por ejemplo, las matrices se coaccionan por defecto a strings simplemente uniendo todos los valores con comas (`,`) . Podría pensar que dos matrices con el mismo contenido serían `==` iguales, pero no lo son:

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;        // true
b == c;        // true
a == b;        // false
```

Nota: Para obtener más información acerca de las reglas de comparación de igualdad, consulte la especificación ES5 (sección 11.9.3) y consulte también el Capítulo 4 del título de Tipos y Gramática de esta serie; Consulte el Capítulo 2 para obtener más información acerca de valores versus referencias.

Desigualdad

Los operadores `<` , `>` , `<=` , y `>=` se utilizan para la desigualdad, a la que se hace referencia en la especificación como "comparación relacional". Normalmente se utilizarán con valores comparables como números. Es fácil entender que `3 < 4` .

Pero los valores `string` de JavaScript también se pueden comparar por desigualdad, usando reglas alfabéticas típicas (`"bar" < "foo"`).

¿Y la coerción? Se usan reglas similares como con la comparación `==` (aunque no exactamente idénticas!) a los operadores de desigualdad. Cabe destacar que no hay operadores de "desigualdad estricta" que desautorizarían la coerción de la misma manera que la "igualdad estricta".

Considere:

```
var a = 41;
var b = "42";
var c = "43";

a < b;         // true
b < c;         // true
```

¿Qué pasa aquí? En la sección 11.8.5 de la especificación ES5, se dice que si ambos valores en la comparación `<` son strings, como ocurre con `b < c` , la comparación se hace lexicográficamente (aka alfabéticamente como un diccionario). Pero si uno o ambos no es

una cadena, como ocurre con `a < b`, entonces ambos valores son forzados a ser números y se produce una comparación numérica típica.

El gotcha más grande que se puede encontrar aquí con comparaciones entre tipos de valor potencialmente diferentes - recuerde, no hay formas de "desigualdad estricta" para usar - es cuando uno de los valores no se puede convertir en un número válido, como:

```
var a = 42;
var b = "foo";

a < b;        // false
a > b;        // false
a == b;        // false
```

Espera, ¿cómo pueden ser falsas las tres comparaciones? Debido a que el valor `b` está siendo coaccionado al "valor de número inválido" `NaN` en las comparaciones `<` y `>`, y la especificación dice que `NaN` no es ni mayor ni menor que cualquier otro valor.

La comparación `==` falla por una razón diferente. `a == b` podría fallar si se interpreta como `42 == NaN` o `"42" == "foo"` - como hemos explicado anteriormente, el primero es el caso.

Nota: Para obtener más información sobre las reglas de comparación de desigualdades, consulte la sección 11.8.5 de la especificación ES5 y consulte también el Capítulo 4 del título Tipos y Gramática de esta serie.

2.2 Variables

En JavaScript, los nombres de las variables (incluidos los nombres de las funciones) deben ser identificadores válidos. Las reglas estrictas y completas para caracteres válidos en identificadores son un poco complejas cuando se consideran caracteres no tradicionales como Unicode. Sin embargo, si sólo se consideran caracteres alfanuméricos ASCII típicos, las reglas son simples.

Un identificador debe comenzar con `a-z`, `A-Z`, `$` o `_`. Entonces puede contener cualquiera de esos caracteres más los números `0-9`.

Generalmente, las mismas reglas se aplican a un nombre de propiedad como a un identificador de variable. Sin embargo, ciertas palabras no se pueden utilizar como variables, pero son correctas como nombres de propiedad. Estas palabras se llaman "palabras reservadas" e incluyen las palabras clave JS (`for`, `in`, `if`, etc.) así como `null`, `true` y `false`.

Nota: Para obtener más información sobre las palabras reservadas, consulte el Apéndice A del título Tipos y Gramática de esta serie.

Ámbitos (scope) de las funciones

Utilice la palabra clave `var` para declarar una variable que pertenecerá al ámbito de la función actual o al ámbito global si está en el nivel superior fuera de cualquier función.

Hoisting (Elevación)

Siempre que aparezca una `var` dentro de un ámbito, esa declaración se toma para pertenecer a todo el ámbito y es accesible en todas partes.

Metafóricamente, este comportamiento se denomina ***hoisting***, cuando una declaración `var` es conceptualmente "movida" a la parte superior de su ámbito de inclusión. Técnicamente, este proceso se explica con más precisión por la forma en que se compila el código, pero podemos omitir esos detalles por ahora.

Considerar:

```
var a = 2;

foo();    // funciona porque la declaracion `foo()` esta "hoisted" (elevada). O sea qu
e primero ha sido llamada
       // la función y después es declarada

function foo() {
    a = 3;

    console.log( a );    // 3

    var a;        // la declaración esta "hoisted" (elevada) en la parte superior de `
foo()`
}

console.log( a );    // 2
```

Advertencia: No es común o una buena idea confiar en la elevación (hoisting) de la variable para usar esa variable en su alcance que aparece en su declaración var; Puede ser bastante confuso. Es mucho más común y aceptado el uso de declaraciones de funciones *hoisting*, como lo hacemos con la llamada `foo()` que aparece antes de su declaración formal.

Ámbitos anidados

Cuando declara una variable, ella estará disponible en cualquier parte de ese ámbito, así como en cualquier ámbito inferior/interno. Por ejemplo:

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

            function baz() {
                var c = 3;

                console.log( a, b, c );    // 1 2 3
            }

            baz();
            console.log( a, b );        // 1 2
    }

    bar();
    console.log( a );               // 1
}

foo();
```

Observe que `c` no está disponible dentro de `bar()`, porque se declara sólo dentro del ámbito interno `baz()` y que `b` no está disponible para `foo()` por la misma razón.

Si intenta acceder al valor de una variable en un ámbito en el que no está disponible, obtendrá un `ReferenceError`. Si intenta establecer una variable que no ha sido declarada, o bien terminará creando una variable en el ámbito global de nivel superior (¡mal!) O recibiendo un error, dependiendo del "modo estricto" (consulte "Modo estricto "). Vamos a ver:

```
function foo() {
    a = 1;    // `a` no esta formalmente declarada con `var`
}

foo();
a;            // 1 -- oops, variable automatica global :(
```

Esta es una muy mala práctica. ¡No lo hagas! Siempre declare formalmente sus variables.

Además de crear declaraciones para variables en el nivel de función, ES6 le permite declarar que las variables pertenecen a bloques individuales (pares de `{..}`), usando la palabra clave `let`. Además de algunos detalles matizados, las reglas de alcance se comportarán aproximadamente iguales a las que acabamos de ver con funciones:

```
function foo() {
    var a = 1;

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

Debido al uso de `let` en lugar de `var`, `b` pertenecerá sólo a la instrucción `if` y, por tanto, no al ámbito de la función `foo()`. Del mismo modo, `c` sólo pertenece al bucle `while`. El ámbito de bloque es muy útil para administrar sus ámbitos variables de una manera más refinada, lo que puede hacer que su código sea mucho más fácil de mantener con el tiempo.

Nota: Para obtener más información sobre los ámbitos, consulte el título Scope & Closures de esta serie. Consulte el título ES6 & Beyond de esta serie para obtener más información sobre el ámbito de bloque `let`.

2.3 Condicionales

Además de la declaración `if` que introdujimos brevemente en el Capítulo 1, JavaScript proporciona algunos otros mecanismos condicionales a los que deberíamos echar un vistazo.

A veces se puede encontrar escribiendo una serie de declaraciones `if..else..if` como esta:

```
if (a == 2) {
    // do something
}
else if (a == 10) {
    // do another thing
}
else if (a == 42) {
    // do yet another thing
}
else {
    // fallback to here
}
```

Esta estructura funciona, pero es un poco verbosa porque es necesario especificar la comparación `a` para cada caso. Aquí hay otra opción, la instrucción `switch` :

```
switch (a) {
    case 2:
        // do something
        break;
    case 10:
        // do another thing
        break;
    case 42:
        // do yet another thing
        break;
    default:
        // fallback to here
}
```

El `break` es importante si sólo desea que se ejecute la instrucción(es) en un `case` (caso). Si omite el `break` de un `case` , y ese `case` coincide o se ejecuta, la ejecución continuará con las declaraciones del `case` siguiente, independientemente de la coincidencia de ese `case` . A veces, este llamado "caer a través (fall through)" es útil/deseado:

```
switch (a) {
    case 2:
    case 10:
        // some cool stuff
        break;
    case 42:
        // other stuff
        break;
    default:
        // fallback
}
```

Aquí, si `a` es `2` o `10`, ejecutará el código "// some cool stuff"

Otra forma de condicional en JavaScript es el "operador condicional", a menudo llamado el "operador ternario". Es como una forma más concisa de una sola declaración `if..else`, como:

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// similar to:

// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

Si la expresión de prueba (`a > 41` aquí) se evalúa como verdadera, se obtiene la primera cláusula (`"hola"`), de lo contrario se obtiene la segunda cláusula (`"mundo"`).

El operador condicional no tiene que ser utilizado en una asignación, pero eso es definitivamente el uso más común.

Nota: Para obtener más información sobre las condiciones de prueba y otros patrones de `switch` y `? :`, Vea el título Tipos y Gramática de esta serie.

2.4 Modo estricto

ES5 agregó un "modo estricto" al lenguaje, que aprieta las reglas para ciertos comportamientos. Generalmente, estas restricciones se consideran como mantener el código a un conjunto de directrices más seguro y más apropiado. Además, la adhesión al modo estricto hace que su código en general sea más optimizable por el motor. El modo estricto es un gran triunfo para el código, y deberías usarlo para todos tus programas.

Puede optar por el modo estricto para una función individual, o un archivo entero, dependiendo de donde se puso el modo estricto:

```
function foo() {
    "use strict";

    // este código esta en modo estricto

    function bar() {
        // este código esta en modo estricto
    }
}

// este código no esta en modo estricto
```

Compare esto con:

```
"use strict";

function foo() {
    // este código esta en modo estricto

    function bar() {
        // este código esta en modo estricto
    }
}

// este código esta en modo estricto
```

Una diferencia clave (¡mejora!) con el modo estricto es rechazar la declaración implícita de variables globales automáticas al omitir la variable:

```
function foo() {
    "use strict";    // activa el modo etsricto
    a = 1;        // `var` olvidada, es igual a ReferenceError
}

foo();
```

Si activa el modo estricto en su código, y obtiene errores, o el código comienza a comportarse con errores, su tentación podría ser evitar el modo estricto. Pero ese instinto sería una mala idea para complacerse. Si el modo estricto causa problemas en su programa, casi seguro que es una señal de que tiene cosas en su programa que debe arreglar.

No sólo el modo estricto mantendrá su código en un camino más seguro, y no sólo hará que su código sea más optimizable, sino que también representa la dirección futura del lenguaje. Sería más fácil para ti acostumbrarte al modo estricto ahora que seguir poniéndolo en el código, ¡sólo será más difícil cambiarlo más tarde!

Nota: Para obtener más información sobre el modo estricto, consulte el Capítulo 5 del título Tipos y Gramática de esta serie.

2.5 Funciones como Valores

Hasta ahora, hemos discutido las funciones como el principal mecanismo de scope en JavaScript. Puede recordar la sintaxis típica de la declaración de funciones de la siguiente manera:

```
function foo() {
    // ..
}
```

Aunque no parezca obvio de esa sintaxis, `foo` es básicamente sólo una variable en el ámbito exterior que incluye una referencia a la función que se está declarando. Es decir, la función en sí es un valor, al igual que sería `42` o `[1,2,3]` .

Esto puede sonar como un concepto extraño al principio, así que tómese un momento para reflexionar sobre él. No sólo puede pasar un valor (argumento) a una función, sino que una función en sí puede ser un valor que se asigna a variables, o se pasa a, o se devuelve a otras funciones.

Como tal, un valor de función debe ser pensado como una expresión, al igual que cualquier otro valor o expresión.

Considerar:

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

La primera expresión de función asignada a la variable `foo` se llama anónima porque no tiene nombre.

La segunda expresión de función llamada (`bar`), incluso como una referencia a ella también se asigna a la variable `x` . Las expresiones de función nombradas son generalmente más preferibles, aunque las expresiones de función anónimas todavía son extremadamente comunes.

Para obtener más información, consulte el título Scope & Closures de esta serie.

Immediately Invoked Function Expressions (IIFEs) - Expresiones de función invocadas Inmediatamente

En el snippet anterior, ninguna de las expresiones de función se ejecutan - podríamos si hubiéramos incluido `foo()` o `x()` , por ejemplo.

Hay otra forma de ejecutar una expresión de función, que normalmente se denomina expresión de función inmediatamente invocada (IIFE):

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

El exterior `(..)` que rodea la expresión de función `(función IIFE () {..})` es sólo un matiz de la gramática JS necesaria para evitar que se trate como una declaración de función normal.

El final `()` al final de la expresión - el `}) ()` ; - es lo que en realidad ejecuta la expresión de función referenciada inmediatamente antes de ella.

Eso puede parecer extraño, pero no es tan extraño como a primera vista. Considere las similitudes entre `foo` y `IIFE` aquí:

```
function foo() { .. }

// `foo` function reference expression,
// then `()` executes it
foo();

// expresión de función `IIFE`,
// entonces `()` lo ejecuta
(function IIFE(){ .. })();
```

Como se puede ver, la lista de la función `(IIFE () {..})` antes de su ejecución `()` es esencialmente el mismo que incluye `foo` antes de su ejecución `()` ; En ambos casos, la referencia de función se ejecuta con `()` inmediatamente después de ella.

Debido a que un `IIFE` es sólo una función y las funciones crean un ámbito variable, el uso de un `IIFE` de esta manera se utiliza a menudo para declarar variables que no afectarán al código circundante fuera del `IIFE` :

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );    // 10
})();

console.log( a );        // 42
```

Los `IIFE` también pueden tener valores de retorno:

```
var x = (function IIFE(){
    return 42;
})();

x;    // 42
```

El valor `42` se devuelve de la función denominada `IIFE` que se ejecuta, y luego se asigna a `x`.

Closure

El closure es uno de los conceptos más importantes, ya menudo menos comprendidos, en JavaScript. No lo voy a cubrir en detalle aquí y en su lugar le remito al título Scope & Closures de esta serie. Pero quiero decir algunas cosas sobre el para que entiendas el concepto general. Será una de las técnicas más importantes en su nivel de habilidades JS.

Puede pensar en el closure como una forma de "***recordar***" y seguir accediendo al ámbito de una función (y sus variables) incluso una vez que la función ha terminado de ejecutarse.

Considerar:

```
function makeAdder(x) {
    // El parámetro `x` es una variable interna

    // función interna `add ()` usa `x`, así que
    // tiene un "closure" sobre él
    function add(y) {
        return y + x;
    };

    return add;
}
```

La referencia a la función interna `add(..)` que se retorna con cada llamada al `makeAdder(..)` externo es capaz de recordar cualquier valor `x` que se pasó en `makeAdder(..)`. Ahora, vamos a usar `makeAdder(..)`:

```
// `plusOne` obtiene una referencia a la función `add(..)` interna
//  con closure sobre el parámetro `x` de
// el exterior de `makeAdder(..)`
var plusOne = makeAdder( 1 );

// `plusTen` obtiene una referencia a la función `add (..)`interna
// con closure sobre el parámetro `x` de
// el exterior `makeAdder (..)`
var plusTen = makeAdder( 10 );

plusOne( 3 );         // 4  <-- 1 + 3
plusOne( 41 );         // 42 <-- 1 + 41

plusTen( 13 );         // 23 <-- 10 + 13
```

Más información sobre cómo funciona este código:

1. Cuando llamamos `makeAdder(1)`, obtenemos una referencia a su `add(..)` interna que recuerda `x` como `1`. Llamamos a esta referencia de función `plusOne(..)`.
2. Cuando llamamos `makeAdder(10)`, obtenemos otra referencia a su `add(..)` interna que recuerda `x` como `10`. Llamamos a esta referencia de función `plusTen(..)`.
3. Cuando llamamos `plusOne(3)`, añade `3` (su `y` interno) al `1` (recordado por `x`), y obtenemos `4` como el resultado.
4. Cuando llamamos `plusTen(13)`, añade `13` (su `y` interno) al `10` (recordado por `x`), y obtenemos `23` como el resultado.

No se preocupe si esto parece extraño y confuso al principio - puede serlo! Tomará mucha práctica entenderlo completamente.

Pero confía en mí, una vez que lo hagas, es una de las técnicas más poderosas y útiles en toda la programación. Definitivamente vale la pena el esfuerzo para dejar que su cerebro cocine a fuego lento los closures poco a poco. En la siguiente sección, tendremos un poco más de práctica con los closures.

Modules

El uso más común de los closures en JavaScript es el patrón module. Los modules le permiten definir detalles de implementación privados (variables, funciones) ocultos del mundo exterior, así como una API pública accesible desde el exterior.

Considere:

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // do the rest of the login work
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// create a `User` module instance
var fred = User();

fred.login( "fred", "12Battery34!" );
```

La función `User()` sirve como un ámbito externo que contiene las variables `username` y `password`, así como la función interna `doLogin()`; Estos son todos los detalles internos privados de este módulo de usuario que no se puede acceder desde el mundo exterior.

Advertencia: No estamos llamando a `new User()` aquí, a propósito, a pesar del hecho de que probablemente parece más común a la mayoría de los lectores. `User()` es sólo una función, no una clase a instanciar, por lo que se llama normalmente. El uso de `new` sería inapropiado y realmente desperdiciaría recursos.

Ejecutar `User()` crea una instancia del módulo `User` - se crea un nuevo ámbito y, por lo tanto, una copia completamente nueva de cada una de estas variables/funciones internas. Asignamos esta instancia a `fred`. Si ejecutamos `User()` de nuevo, obtendremos una nueva instancia completamente distinta de `fred`.

La función interna `doLogin()` tiene un closure sobre el `username` y `password`, lo que significa que conservará su acceso a ellos incluso después de que la función `User()` termine de ejecutarse.

`PublicAPI` es un objeto con una propiedad/método en él, `login`, que es una referencia a la función interna `doLogin()`. Cuando devolvemos `publicAPI` de `User()`, se convierte en la instancia que llamamos `fred`.

En este punto, la función externa `User()` ha terminado de ejecutarse. Normalmente, se piensa que las variables internas como `username` y `password` se han ido. Pero aquí no lo han hecho, porque hay un closure en la función `login()` que los mantiene vivos.

Es por eso que podemos llamar a `fred.login(..)` - lo mismo que llamar a la interna `doLogin(..)` - y todavía puede acceder a las variables internas de `username` y `password`.

Hay una buena probabilidad de que con sólo este breve vistazo al closures y el patrón module, algunos de ellos sean todavía un poco confusos. ¡Está bien! Toma un poco de trabajo para envolver su cerebro alrededor de él.

A partir de aquí, vaya a leer el título Scope & Closures de esta serie para una exploración mucho más profunda.

2.6 Identificador `This`

Otro concepto muy comúnmente mal entendido en JavaScript es el identificador `this` . Una vez más, hay un par de capítulos sobre él de esta serie, así que aquí vamos a introducir brevemente el concepto.

Aunque a menudo parece que `this` está relacionado con "patrones orientados a objetos", en JS este es un mecanismo diferente.

Si una función tiene una referencia `this` dentro de ella, esa referencia `this` normalmente apunta a un `object` . Pero el `object` al que apunta depende de cómo se llamó la función.

Es importante darse cuenta de que `this` no se refiere a la función en sí, y ése es el error más común.

He aquí una ilustración rápida:

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();               // "global"
obj1.foo();           // "obj1"
foo.call( obj2 );       // "obj2"
new foo();           // undefined
```

Hay cuatro reglas para cómo se establece `this` , y se muestran en las cuatro últimas líneas de ese fragmento.

1. `foo()` termina configurando `this` para el objeto global en modo no estricto - en modo estricto, esto sería `undefined` y obtendría un error en el acceso a la propiedad `bar` - por lo que " `global` " es el valor encontrado para este `.bar` .

2. `obj1.foo()` establece `this` en el object `obj1` .
3. `foo.call(obj2)` establece `this` en el objeto `obj2` .
4. `new foo()` establece `this` en un nuevo objeto vacío.

En pocas palabras: para entender a qué se refiere `this` , hay que examinar cómo se llamó la función en cuestión. Será una de esas cuatro maneras que acabamos de mostrar, y eso responderá a lo que es `this` .

Nota: Para obtener más información al respecto, consulte los capítulos 1 y 2 del título This y Objetos Prototipos de esta serie.

2.7 Prototypes

El mecanismo prototype en JavaScript es bastante complicado. Sólo le echaremos un vistazo aquí. Usted querrá pasar un montón de tiempo revisando los capítulos 4-6 del título de this y los objetos prototipos de esta serie para todos los detalles.

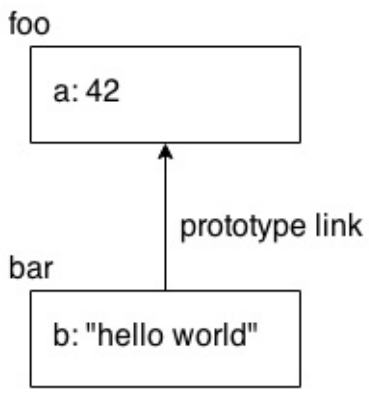
Cuando hace referencia a una propiedad en un objeto, si esa propiedad no existe, JavaScript utilizará automáticamente la referencia de prototipo interno de ese objeto para buscar otro objeto en el que buscar la propiedad. Se podría pensar en esto casi como un fallback si la propiedad está desaparecida.

El vínculo de referencia del prototipo interno de un objeto a su fallback ocurre en el momento en que se crea el objeto. La forma más sencilla de ilustrarlo es con una utilidad incorporada llamada `Object.create(..)`.

Considere:

```
var foo = {
    a: 42
};

// crear `bar` y vincularlo a` foo`
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;        // "hello world"
bar.a;        // 42 <-- delegado a `foo`
```

Puede ayudar a visualizar los objetos `foo` y `bar` y su relación:



La propiedad `a` no existe en realidad en el objeto `bar`, sino porque `bar` está vinculada al prototipo de `foo`, JavaScript automáticamente vuelve a buscar en el objeto `foo`, donde se encuentra.

Este vínculo puede parecer una característica extraña del lenguaje. La forma más común de usar esta característica -y yo diría, abusada- es tratar de emular/falsificar un mecanismo de "clase" con "herencia".

Pero una forma más natural de aplicar prototipos es un patrón llamado "delegación de comportamiento", en el que intencionalmente diseña sus objetos vinculados para poder delegar de uno a otro las partes del comportamiento necesario.

Nota: Para obtener más información sobre los prototipos y la delegación de comportamiento, consulte los capítulos 4-6 del título This y Objetos Prototipos de esta serie.

2.8 Lo Viejo y Lo Nuevo

Algunas de las características de JS que ya hemos cubierto, y ciertamente muchas de las características cubiertas en el resto de esta serie, son nuevas incorporaciones y no necesariamente estarán disponibles en navegadores antiguos. De hecho, algunas de las características más recientes de la especificación aún no están implementadas en ningún navegador estable.

Entonces, ¿qué hacemos con las cosas nuevas? ¿Sólo tenemos que esperar alrededor de unos años o décadas para que todos los viejos navegadores se desvanezcan en la oscuridad?

Esta es la forma en que la gente que piensa acerca de esta situación, pero en realidad no es un enfoque saludable para JS.

Hay dos técnicas principales que puede utilizar para "llevar" las cosas nuevas de JavaScript a los navegadores más antiguos: polyfilling y transpiling.

Polyfilling

La palabra "polyfill" es un término inventado (por Remy Sharp) (https://remysharp.com/2010/10/08/what-is-a-polyfill) utilizado para referirse a tomar la definición de una característica más reciente y producir un pedazo de código que es equivalente al comportamiento, pero es capaz de ejecutarse en entornos JS más antiguos.

Por ejemplo, ES6 define una utilidad denominada `Number.isNaN(..)` para proporcionar una verificación exacta y sin errores en los valores `NaN`, despreciando la utilidad original `isNaN(..)`. Pero es fácil "polifilear" la utilidad para que pueda empezar a utilizarlo en su código, independientemente de si el usuario final está en un navegador ES6 o no.

Considere:

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

La instrucción `if` lo protege contra la aplicación de la definición de polyfill en los exploradores ES6 donde ya existirá. Si no está presente, definimos `Number.isNaN(..)`.

`Nota` : El chequeo que hacemos aquí se aprovecha de una peculiaridad con valores `NaN` , que es que son el único valor en todo el lenguaje que no es igual a sí mismo. Así que el valor `NaN` es el único que haría que `x !== x` sea `true` .

No todas las nuevas características son completamente polifilables. A veces la mayor parte del comportamiento puede ser polifileado, pero todavía hay pequeñas desviaciones. Usted debe ser muy, muy cuidadoso en la aplicación de un polyfill que haga usted mismo, para asegurarse de que se adhieren a la especificación tanto como sea posible.

O mejor aún, utilice un conjunto de polyfills ya probado en los que puede confiar, como los proporcionados por ES5-Shim (https://github.com/es-shims/es5-shim) y ES6-Shim (https: // Github.com/es-shims/es6-shim).

Transpiling

No hay manera de polifilear nueva sintaxis que se ha agregado al lenguaje. La nueva sintaxis arrojaría un error en el antiguo motor JS como unrecognized/invalid.

Por lo tanto, la mejor opción es utilizar una herramienta que convierte su código más reciente en equivalentes de código antiguo. Este proceso es comúnmente llamado "transpiling", un término para transformar + compilar.

Esencialmente, su código fuente es creado en el nuevo formulario de sintaxis, pero lo que implementa en el navegador es el código transpilado en forma de sintaxis antigua. Por lo general, insertar el transpiler en su proceso de construcción, similar a su linter code o su minifier.

Usted podría preguntarse por qué me tomaría la molestia de escribir nueva sintaxis sólo para que se traslade lejos de código antiguo - ¿por qué no sólo escribir el código más antiguo directamente?

Hay varias razones importantes por las que debe preocuparse por el transpiling:

- La nueva sintaxis agregada al lenguaje está diseñada para hacer que su código sea más legible y mantenible. Los equivalentes más viejos son a menudo mucho más enrevesados. Debería preferir escribir una sintaxis nueva y más limpia, no sólo para usted sino para todos los demás miembros del equipo de desarrollo.
- Si transpila sólo para los navegadores más antiguos, pero sirve la nueva sintaxis a los navegadores más recientes, podrá aprovechar las optimizaciones del rendimiento del navegador con la nueva sintaxis. Esto también permite a los fabricantes del navegador tener más código del mundo real para probar sus implementaciones y optimizaciones.
- El uso de la nueva sintaxis anticipadamente permite que se pruebe de forma más robusta en el mundo real, lo que proporciona retroalimentación al comité JavaScript (TC39). Si los problemas se encuentran a tiempo, pueden ser cambiados/arreglados

> antes de que los errores de diseño de lenguaje se conviertan en permanentes.

Aquí está un ejemplo rápido de transpiling. ES6 agrega una característica llamada "valores de parámetros predeterminados". Se parece a esto:

```
function foo(a = 2) {
    console.log( a );
}

foo();        // 2
foo( 42 );    // 42
```

Simple, ¿verdad? ¡Útil, también! Pero es una nueva sintaxis que no es válida en motores pre-ES6. Entonces, ¿qué hará un transpiler con ese código para que funcione en entornos más antiguos?

```
function foo() {
    var a = arguments[0] !== (void 0) ? arguments[0] : 2;
    console.log( a );
}
```

Como puede ver, comprueba si el valor de los `arguments[0]` es `void 0` (aka `undefined`), y si es así, proporciona el valor predeterminado `2`; De lo contrario, asigna lo que se pasó.

Además de poder utilizar ahora la sintaxis más agradable incluso en los navegadores más antiguos, ver el código transpilado explica realmente el comportamiento previsto más claramente.

Es posible que no se haya dado cuenta sólo de mirar la versión de ES6 que `undefined` es el único valor que no se puede pasar explícitamente en un parámetro de valor predeterminado, pero el código transpilado lo hace mucho más claro.

El último detalle importante a destacar sobre los transpiladores es que ahora deben ser considerados como una parte estándar del ecosistema y proceso de desarrollo de JS. JS va a seguir evolucionando, mucho más rápidamente que antes, por lo que cada pocos meses se añadirá nueva sintaxis y nuevas características.

Si utiliza un transpilador de forma predeterminada, siempre podrá hacer que cambie a la sintaxis más reciente siempre que lo encuentre útil, en lugar de esperar años para que los navegadores de hoy se eliminen progresivamente.

Hay bastantes transpilers grandes para que usted elija de. Aquí hay algunas buenas opciones en el momento de escribir esto:

- Babel (https://babeljs.io) (formerly 6to5): Transpila ES6+ dentro de ES5

- Traceur (https://github.com/google/traceur-compiler): Transpila ES6, ES7, y más alla en ES5

2.9 Non-JavaScript

Hasta ahora, las únicas cosas que hemos cubierto están en el propio lenguaje JS. La realidad es que la mayoría de JS está escrito para funcionar e interactuar con entornos como los navegadores. Un buen pedazo de las cosas que usted escribe en su código es, estrictamente hablando, no controlado directamente por JavaScript. Eso probablemente suena un poco extraño.

El JavaScript más común que no se encuentra en JavaScript es la `API DOM`. Por ejemplo:

```
var el = document.getElementById( "foo" );
```

La variable de `document` existe como variable global cuando el código se ejecuta en un explorador. No es proporcionado por el motor JS, ni es particularmente controlado por la especificación de JavaScript. Toma la forma de algo que se parece mucho a un objeto JS normal, pero no es exactamente eso. Es un objeto especial, a menudo llamado "host object".

Por otra parte, el método `getElementById(..)` en el documento se parece a una función JS normal, pero es sólo una interfaz expuesta a un método incorporado proporcionado por el DOM de su navegador. En algunos navegadores (de nueva generación), esta capa también puede estar en JS, pero tradicionalmente el DOM y su comportamiento se implementa en algo más parecido a C / C ++.

Otro ejemplo es con entrada/salida (Input / Optput - I/O).

El favorito de todo el mundo `alert(...)` hace que aparezca un cuadro de mensaje en la ventana del navegador del usuario. `alert(...)` se proporciona a su programa JS por el navegador, no por el propio motor JS. La llamada que realiza envía el mensaje a los elementos internos del navegador y se encarga de dibujar y mostrar el cuadro de mensaje.

Lo mismo ocurre con `console.log(..)`; Su navegador proporciona estos mecanismos y los vincula a las herramientas de desarrollo.

Este libro, y toda esta serie, se centra en el lenguaje JavaScript. Es por eso que no ve ninguna cobertura sustancial de estos mecanismos JavaScript non-JavaScript. Sin embargo, usted necesita ser consciente de ellos, como será en cada programa JS que escriba!

2.10 Revisión

El primer paso para aprender el sabor de la programación en JavaScript es obtener una comprensión básica de sus mecanismos básicos como valores, tipos, closures functions, this y prototipos.

Por supuesto, cada uno de estos temas merece una cobertura mucho mayor de lo que has visto aquí, pero es por eso que tienen capítulos y libros dedicados a ellos durante el resto de esta serie. Después de sentirse bastante cómodo con los conceptos y las muestras de código en este capítulo, el resto de la serie lo espera para profundizar y conocer el lenguaje profundamente.

El último capítulo de este libro resumirá brevemente cada uno de los otros títulos de la serie y los otros conceptos que cubren, además de lo que ya hemos explorado.

3- En YDKJS

¿De qué trata esta serie? En pocas palabras, se trata de tomar en serio la tarea de aprender todas las partes de JavaScript, no sólo un subconjunto del lenguaje que alguien llamó "las partes buenas", y no cualquier cantidad mínima que necesita saber para hacer su trabajo.

Los desarrolladores serios en otros lenguajes esperan poner el esfuerzo de aprender la mayoría o todos los lenguajes en los que escriben, pero los desarrolladores de JS parecen destacarse de la multitud en el sentido de que normalmente no aprenden mucho del lenguaje. Esto no es algo bueno, y no es algo que deberíamos seguir permitiendo que sea la norma.

La serie You Do not Know JS (YDKJS) está en marcando contraste con los enfoques típicos para aprender JS, y es esa la diferencia de casi cualquier otro libro JS que lea. Te desafía a ir más allá de tu zona de confort y a hacer las más profundas, y preguntas de "por qué" para cada comportamiento que encuentres. ¿Estás preparado para ese desafío?

Voy a utilizar este capítulo final para resumir brevemente qué esperar del resto de los libros de la serie, y cómo ir más eficazmente sobre la construcción de una base solida de aprendizaje de JS en la parte superior de YDKJS.

3.1 Scope & Closures

Quizás una de las cosas más fundamentales que necesitará para llegar a un entendimiento rápido es cómo el scope de las variables realmente funcionan en JavaScript. No basta con tener creencias anecdóticas difusas sobre el scope.

El título Scope & Closures comienza por desacreditar la idea errónea común de que JS es un "lenguaje interpretado" y por lo tanto no se compila. Nope.

El motor JS compila su código justo antes (y a veces durante!) la ejecución. Así que buscamos una comprensión más profunda de la aproximación del compilador a nuestro código para entender cómo encuentra y trata las declaraciones de variables y funciones. A lo largo del camino, vemos la metáfora típica para la gestión del scope de variables en JS, "Hoisting".

Esta comprensión crítica del "alcance léxico" es en lo que entonces basamos nuestra exploración de los closures para el último capítulo del libro. Los closures son tal vez el concepto más importante en todos los de JS, pero si no ha comprendido con firmeza cómo funciona el scope, el closure probablemente permanecerá fuera de su alcance.

Una aplicación importante de closures es el patrón de module, como brevemente se introdujo en este libro en el capítulo 2. El patrón de módulo es quizás el patrón de organización de código más prevalente en todo el código JavaScript; La comprensión profunda de ella debe ser una de sus prioridades más altas.

3.1 This & Object Prototypes

Tal vez uno de los más difundidos y persistentes mistruths sobre JavaScript es que la palabra clave `this` se refiere a la función en la que aparece. Grave error.

La palabra clave `this` se enlaza dinámicamente en función de cómo se ejecuta la función en cuestión y resulta que hay cuatro reglas simples para comprender y determinar completamente la vinculación `this`.

Muy relacionado con la palabra clave `this` está el mecanismo de prototipo de objeto, que es una cadena de búsqueda de propiedades, similar a cómo se encuentran las variables de alcance de léxico. Pero envuelto en los prototipos es el otro gran error sobre JS: la idea de emular (fake) clases y (lo que se llama "prototipo") la herencia.

Desafortunadamente, el deseo de traer el pensamiento de patrones de diseño de clase y herencia a JavaScript es casi lo peor que puedes intentar hacer, porque mientras la sintaxis te engañe pensando que hay algo como clases presentes, de hecho el mecanismo prototipo es fundamentalmente opuesto en su comportamiento.

Lo que está en cuestión es si es mejor ignorar el desajuste y pretender que lo que está implementando es "herencia", o si es más apropiado aprender y abrazar cómo funciona realmente el prototipo del sistema. Esta última es más apropiadamente llamada "delegación de comportamiento".

Esto es más que una preferencia sintáctica. **La delegación es un patrón de diseño totalmente diferente, y más potente, que reemplaza la necesidad de diseñar con clases y herencia**. Sin embargo, estas afirmaciones vuelan absolutamente en la cara de casi todos los demás blog post, libros y conferencias que hablan sobre el tema de la totalidad de la vida de JavaScript.

Las afirmaciones que hago con respecto a la delegación versus la herencia vienen no de una aversión al lenguaje y su sintaxis, sino del deseo de ver la verdadera capacidad del lenguaje adecuadamente apalancado y la interminable confusión y frustración por fin borrada.

Pero el caso que hago con respecto a los prototipos y la delegación es mucho más complicado que lo que voy a disfrutar aquí. Si estás listo para reconsiderar todo lo que piensas que sabes sobre las "clases" de JavaScript y la "herencia", te ofrezco la oportunidad de "tomar la píldora roja" (Matrix 1999) y echa un vistazo a los capítulos 4-6 del Título This & de los prototipos de esta serie.

3.2 Tipos & Gramática

El tercer título de esta serie se centra principalmente en abordar otro tema muy polémico: la coerción de tipos. Tal vez ningún tema causa más frustración en los desarrolladores de JS que cuando se habla de las confusiones que rodean la coerción implícita.

Con mucho, la sabiduría convencional es que la coerción implícita es una "parte mala" del lenguaje y debe ser evitada a toda costa. De hecho, algunos han ido tan lejos como para llamarlo un "defecto" en el diseño del lenguaje. De hecho, hay herramientas cuyo trabajo completo es no hacer nada, sino escanear su código y quejarse si está haciendo algo incluso remotamente como coerción.

Pero, ¿es la coerción realmente tan confusa, tan mala, tan traicionera, que tu código está condenado desde el principio si lo usas?

Yo digo que no. Después de haber entendido cómo los tipos y valores realmente funcionan en los Capítulos 1-3, el Capítulo 4 asume este debate y explica completamente cómo funciona la coerción, en todos sus rincones y grietas. Vemos exactamente qué partes de la coerción realmente son sorprendentes y qué partes realmente tienen sentido completo si se les da el tiempo para aprender.

Pero no estoy simplemente sugiriendo que la coerción es sensible y aprendible, estoy afirmando que la coerción es una herramienta increíblemente útil y totalmente subestimada que debería usar en su código. Estoy diciendo que la coerción, cuando se utiliza correctamente, no sólo funciona, sino que hace que su código sea mejor. Todos los que niegan y dudan seguramente se burlan de tal posición, pero creo que es una de las claves principales para aumentar su juego en JS.

¿Quieres seguir lo que dice la multitud, o estás dispuesto a dejar todas las suposiciones a un lado y mirar la coerción con una nueva perspectiva? El título Tipos y Gramática de esta serie coaccionará su pensamiento.

3.4 Async & Performance

Los tres primeros títulos de esta serie se centran en la mecánica básica del lenguaje, pero el cuarto título se ramifica ligeramente para cubrir los patrones de la mecánica del lenguaje para gestionar la programación asincrónica. La asincronía no sólo es crítica para el rendimiento de nuestras aplicaciones, sino que se está convirtiendo cada vez más en un factor crítico en la capacidad de escritura y mantenimiento.

El libro comienza primero por aclarar una gran cantidad de terminología y confusión de conceptos en torno a cosas como "asíncrona", "paralela" y "concurrente", y explica en profundidad cómo estas cosas se aplican y no se aplican a JS.

Entonces nos movemos a examinar las devoluciones de llamada (callbacks) como el método primario de permitir la asincronía. Pero es aquí que vemos rápidamente que las callbacks por sí solas son irremediablemente insuficientes para las demandas modernas de la programación asíncrona. Identificamos dos deficiencias principales de las devoluciones de llamada: sólo la codificación: pérdida de confianza de inversión de control (IoC) y falta de capacidad lineal razonable .

Para abordar estas dos grandes deficiencias, ES6 introduce dos nuevos mecanismos (y, de hecho, patrones): las promesas y los generadores (promises y generators)

Las promesas son una envoltura independiente del tiempo alrededor de un "valor futuro", que le permite razonar y componer independientemente de si el valor está listo o no. Por otra parte, resuelven eficazmente las ediciones del trust de IoC encaminando devoluciones de llamada a través de un mecanismo confiables y componibles de la promesa.

Los generadores introducen un nuevo modo de ejecución para las funciones JS, con lo que el generador puede pausarse en los puntos de rendimiento y reanudarse asincrónicamente más tarde. La capacidad de pausa y reanudación permite que el código de búsqueda secuencial y secuencial en el generador se procese asincrónicamente detrás de escena. Haciendo esto, abordamos las confusiones de devoluciones de llamada no lineales, no locales y, por lo tanto, hacemos que nuestro código asincrónico sincronice el código para ser más razonable.

Pero es la combinación de promesas y generadores que "rinde (yields)" nuestro patrón de codificación asíncrono más efectivo hasta la fecha en JavaScript. De hecho, gran parte de la sofisticación futura de la asincronía que viene en ES7 y después se construirá seguramente sobre esta base. Para ser serio acerca de la programación eficaz en un mundo asíncrono, usted va a tener que sentirse muy cómodo con la combinación entre promesas y generadores.

Si las promesas y los generadores tienen que ver con la expresión de patrones que permiten que nuestros programas funcionen mejor al mismo tiempo y así lograr más procesamiento en un período más corto, JS tiene muchas otras facetas de optimización de rendimiento que vale la pena explorar.

El capítulo 5 profundiza en temas como el paralelismo de programas con los Workers Web y el paralelismo de datos con SIMD, así como técnicas de optimización de bajo nivel como ASM.js. El Capítulo 6 echa un vistazo a la optimización del rendimiento desde la perspectiva de técnicas de benchmarking apropiadas, incluyendo por qué tipo de rendimiento preocuparse y qué ignorar.

Escribir JavaScript significa efectivamente escribir código que puede romper las restricciones de estar ejecutado dinámicamente en una amplia gama de navegadores y otros entornos. Requiere una gran cantidad de intrincada y detallada planificación y esfuerzo de nuestras partes para llevar un programa de "funciona" a "funciona bien".

El título Async & Performance está diseñado para ofrecerte todas las herramientas y habilidades que necesitas para escribir código JavaScript razonable y eficaz.

3.5 ES6 & Más allá

No importa cuánto sientas que has dominado JavaScript hasta este punto, la verdad es que JavaScript nunca va a dejar de evolucionar, y además, la tasa de evolución está aumentando rápidamente. Este hecho es casi una metáfora para el espíritu de esta serie, para abrazar que nunca sabremos completamente todas las partes de JS, porque tan pronto como lo domines todo, habrá cosas nuevas online que vas a necesitar aprender.

Este título está dedicado tanto a las visiones a corto y mediano plazo de donde se dirige el lenguaje, no sólo las cosas conocidas como ES6, sino las cosas probables del más allá.

Mientras que todos los títulos de esta serie abarcan el estado de JavaScript en el momento de escribir este artículo, que está a medio camino con la adopción de ES6, el foco principal en la serie ha sido más en ES5. Ahora, queremos dirigir nuestra atención a ES6, ES7 y ...

ES6 & Beyond comienza dividiendo el material de hormigón del paisaje ES6 en varias categorías clave, incluyendo nueva sintaxis, nuevas estructuras de datos (colecciones) y nuevas capacidades de procesamiento y API. Cubrimos cada una de estas nuevas características de ES6, en diversos niveles de detalle, incluyendo la revisión de detalles que se mencionan en otros libros de esta serie.

Algunas cosas emocionantes ES6 para mirar hacia adelante es leer acerca de: desestructuración, valores de parámetros por defecto, símbolos, métodos concisos, propiedades calculadas, funciones de flecha, bloque de alcance, promesas, generadores, iteradores, módulos, proxies, weakmaps y mucho, mucho más! Phew, ES6 packs un buen golpe!

La primera parte del libro es una hoja de ruta para todas las cosas que usted necesita aprender para prepararse para lo nuevo y lo mejorado que va a escribir JavaScript y explorarlo en los próximos dos años.

La última parte del libro da vuelta la atención brevemente a la mirada en las cosas que podemos probablemente esperar ver en el futuro próximo de Javascript. La realización más importante aquí es que después de ES6, JS es probable que vaya a evolucionar característica por característica en lugar de versión por versión, lo que significa que podemos esperar para ver estas cosas en un futuro cercano mucho más pronto de lo que pueda imaginar.

El futuro de JavaScript es brillante. ¿No es hora de que empecemos a aprenderlo?

3.6 Revisión

La serie YDKJS está dedicada a la proposición de que todos los desarrolladores de JS pueden y deben aprender todas las partes de este gran lenguaje. La opinión de alguna persona, las premisas sobre el framework y el plazo del proyecto no deben ser la excusa para que nunca aprendas y entiendas profundamente el JavaScript.

Tomamos cada área importante de enfoque en el lenguaje y dedicar un libro corto pero muy denso para explorar todas las partes de ella que tal vez pensó que sabía pero probablemente no lo hizo plenamente.

"You Do not Know JS" no es una crítica o un insulto. Es una comprensión de que todos nosotros, yo incluido, debemos llegar a un acuerdo. Aprender JavaScript no es un objetivo final, sino un proceso. No sabemos JavaScript, todavía. ¡Pero lo haremos!

II- Scope & Closures

Prefacio

Tuve el honor de escribir el prólogo del primer libro, Scope & Closures, de la serie You Do not Know JS de Kyle Simpson. Le ruego que compre el libro, definitivamente vale la pena leerlo sin importar su habilidad o experiencia, pero también he incluido el prólogo abajo.

Cuando yo era un niño pequeño, a menudo me gustaba separar las cosas y volver a juntarlas. Teléfonos móviles antiguos, equipos de música Hi-Fi y cualquier otra cosa que pudiera poner en mis manos. Yo era demasiado joven para usar realmente estos dispositivos, pero cada vez que se rompían, me preguntaba instantáneamente si podía averiguar cómo funcionaba.

Recuerdo que una vez vi una placa de circuito para una radio vieja. Tenía este extraño tubo largo con alambre de cobre envuelto alrededor de él. No pude resolver su propósito, pero inmediatamente entré en el modo de investigación. ¿Qué hace? ¿Por qué está en una radio? No se parece a las otras partes de la placa de circuito, ¿por qué? ¿Por qué tiene cobre envuelto alrededor? ¿Qué sucede si retiro el cobre? Ahora sé que era una antena de bucle, hecha por el cable de cobre envolvente alrededor de una barra de ferrita, que se utilizan a menudo en radios de transistores.

¿Alguna vez se convirtió en adicto a averiguar todas las respuestas a cada pregunta 'por qué'? La mayoría de los niños lo hacen. De hecho, es probablemente mi cosa favorita acerca de los niños - su deseo de aprender.

Desafortunadamente, ahora soy considerado un "profesional" y paso mis días haciendo cosas. Cuando yo era joven, me encantaba la idea de re-hacer un día las cosas que deshice. Por supuesto, la mayoría de las cosas que hago ahora son con JavaScript y no con barras de ferrita ... pero lo suficientemente cerca! Sin embargo, a pesar de amar una vez la idea de hacer cosas, ahora me encuentro anhelando el deseo de resolver las cosas. Seguro - a menudo averiguo la mejor manera de resolver un problema o de corregir un error, pero rara vez me tomo el tiempo para cuestionar mis herramientas.

Y eso es exactamente por lo que estoy tan entusiasmado con la serie de libros de Kyle "You Do not Know JS". Porque tiene razón. No conozco JS. Utilizo Javascript en el dia a dia, de adentro hacia fuera y lo he hecho por muchos años, pero lo entiendo realmente? No. Claro, entiendo mucho y a menudo leo las especificaciones y las listas de correo, pero no, no entiendo tanto .

Scope y Closures, es un comienzo brillante de la serie. Está muy bien dirigido a personas como yo (y espero que hacia ti también), no enseña JavaScript como si nunca lo hubieras usado, y te hace darte cuenta de lo poco que sabes sobre el funcionamiento interno.

También está saliendo en el momento perfecto, ES6 finalmente se está estableciendo y la aplicación a través de los navegadores va bien. Si aún no has aprendido las nuevas características (como `let` y `const`), este libro será una gran introducción.

Así que espero que disfruten de este libro, pero más que la forma en que Kyle piensa críticamente acerca de cómo funciona cada pequeño fragmento del lenguaje, fluirá en su forma de pensar y flujo de trabajo en general. En lugar de utilizar la antenna, averiguar cómo y por qué funciona.

0- Prefacio

Estoy seguro de que se dio cuenta, pero "JS" en el título de la serie de libros no es una abreviatura de palabras utilizadas para maldecir sobre JavaScript, aunque maldecir a las peculiaridades del lenguaje es algo con lo que probablemente todos se pueden identificar.

Desde los primeros días de la web, JavaScript ha sido una tecnología fundamental que ha impulsado la experiencia interactiva en torno al contenido que consumimos. Mientras que el apuntador parpadeante y los molestos avisos emergentes podrían ser donde comenzó JavaScript, casi dos décadas después, la tecnología y la capacidad de JavaScript ha crecido en muchos ámbitos de magnitud, y pocos dudan de su importancia en el corazón de la plataforma de software más ampliamente disponible: La web.

Pero como lenguaje, ha sido perpetuamente un blanco para mucha crítica, debido en parte a su "herencia", pero más aún debido a su filosofía de diseño. Incluso el nombre evoca, como Brendan Eich una vez lo dijo, "el hermano bebe estúpido" comparado junto a su hermano mayor más maduro "Java". Pero el nombre es simplemente un accidente de la política y el marketing. Los dos idiomas son muy diferentes en muchos aspectos importantes. "JavaScript" está relacionado con "Java" como "Carnival" es a "Car".

Debido a que JavaScript toma conceptos y sintaxis idiomáticos de varios idiomas, incluyendo orgullosas raíces procedimentales de estilo C, así como raíces funcionales sutiles, menos obvias de estilo Scheme/Lisp, es sumamente accesible a una amplia audiencia de desarrolladores, incluso aquellos con poca o sin experiencia en programación. El "Hello World" de JavaScript es tan simple que el idioma se hace atractivo y es fácil de ponerse cómodo con la comprensión temprana.

Mientras que JavaScript es quizás uno de los idiomas más fáciles de poner en funcionamiento, sus excentricidades hacen que el dominio sólido del lenguaje sea una ocurrencia mucho menos común que en muchos otros idiomas. Cuando se necesita un conocimiento bastante profundo de un lenguaje como C o C++ para escribir un programa a gran escala, la producción a gran escala de JavaScript puede, y con frecuencia lo es, apenas se raya con la superficie de lo que el lenguaje puede realmente hacer.

Los conceptos sofisticados, que están profundamente arraigados en el lenguaje, tienden a aparecer en formas aparentemente simplistas, como pasar las funciones como devoluciones de llamada, lo que anima al desarrollador de JavaScript a utilizar el lenguaje tal como está y ha no preocuparse demasiado por lo que está pasando bajo la capucha.

Es simultáneamente un lenguaje sencillo y fácil de usar que tiene un amplio atractivo y una colección compleja y matizada de mecánica del lenguaje que sin un estudio cuidadoso escapará a la verdadera comprensión, incluso de los más experimentados desarrolladores de JavaScript.

Ahí radica la paradoja de JavaScript, el talón de Aquiles de la lengua, el desafío que estamos abordando actualmente. Debido a que JavaScript se puede utilizar sin comprender, la comprensión del lenguaje a menudo nunca se logra.

Misión

Si en cada punto que encuentres una sorpresa o frustración en JavaScript, tu respuesta es añadirlo a la lista negra, como algunos están acostumbrados a hacer, pronto serás relegado a una concha hueca de la riqueza de JavaScript.

Si bien este subconjunto ha sido conocido como "The Good Parts", le pediría a usted, querido lector, que lo considere "The Easy Parts", "The Safe Parts" o incluso "The Incomplete Parts".

Esta serie de libros de Javascript ofrece un desafío contrario: aprenda y entienda profundamente todo Javascript, incluso y especialmente "las piezas resistentes (The Tough Parts)".

Aquí, nos dirigimos a los desarrolladores de JS que tienen en la cabeza la mentalidad de aprender "lo suficiente" para continuar, sin nunca obligarse a aprender exactamente cómo y por qué el lenguaje se comporta de la manera que lo hace. Además, evitamos el consejo común de retirarse cuando el camino se vuelve áspero.

No estoy contento, ni debes estar, en parar una vez que algo funciona correctamente, y no saber realmente por qué. Le desafío a viajar por ese "camino costoso" y abrazar todo lo que JavaScript es y puede hacer. Con ese conocimiento, ninguna técnica, ningún framework, ningún acrónimo popular de moda, estará más allá de su comprensión.

Estos libros toman partes específicas del lenguaje que son las comúnmente mal entendidas o no comprendidas, y se sumerge muy profundamente y exhaustivamente en ellas. Usted debe termina la lectura con una firme confianza en su comprensión, no sólo de lo teórico, sino lo práctico "lo que necesita saber".

El JavaScript que usted conoce ahora mismo es probablemente la partes dada a usted por otros que han sido "quemados" por una comprensión incompleta. Ese JavaScript es sólo una sombra del verdadero lenguaje. Realmente no sabes JavaScript, pero si crees en esta serie, lo harás. Sigue leyendo, amigos. JavaScript te espera.

Resumen

JavaScript es impresionante. Es fácil de aprender en parte, y mucho más difícil de aprender por completo (o incluso lo suficiente). Cuando los desarrolladores encuentran confusión, suelen culpar al idioma en lugar de su falta de comprensión. Estos libros apuntan a arreglar eso, inspirando una apreciación fuerte del lenguaje para que usted pueda entenderlo ahora, y deba, profundamente entenderlo.

Nota: Muchos de los ejemplos en este libro asumen los entornos de motor de JavaScript modernos (y futuros), como ES6. Es posible que algunos códigos no funcionen como se describe si se ejecutan en motores anteriores (pre-ES6).

1- ¿Qué es el Scope (Ámbito)?

Uno de los paradigmas más fundamentales de casi todos los lenguajes de programación es la capacidad de almacenar valores en variables, y posteriormente recuperar o modificar esos valores. De hecho, la capacidad de almacenar valores y extraer valores de variables es lo que da un estado de programa.

Sin tal concepto, un programa podría realizar algunas tareas, pero serían extremadamente limitados y terriblemente poco interesantes.

Pero la inclusión de variables en nuestro programa engendra las preguntas más interesantes que ahora abordaremos: ¿dónde viven esas variables? En otras palabras, ¿dónde están almacenadas? Y, lo más importante, ¿cómo nuestro programa los encuentra cuando los necesita?

Estas preguntas hablan de la necesidad de un conjunto bien definido de reglas para almacenar variables en algún lugar, y para encontrar esas variables en un momento posterior. Llamaremos a ese conjunto de reglas: Scope (Ámbito).

Pero, ¿dónde y cómo se establecen estas reglas de ámbito?

1.2 Teoría del Compilador

Puede ser evidente por sí mismo, o puede ser sorprendente, dependiendo de su nivel de interacción con varios idiomas, pero a pesar de que JavaScript pertenece a la categoría general de lenguajes "dinámicos" o "interpretados", es de hecho un lenguaje compilado. No se compila con suficiente antelación, al igual que muchos lenguajes tradicionalmente compilados, ni tampoco los resultados de la compilación son portables entre varios sistemas distribuidos.

Pero, sin embargo, el motor de JavaScript realiza muchos de los mismos pasos, aunque de formas más sofisticadas de lo que comúnmente se sabe, que cualquier compilador de lenguaje tradicional.

En el proceso tradicional del lenguaje compilado, un fragmento del código fuente, su programa, experimentará típicamente tres pasos antes de que se ejecute, comúnmente llamado "compilación":

1. **Tokenizing / Lexing**: romper una cadena de caracteres en pedazos significativos (al lenguaje), llamados tokens. Por ejemplo, considere el programa: `var a = 2;`. Este programa probablemente se dividiría en los siguientes tokens: `var`, `a`, `=`, `2`, y `;`. Los espacios en blanco pueden o no ser persistidos como un símbolo, dependiendo de si es significativo o no.

   Nota: La diferencia entre tokenizing y lexing es sutil y académica, pero se centra en si estos símbolos se identifican de manera apátrida o con estado. En pocas palabras, si el tokenizer invocara reglas de análisis sintáctico para averiguar si `a` debería considerarse un token distinto o simplemente parte de otro token, eso sería lexing

2. **Parsing**: toma una corriente (array) de tokens y lo convierte en un árbol de elementos anidados, que colectivamente representan la estructura gramatical del programa. Este árbol se denomina "AST (Abstract Syntax Tree)" (Árbol de sintaxis abstracto).

   El árbol para `var a = 2`; Puede comenzar con un nodo de nivel superior llamado `VariableDeclaration`, con un nodo secundario llamado `Identifier` (cuyo valor es `a`), y otro llamado `AssignmentExpression` que tiene un hijo llamado `NumericLiteral` (cuyo valor es `2`).

3. **Code-Generation**: es el proceso de tomar un AST y convertirlo en código ejecutable. Esta parte varía mucho dependiendo del lenguaje, la plataforma a la que se está dirigiendo, etc.

Por lo tanto, en lugar de quedar atascados en los detalles, sólo vamos a mano y decir que hay una manera de tomar nuestro AST descrito anteriormente para `var a = 2;` y convertirlo en un conjunto de instrucciones de máquina para crear realmente una variable llamada `a` (incluida la reserva de memoria, etc.), y luego almacenar un valor en `a`.

Nota: Los detalles de cómo el motor gestiona los recursos del sistema son más profundos de lo que cavaremos, por lo que sólo daremos por sentado que el motor es capaz de crear y almacenar variables según sea necesario.

El motor JavaScript es mucho más complejo que estos tres pasos, al igual que la mayoría de los compiladores de otros lenguajes. Por ejemplo, en el proceso de análisis y generación de código, hay ciertamente pasos para optimizar el rendimiento de la ejecución, incluyendo el colapso de elementos redundantes, etc.

Por lo tanto, estoy pintando sólo con trazos amplios aquí. Pero creo que verá pronto por qué estos detalles que cubrimos, incluso a un nivel alto, son relevantes.

Por un lado, los motores de JavaScript no obtienen el lujo (como otros compiladores de lenguaje) de tener un montón de tiempo para optimizar, porque la compilación de JavaScript no ocurre en un paso de construcción con antelación, como si ocurre con otros lenguajes.

Para JavaScript, la compilación sucede, en muchos casos, en microsegundos (o menos) antes de ejecutar el código. Para garantizar un rendimiento más rápido, los motores JS utilizan todo tipo de trucos (como JITs, compilación perezosa e incluso re-compilación en caliente, etc.) que están mucho más allá del "alcance" de nuestra discusión aquí.

Digamos, por simplicidad, que cualquier fragmento de JavaScript tiene que ser compilado antes (normalmente antes de que!) se ejecute. Por lo tanto, el compilador JS tomará el programa `var a = 2;` y lo compila primero, y luego esta listo para ejecutarlo, normalmente de inmediato.

1.2 Entendiendo el Scope

La forma en que abordaremos el aprendizaje sobre el scope es pensar en el proceso en términos de una conversación. Pero, ¿quién está teniendo la conversación?

El elenco

Vamos a conocer el elenco de personajes que interactúan para procesar el programa `var a = 2;`, por lo que entendemos sus conversaciones que escucharemos en breve:

1. Motor: responsable de la compilación y ejecución de principio a fin de nuestro programa JavaScript.
2. Compilador: uno de los amigos del Motor; Maneja todo el trabajo sucio de análisis y generación de código (ver sección anterior).
3. Scope: otro amigo del Motor; Recopila y mantiene una lista de búsqueda de todos los identificadores declarados (variables), y aplica un conjunto estricto de reglas sobre cómo estos son accesibles para el código que se está ejecutando actualmente.

Para que entiendas completamente cómo funciona JavaScript, debes comenzar a pensar como Motor (y amigos), hacer las preguntas que hacen y responder a esas preguntas de la misma manera.

Atrás y adelante

Cuando vea el programa `var a = 2;`, lo más probable es pensar en eso como una sola declaración. Pero no es así como nuestro nuevo amigo Motor lo ve. De hecho, Motor ve dos declaraciones distintas, una que el compilador manejará durante la compilación, y una que el motor manejará durante la ejecución.

Así que, vamos a desglosar cómo Motor y amigos se acercará al programa `var a = 2` ;.

Lo primero que Compilador hará con este programa es realizar lexing para dividirlo en fichas, que luego analizará en un árbol. Pero cuando Compilador llega a la generación de código, tratará este programa de manera algo diferente de lo que se supone.

Una suposición razonable sería que el compilador producirá código que podría ser resumido por este pseudo-código: "Asignar memoria para una variable, etiquetar la `a`, y luego pegar el valor `2` en esa variable". Desafortunadamente, eso no es muy exacto.

El compilador procederá como sigue:

1. Encuentro `var a`, Compilador pregunta a Scope para ver si ya existe una variable `a`

para esa colección de ámbito en particular. Si es así, el compilador omite esta declaración y sigue adelante. De lo contrario, el compilador solicita a Scope que declare una nueva variable llamada `a` para esa colección de ámbito.
2. El Compilador entonces produce código para que el motor ejecute más adelante, para manejar la asignación `a = 2`. El código que Motor ejecuta primero preguntará a Scope si hay una variable llamada accesible en la colección de alcance actual. Si es así, el motor utiliza esa variable. Si no, el motor busca en otra parte (vea la sección de ámbito anidada abajo).

Si finalmente el Motor encuentra una variable, le asigna el valor `2`. ¡Si no, el motor levantará su mano y gritará hacia fuera un error!

En resumen, se toman dos acciones distintas para una asignación de variables: Primero, el Compilador declara una variable (si no se ha declarado previamente en el ámbito actual) y segundo, al ejecutar, Motor busca la variable en Scope y la asigna a ella, si se encuentra.

Compiler Speak

Necesitamos un poco más de terminología del compilador para continuar con la comprensión.

Cuando el Motor ejecuta el código que el Compilador produjo para el paso (2), tiene que buscar la variable `a` para ver si se ha declarado, y esta búsqueda está consultando el Scope. Pero el tipo de Motor de búsqueda que realiza afecta el resultado de la búsqueda.

En nuestro caso, se dice que Motor estaría realizando una búsqueda "LHS" para la variable `a`. El otro tipo de búsqueda se llama "RHS".

Apuesto a que usted puede adivinar lo que la "L" y "R" significa. Estos términos representan "Lado Izquierdo" y "Lado Derecho".

Lado ... ¿de qué? **De una operación de asignación**.

En otras palabras, una búsqueda LHS se hace cuando aparece una variable en el lado izquierdo de una operación de asignación y una búsqueda RHS se hace cuando aparece una variable en el lado derecho de una operación de asignación.

En realidad, seamos un poco más precisos. Una búsqueda de RHS es indistinguible, para nuestros propósitos, de simplemente una mirada hacia arriba del valor de alguna variable, mientras que la búsqueda de LHS está tratando de encontrar el contenedor de la variable en sí, para que pueda asignar. De esta manera, RHS no significa realmente "lado derecho de una asignación" per se, sino que, más exactamente, significa "no a la izquierda".

Siendo ligeramente glib por un momento, también podría pensar "RHS" en su lugar significa "recuperar su fuente (valor)", lo que implica que RHS significa "ir a buscar el valor de ...".

Vamos a cavar en eso más profundo.

Cuando yo digo:

```
console.log( a );
```

La referencia `a` es una referencia RHS, porque nada está siendo asignado aun aquí. En su lugar, estamos buscando para recuperar el valor de `a`, por lo que el valor se puede pasar a `console.log(..)`.

En contraste:

```
a = 2;
```

La referencia `a` aquí es una referencia LHS, porque realmente no nos importa cuál es el valor actual, simplemente queremos encontrar la variable como un objetivo para la operación de asignación `= 2`.

Nota: LHS y RHS que significan "Lado izquierdo / derecho de una asignación" no significa necesariamente literalmente "lado izquierdo / derecho del operador de asignación". Hay varias otras maneras en que ocurren las asignaciones, por lo que es mejor pensar conceptualmente en ella como: "*quién es el objetivo de la asignación (LHS) - estoy asignando*" y "*quién es la fuente de la asignación (RHS) - estoy buscando*".

Considere este programa, que tiene tanto referencias LHS y RHS :

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

La última línea que invoca `foo(..)` como una llamada de función requiere una referencia RHS a `foo`, que significa "ir a buscar el valor de `foo`, y dármelo". Además, `(..)` significa que el valor de `foo` debe ser ejecutado, por lo que sería en realidad una función!

Hay una asignación sutil pero importante aquí. ¿Lo viste?

Es posible que haya omitido el implícito `a = 2` en este fragmento de código. Sucede cuando el valor `2` se pasa como argumento a la función `foo(...)`, en cuyo caso el valor `2` se asigna al parámetro `a`. Para asignar (implícitamente) al parámetro `a`, se realiza una búsqueda LHS.

También hay una referencia RHS para el valor de `a`, y ese valor resultante se pasa a `console.log(..)`. `console.log(..)` necesita una referencia para ejecutarse. Es una búsqueda de RHS para el objeto de `console`, entonces se produce una resolución de propiedad para ver si tiene un método llamado `log`.

Por último, podemos conceptualizar que hay un intercambio LHS / RHS de pasar el valor `2` (a través de la variable `a` de búsqueda RHS) en `log(..)`. Dentro de la implementación nativa de `log(..)`, podemos suponer que tiene parámetros, el primero de los cuales (tal vez llamado `arg1`) tiene una referencia de referencia LHS, antes de asignar `2` a ella.

Nota: Es posible que se sienta tentado a conceptualizar la función de declaración de `function foo(a) {...` como una declaración y asignación de variables normales, como `var foo` y `foo = function(a) {...`. Al hacerlo, sería ser tentador pensar que esta declaración de función implica una búsqueda de LHS.

Sin embargo, la diferencia sutil pero importante es que el Compilador maneja la declaración y la definición de valor durante la generación de código, de modo que cuando el Motor está ejecutando el código, no hay ningún procesamiento necesario para "asignar" un valor de función a `foo`. Por lo tanto, no es realmente apropiado pensar en una declaración de función como una asignación de búsqueda LHS en la forma en que estamos discutiendo aquí.

Conversación entre el Motor y el Scope

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Imaginemos el intercambio anterior (que procesa este fragmento de código) como una conversación. La conversación sería algo como esto:

> Motor: Hey Scope, tengo una referencia RHS para `foo`. ¿Has oído hablar de él?
>
> Scope: Por qué? sí, lo he hecho. El Compilador lo declaró hace apenas un segundo. Es una función. Aqui tienes.
>
> Motor: Genial, gracias! OK, estoy ejecutando `foo`.
>
> Motor: Hey, Scope, tengo una referencia de LHS para `a`, alguna vez oído hablar de él?
>
> Scope: Por qué sí?, lo he hecho. El Compilador lo declaró como un parámetro formal para `foo` hace poco. Aqui tienes.
>
> Motor: Útil como siempre, Scope. Gracias de nuevo. Ahora, es tiempo para asignar `2` a `a`.
>
> Motor: Oye, Scope, siento molestarte de nuevo. Necesito una búsqueda de RHS para la `console`. ¿Has oído hablar de él?
>
> Scope: No hay problema, Motor, esto es lo que hago todo el día. Sí, tengo `console`. Está integrado. Aqui tienes.
>
> Motor: Perfecto. Buscando el `log(..)`. OK, genial, es una función.
>
> Motor: Yo, Scope. ¿Puedes ayudarme con una referencia de RHS a `a`. Creo que lo recuerdo, pero sólo quiero comprobarlo.
>
> Scope: Usted tiene razón, Motor. El mismo tipo, no ha cambiado. Aqui tienes.
>
> Motor: Cool. Pasando el valor de `a`, que es `2`, en `log(..)`.
>
> ...

Examen

Compruebe su comprensión hasta ahora. Asegúrese de jugar con la parte de Motor y tener una "conversación" con el Scope:

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

var c = foo( 2 );
```

1. Identifique todas las consultas LHS (hay 3!).
2. Identifique todas las consultas RHS (hay 4!).

Nota: ¡Consulte la revisión del capítulo para las respuestas del examen!

1.3 Scopes Anidados

Dijimos que Scope es un conjunto de reglas para buscar variables por su nombre de identificador. Sin embargo, generalmente hay más de un ámbito a considerar.

Así como un bloque o función está anidado dentro de otro bloque o función, los ámbitos están anidados dentro de otros ámbitos. Por lo tanto, si no se puede encontrar una variable en el ámbito inmediato, Motor consulta el siguiente ámbito externo que contiene, hasta que se encuentre o hasta que se alcance el acope más externo (aka, global).

Considere:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

La referencia RHS para `b` no se puede resolver dentro de la función `foo`, pero puede resolverse en el ámbito que lo rodea (en este caso, global).

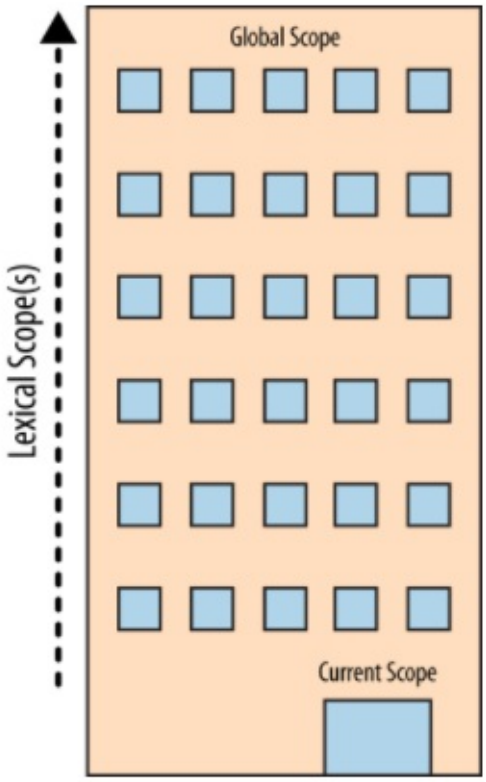
Así que, revisitando las conversaciones entre Motor y Scope, escucharíamos:

> Motor: "Oye, Scope de `foo`, alguna vez oído hablar de `b`? Tengo una referencia de RHS para el."
>
> Scope: "No, nunca he oído hablar de él"
>
> Motor: "Oye, Scope fuera de `foo`, oh! eres el alcance global, ok cool. ¿Has oído hablar de `b`? Tengo una referencia de RHS para el."
>
> Scope: "Sí, claro que sí"

Las reglas simples para atravesar un Scope anidado: Motor se inicia en el scope que está actualmente en ejecución, busca la variable allí, luego si no se encuentra, sigue subiendo un nivel, y así sucesivamente. Si se alcanza el alcance global más externo, la búsqueda se detiene, si encuentra o no la variable.

Construyendo sobre metáforas

Para visualizar el proceso de resolución anidada de un Scope, quiero que pienses en este alto edificio.



El edificio representa el conjunto de reglas de scope anidado de nuestro programa. El primer piso del edificio representa el ámbito actualmente en ejecución, dondequiera que se encuentre. El nivel superior del edificio es el alcance global.

Usted resuelve las referencias de LHS y RHS mirando su piso actual, y si no lo encuentra, toma el ascensor al siguiente piso, buscando allí, luego al siguiente, y así sucesivamente. Una vez que llegas a la planta superior (el alcance global), o bien encuentras lo que estás buscando, o no lo haces. Pero tienes que parar en algún momento.

1.4 Errores

¿Por qué importa si lo llamamos LHS o RHS?

Debido a que estos dos tipos de consultas se comportan de manera diferente en las circunstancias en las que la variable aún no ha sido declarada (no se encuentra en ningún ámbito consultado).

Considere:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

Cuando la búsqueda RHS ocurre para `b` la primera vez, no se encuentra. Se dice que esto es una variable "no declarada", porque no se encuentra en el ámbito.

Si una búsqueda RHS falla al encontrar una variable, en cualquier lugar de los Scopes anidados, esto resulta en un `ReferenceError` que está siendo lanzado por el Motor. Es importante tener en cuenta que el error es del tipo `ReferenceError` .

Por el contrario, si el Motor está realizando una búsqueda LHS y llega a la planta superior (ámbito global) sin encontrarlo, y si el programa no se está ejecutando en "modo estricto" [^note-strictmode], entonces el ámbito global creará una nueva variable de ese nombre en el ámbito global y la devolverá al Motor.

> "No, no había una antes, pero fui útil y creé una para ti".

El "Modo estricto" [^note-strictmode], que se agregó en ES5, tiene una serie de comportamientos diferentes del modo normal/relajado/perezoso. Una de estas conductas es que no permite la creación de la variable global automática/implícita. En ese caso, no habría ninguna variable Scope'd global a devolver de una búsqueda LHS, y el Motor lanzaría un `ReferenceError` similarmente al caso RHS.

Ahora, si se encuentra una variable para una búsqueda RHS, pero se intenta hacer algo con su valor que es imposible, como intentar ejecutar como función un valor sin función o hacer referencia a una propiedad en un valor `null` o u `ndefined` , entonces el Motor arroja un tipo diferente de error, llamado `TypeError` .

`ReferenceError` es Scope resolution-failure related, mientras que `TypeError` implica que la resolución Scope tuvo éxito, pero que se ha intentado una acción ilegal/imposible contra el resultado.

1.5 Revisión (TL; DR)

El Scope es un conjunto de reglas que determina dónde y cómo se puede buscar una variable (identificador). Esta búsqueda puede ser con el propósito de asignar a la variable, que es una referencia LHS (izquierda), o puede ser con el propósito de recuperar su valor de referencia, que es un RHS (lado derecho).

Las referencias LHS resultan de las operaciones de asignación. Las asignaciones relacionadas con el ámbito pueden ocurrir con el operador `=` o pasando argumentos a (asignar a) parámetros de función.

El Motor de JavaScript primero compila el código antes de que se ejecute, y al hacerlo, divide declaraciones como `var a = 2` ; En dos pasos separados:

1. Primero, `var a` para declararlo en ese ámbito. Esto se realiza al principio, antes de la ejecución del código.
2. Después, `a = 2` para buscar la variable (referencia LHS) y asignarla si se encuentra.

Las consultas de referencia de LHS y RHS comienzan en el ámbito actualmente en ejecución, y si es necesario (es decir, no encuentran lo que buscan allí), siguen su camino hasta el ámbito anidado, un ámbito (piso) a la vez, buscando el identificador, hasta que lleguen al global (piso superior) y se detengan, y lo encuentren, o no lo hagan.

Las referencias no encontradas de RHS dan lugar a la emisión de `ReferenceErrors` . Las referencias no cumplidas de LHS resultan en una variable global automática creada implícitamente de ese nombre (si no en "Strict Mode" [^note-strictmode]), o un `ReferenceError` (si en "Strict Mode" [^note-strictmode]).

Respuestas del Examen

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

var c = foo( 2 );
```

1. Identifique todas las busquedas LHS (hay 3!).

   `c = ..` , `a = 2` (asignación implícita de parámetros) y `b = ..`

2. Identificar todas las consultas RHS (hay 4!)

`foo(2..` , `= a;` , `a + ..` y `.. + b`

[^note-strictmode]: MDN: Strict Mode

2- Lexical Scope (Ámbito Léxico)

En el capítulo 1, definimos "scope" como el conjunto de reglas que gobiernan cómo el motor puede buscar una variable por su nombre de identificador y buscarlo, ya sea en el ámbito actual o en cualquiera de los ámbitos anidados que contiene.

Hay dos modelos predominantes de cómo funciona el scope. El primero de ellos es, con mucho, el más común, utilizado por la gran mayoría de lenguajes de programación. Se llama Ámbito Léxico, y lo examinaremos en profundidad. El otro modelo, que sigue siendo utilizado por algunos lenguajes (como Bash scripting, algunos modos en Perl, etc.) se llama Dynamic Scope.

El alcance dinámico está cubierto en el Apéndice A. Lo menciono aquí sólo para proporcionar un contraste con el Ámbito Léxico, que es el modelo de ámbito que emplea JavaScript.

2.1 Tiempo de Lex

Como discutimos en el capítulo 1, la primera fase tradicional de un Compilador de lenguaje estándar se llama lexing (aka, tokenizing). Si recuerda, el proceso de lexing examina una cadena de caracteres de código fuente y asigna significado semántico a los tokens como resultado de algún análisis de estado.

Es este concepto el que proporciona la base para entender qué es el ámbito léxico y de dónde proviene el nombre.

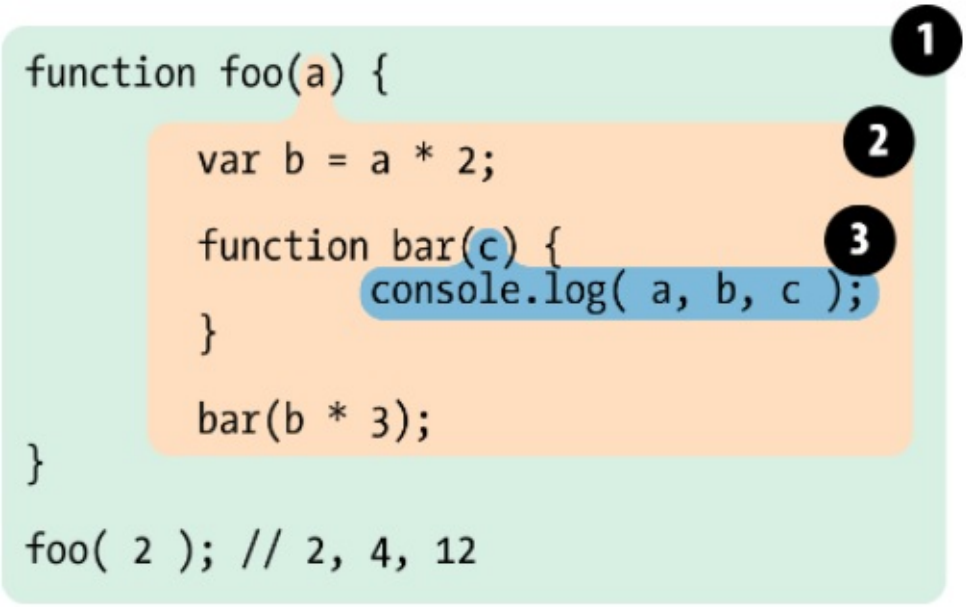
Para definirlo, el ámbito léxico es el alcance que se define en el tiempo de lexing. En otras palabras, el ámbito léxico se basa en donde son escritas por usted las variables y los bloques de ámbito, en tiempo de escritura, y por lo tanto es (en su mayoría) establecidos invariablemente en el momento en que el lexer procesa su código.

Nota: Vamos a ver en un poco que hay algunas maneras de engañar el ámbito léxico, por lo tanto, podemos modificarlo después de que el lexer ha pasado, pero esto es algo mal visto. Se considera la mejor práctica para tratar el ámbito léxico como, de hecho, sólo léxico, y por lo tanto totalmente invariable en el tiempo por naturaleza.

Consideremos este bloque de código:

```
function foo(a) {

    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2 4 12
```

Hay tres ámbitos anidados inherentes en este ejemplo de código. Puede ser útil pensar en estos ámbitos como burbujas dentro de los demás.



Burbuja 1 abarca el ámbito global, y tiene sólo un identificador en ella: `foo`.

Burbuja 2 abarca el alcance de `foo`, que incluye los tres identificadores: `a`, `bar` y `b`.

Burbuja 3 abarca el alcance de `bar`, e incluye sólo un identificador: `c`.

Las burbujas de alcance se definen por donde se escriben los bloques de alcance, el cual está anidado uno dentro del otro, etc. En el capítulo siguiente, discutiremos diferentes unidades de alcance, pero *por ahora, supongamos que cada función crea una nueva Burbuja de alcance*.

La burbuja para `bar` está totalmente contenida dentro de la burbuja para `foo`, porque (y sólo porque) es donde elegimos definir las funciones de `bar`.

Observe que estas burbujas anidadas están anidadas estrictamente. No estamos hablando de diagramas de Venn donde las burbujas pueden cruzar los límites. En otras palabras, ninguna burbuja para alguna función puede existir simultáneamente (parcialmente) dentro de otras dos burbujas de alcance externo, así como ninguna función puede estar parcialmente dentro de cada una de las dos funciones principales.

Consultas

La estructura y la ubicación relativa de estas burbujas de alcance explica completamente al Motor todos los lugares en los que necesita buscar para encontrar un identificador.

En el fragmento de código anterior, el Motor ejecuta la instrucción `console.log(..)` y busca las tres variables referenciadas `a`, `b` y `c`. Primero comienza con la burbuja más interna del ámbito, el ámbito de la `function bar(..)`. No encontrará una `a` allí, *por lo que sube un*

nivel, a la próxima burbuja de ámbito más cercano, el ámbito de `foo(..)`. Encuentra `a` allí, por lo que usa esa `a`. Lo mismo para `b`. Pero `c`, se encuentra dentro de `bar(..)`.

Si hubiese habido `c` tanto dentro de `bar(..)` como dentro de `foo(...)`, la sentencia `console.log(..)` habría encontrado y usado el de `bar(..)`, sin llegar nunca a la de `foo`

La búsqueda de alcance se detiene una vez que encuentra la primera coincidencia. El mismo nombre de identificador se puede especificar en varias capas de ámbito anidado, lo cual se denomina "sombreado (shadowing)" (el identificador interno "sombrea (shadows)" el identificador externo). Independientemente del sombreado, la búsqueda del ámbito siempre comienza en el ámbito más interior del que se está ejecutando en ese momento, y va hacia el exterior/hacia-arriba hasta que encuentre la primera coincidencia y se detiene.

Nota: Las variables globales también son automáticamente propiedades del objeto global (ventana en los navegadores, etc.), por lo que es posible hacer referencia a una variable global no directamente por su nombre léxico, sino indirectamente como una referencia de propiedad del objeto global.

```
window.a
```

Esta técnica da acceso a una variable global que de otro modo sería inaccesible debido a que se sombreaba. No obstante, no se puede acceder a variables no globales sombreadas.

No importa de dónde se invoque una función, ni siquiera cómo se invoca, su alcance léxico sólo se define por donde se declaró la función.

El proceso de búsqueda de alcance léxico sólo se aplica a identificadores de primera clase, como los `a`, `b` y `c`. Si tuviera una referencia a `foo.bar.baz` en un pedazo de código, la búsqueda de alcance léxico se aplicaría a encontrar el identificador `foo`, pero una vez que localiza esa variable, las reglas de acceso a la propiedad del objeto asumen el control para resolver `bar` y `baz`, respectivamente.

2.2 Trucos léxicos

Si el ámbito léxico está definido sólo por el lugar en el que se declara una función, que es enteramente una decisión de autor-tiempo, ¿cómo podría haber una forma de "modificar" (aka, cheat) el alcance léxico en tiempo de ejecución?

JavaScript tiene dos mecanismos. Ambos son igualmente mal vistos en la comunidad en general como malas prácticas para usar en su código. Pero los argumentos típicos en contra de ellos a menudo les falta el punto más importante: **engañar el alcance léxico conduce a un rendimiento más pobre**.

Antes de explicar el problema de rendimiento, veamos cómo funcionan estos dos mecanismos.

`eval`

La función `eval(..)` en JavaScript toma una cadena como argumento y trata el contenido de la cadena como si realmente hubiera sido el código de autor en ese punto del programa. En otras palabras, puede generar código de forma programática dentro de su código creado y ejecutar el código generado como si hubiera estado allí en tiempo de autor.

Evaluando `eval(..)` (juego de palabras a propósito) en esa luz, debería estar claro cómo `eval(..)` le permite modificar el entorno del alcance léxico para hacer trampa y fingir que el código de autor-tiempo (aka, lexical) estaba allí todo el tiempo .

En las siguientes líneas de código después de que se haya ejecutado un `eval(..)`, el Motor no "sabrá" o "cuidará" que el código anterior en cuestión se haya interpretado dinámicamente y, por lo tanto, haya modificado el entorno del ámbito léxico. El Motor simplemente realizará sus análisis de alcance léxico como siempre lo hace.

Considere el siguiente código:

```
function foo(str, a) {
    eval( str ); // cheating!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1 3
```

La cadena " `var b = 3;` " es tratada en el punto de la llamada `eval(..)` , como código que estuvo allí todo el tiempo. Debido a que el código pasa a declarar una nueva variable `b` , modifica el ámbito léxico existente de `foo(..)` . De hecho, como se mencionó anteriormente, este código realmente crea la variable `b` dentro de `foo(..)` que sombrea la `b` que se declaró en el ámbito externo (global).

Cuando se produce la llamada `console.log(..)` , encuentra tanto `a` como `b` en el ámbito de `foo(..)` , y nunca encuentra la `b` externa. Así, imprimimos " `1 3` " en vez de " `1 2` " como habría sido normalmente el caso.

Nota: En este ejemplo, por simplicidad, la cadena de "código" que pasamos fue un literal fijo. Pero podría haber sido creada mediante programación agregando caracteres juntos en función de la lógica de su programa. `eval(..)` suele utilizarse para ejecutar código creado dinámicamente, ya que evaluar dinámicamente un código esencialmente estático a partir de una cadena literal no proporcionaría ningún beneficio real a la creación del código directamente.

De forma predeterminada, si una cadena de código que `eval(..)` ejecuta contiene una o más declaraciones (variables o funciones), esta acción modifica el ámbito léxico existente en el que reside el `eval(..)` . Técnicamente, `eval(..)` puede ser invocado "indirectamente", a través de varios trucos (que va más allá de nuestra discusión aquí), lo que hace que en lugar de ejecutar en el contexto del ámbito global, por lo tanto, modificarlo. Pero en cualquier caso, `eval(..)` puede en tiempo de ejecución modificar un ámbito léxico de autor-tiempo.

Nota: `eval(..)` cuando se utiliza en un programa en modo estricto opera en su propio ámbito léxico, lo que significa que las declaraciones hechas dentro del `eval()` no modifican el ámbito de inclusión.

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

Hay otras instalaciones en JavaScript que equivalen a un efecto muy similar al `eval(..`). `setTimeout(..)` y `setInterval(..)` pueden tomar una cadena para su primer argumento respectivo, cuyo contenido se evalúa como el código de una función generada dinámicamente. Se trata de un comportamiento antiguo, heredado y desactualizado desde hace mucho tiempo. ¡No lo hagas!

El nuevo constructor de función `function(...)` toma de forma similar una cadena de código en su último argumento para convertirse en una función generada dinámicamente (el primer argumento, si lo hay, son los parámetros nombrados para la nueva función). Esta sintaxis de función-constructor es ligeramente más segura que `eval(..)`, pero debe evitarse en su código.

Los casos de uso para la generación dinámica de código dentro de su programa son increíblemente raros, ya que las degradaciones del rendimiento casi nunca vale la pena la capacidad.

`with`

El otro aspecto con el ceño fruncido (y ahora desaprobado!) en JavaScript que engaña el ámbito léxico es la palabra clave `with`. Existen múltiples vías válidas que pueden explicarse, pero voy a elegir aquí para explicarlo desde la perspectiva de cómo interactúa y afecta el alcance léxico.

`with` se explica típicamente como un atajo para hacer referencias múltiples de la característica contra un objeto sin repetir la referencia del objeto mismo cada vez.

Por ejemplo:

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};

// más "tedioso" repetir "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;

// atajo "mas fácil"
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

Sin embargo, hay mucho más que hacer aquí que sólo un atajo conveniente para el acceso a la propiedad del objeto. Considere:

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2 -- Oops, leaked global!
```

En este ejemplo de código, se crean dos objetos `o1` y `o2`. Uno tiene una propiedad y el otro no. La función `foo(..)` toma una referencia de objeto `obj` como un argumento, y se llama `with (obj) {..}` en la referencia. Dentro del bloque `with`, hacemos lo que parece ser una referencia léxica normal a una variable `a`, una referencia LHS de hecho (ver Capítulo 1), para asignarle el valor de `2`.

Cuando pasamos a `o1`, la asignación `a = 2` encuentra la propiedad `o1.a` y le asigna el valor `2`, como se refleja en la siguiente sentencia `console.log(o1.a)`. Sin embargo, cuando pasamos en `o2`, ya que no tiene una propiedad `a`, no se crea tal propiedad, y `o2.a` permanece indefinido.

Pero luego observamos un efecto secundario peculiar, el hecho de que una variable global `a` fue creada por la asignación `a = 2`. ¿Cómo puede ser esto posible?

La instrucción `with` toma un objeto, uno que tiene cero o más propiedades, **y trata ese objeto como si fuera un ámbito léxico completamente separado**, por lo tanto las propiedades del objeto son tratadas como identificadores lexicalmente definidos en ese "ámbito".

Nota: A pesar de que un bloque `with` trata un objeto como un ámbito léxico, una declaración `var` normal dentro de ese con el bloque no será delimitada con el bloque, sino el ámbito de la función que contiene.

Mientras que la función `eval(...)` puede modificar el alcance léxico existente si toma una cadena de código con una o más declaraciones en ella, la instrucción `with` **crea realmente un ámbito léxico completamente nuevo** desde el aire, desde el objeto al que se pasa .

Entendido de esta manera, el "ámbito" declarado por la instrucción `with` cuando pasamos en `o1` era `o1` , y "scope" tenía un "identificador" en el que corresponde a la propiedad `o1.a` . Pero cuando usamos `o2` como el "ámbito", no tenía tal "identificador" en él, por lo tanto las reglas normales de identificación de consulta LHS (véase el capítulo 1) se produce.

Ni el "alcance" de `o2` , ni el alcance de `foo(..)` , ni el alcance global incluso, tiene un identificador `a` para ser encontrado, por lo que cuando se ejecuta `a = 2` , resulta en la creación automática-global que se crea (Ya que estamos en modo no estricto).

Es un tipo extraño de pensamiento ver `with` cambiar, en tiempo de ejecución, un objeto y sus propiedades en un "ámbito" con "identificadores". Pero esa es la explicación más clara que puedo dar para los resultados que vemos.

Nota: Además de ser una mala idea de usar, ambos `eval(..)` y `with` son afectados (restringido) por el modo estricto. `With` es descartada completamente, mientras que varias formas de `eval(...)` indirecto o inseguro son rechazadas mientras que retiene la funcionalidad del core.

Rendimiento

Ambos `eval(..)` y `with` engañan el ámbito léxico definido por autor-tiempo modificando o creando un nuevo ámbito léxico en tiempo de ejecución.

Entonces, ¿cuál es el problema, usted preguntará? Si ofrecen funcionalidad más sofisticada y flexibilidad de codificación, ¿no son estas buenas características? No.

El Motor de JavaScript tiene una serie de optimizaciones de rendimiento que realiza durante la fase de compilación. Algunos de estos se reducen a ser capaz de analizar esencialmente estáticamente el código como lexes, y pre-determinar dónde están todas las declaraciones de variables y funciones, por lo que se necesita menos esfuerzo para resolver los identificadores durante la ejecución.

Pero si el Motor encuentra un `eval(...)` o un `with` en el código, esencialmente tiene que asumir que toda su conciencia de la localización del identificador puede ser inválida, porque no puede saber a la hora exacta en que el código puede pasar a `eval(..)` para modificar el ámbito léxico o el contenido del objeto con el que se puede pasar para crear un nuevo ámbito léxico a consultar.

En otras palabras, en el sentido pesimista, la mayoría de las optimizaciones que haría son inútiles si `eval(..)` o `with` están presentes, por lo que simplemente no realiza las optimizaciones en absoluto.

Su código casi siempre tiende a correr más lento simplemente por el hecho de que incluye un `eval(..)` o un `with` en cualquier parte del código. No importa lo inteligente que sea el Motor para tratar de limitar los efectos secundarios de estos supuestos pesimistas, **no hay manera de evitar que, sin las optimizaciones, el código funcione más lentamente**.

2.3 Revisión (TL; DR)

El alcance léxico significa que el alcance se define por las decisiones de tiempo de autor de donde se declaran las funciones. La fase de lexing de la compilación es esencialmente capaz de saber dónde y cómo se declaran todos los identificadores y así predecir cómo se buscarán durante la ejecución.

Dos mecanismos en JavaScript pueden "engañar" el ámbito léxico: `eval(..)` y `with`. El primero puede modificar el alcance léxico existente (en tiempo de ejecución) evaluando una cadena de "código" que tiene una o más declaraciones en él. Mientras que el último esencialmente crea un nuevo ámbito léxico (de nuevo, en tiempo de ejecución) tratando una referencia de objeto como un "ámbito" y las propiedades de ese objeto como identificadores de ámbito.

La desventaja de estos mecanismos es que derrota la capacidad del Motor para realizar optimizaciones en tiempo de compilación en cuanto a la búsqueda de scope, porque el Motor tiene que asumir pesimistamente que tales optimizaciones serán inválidas. El código se ejecutará más lentamente como resultado de usar cualquiera de las funciones. **No las use**.

3- Function vs. Block Scope

Como se exploró en el capítulo 2, el alcance consiste en una serie de "burbujas" y cada uno actúa como un contenedor o cubo, en el que se declaran *identificadores (variables, funciones)*. Estas burbujas se anidan perfectamente dentro de sí, y esta anidación se define a la hora del autor.

Pero, ¿qué hace exactamente una nueva burbuja? ¿Es sólo la función? ¿Pueden otras estructuras en JavaScript crear burbujas de alcance?

3.1 Ámbito de las funciones

La respuesta más común a esas preguntas es que JavaScript tiene un alcance/ámbito/scope basado en funciones. Es decir, cada función que declara crea una nueva burbuja para sí misma, pero ninguna otra estructura crea sus propias burbujas de alcance. Como veremos más adelante, esto no es del todo cierto.

Pero primero, vamos a explorar el alcance de la función y sus implicaciones.

Considere este código:

```
function foo(a) {
    var b = 2;

    // some code

    function bar() {
        // ...
    }

    // more code

    var c = 3;
}
```

En este fragmento, la burbuja de alcance para `foo(..)` incluye los identificadores `a`, `b`, `c` y `bar`. No importa dónde en el ámbito de una declaración aparezca, la variable o función pertenece a la burbuja que contiene el ámbito, independientemente. Vamos a explorar cómo funciona eso exactamente en el próximo capítulo.

`bar(..)` tiene su propia burbuja de alcance. Lo mismo ocurre con el ámbito global, que tiene sólo un identificador adjunto a él: `foo`.

Debido a que `a`, `b`, `c`, y `bar` pertenecen todos a la burbuja de alcance de `foo(..)`, no son accesibles fuera de `foo(..)`. Es decir, el código siguiente resultaría en errores `ReferenceError`, ya que los identificadores no están disponibles para el ámbito global:

```
bar(); // fails

console.log( a, b, c ); // all 3 fail
```

Sin embargo, todos estos identificadores (`a` , `b` , `c` , `foo` y `bar`) son accesibles dentro de `foo(..)` , y de hecho también disponible dentro de `bar(..)` (suponiendo que no hay declaraciones de identificador de sombra dentro de `bar(..)`).

El ámbito de la función fomenta la idea de que todas las variables pertenecen a la función y pueden utilizarse y reutilizarse a lo largo de toda la función (incluso accesibles a los ámbitos anidados). Este enfoque de diseño puede ser muy útil, y ciertamente puede hacer pleno uso de la naturaleza "dinámica" de las variables JavaScript para asumir valores de diferentes tipos según sea necesario.

Por otro lado, si no se toman precauciones cuidadosas, las variables existentes a través de la totalidad de un ámbito puede conducir a algunas trampas inesperadas.

3.2 Ocultación en el ámbito común

La forma tradicional de pensar acerca de las funciones es que declares una función y luego agregas código dentro de ella. Pero el pensamiento inverso es igualmente poderoso y útil: toma cualquier sección arbitraria de código que hayas escrito, y envuelve una declaración de función alrededor de ella, que en efecto "oculta" el código.

El resultado práctico es crear una burbuja de alcance alrededor del código en cuestión, lo que significa que cualquier declaración (variable o función) en ese código ahora estará vinculada al ámbito de la nueva función de empaquetado, en lugar del ámbito de inclusión anterior. En otras palabras, puede "ocultar" variables y funciones encerrándolas en el ámbito de una función.

¿Por qué las variables y funciones "ocultas" serían una técnica útil?

Hay una variedad de razones que motivan este ocultamiento basado en el alcance. Ellos tienden a surgir a partir del principio de diseño de software "Principio de Menor Privilegio", también llamado a veces "Menos Autoridad" o "Menos Exposición". Este principio establece que en el diseño de software, como la API para un módulo/objeto, debe exponer sólo lo que es mínimamente necesario y "ocultar" todo lo demás.

Este principio se extiende a la elección de qué ámbito debe contener variables y funciones. Si todas las variables y funciones estuvieran en el ámbito global, obviamente serían accesibles para cualquier ámbito anidado. Pero esto violaría el principio de "Menos ..." en el que usted está (probablemente) exponiendo muchas variables o funciones que de otra manera debería mantener privadas, ya que el uso apropiado del código desalentaría el acceso a esas variables/funciones.

Por ejemplo:

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

En este fragmento, la variable `b` y la función `doSomethingElse(..)` son probablemente detalles "privados" de cómo `doSomething(..)` hace su trabajo. Dando el ámbito de acceso "acceso" a `b` y `doSomethingElse(..)` no sólo es innecesario, sino también posiblemente "peligroso", ya que puede ser utilizado de manera inesperada, intencionalmente o no, y esto puede violar las suposiciones de pre-condición de `doSomething(..)` .

Un diseño más "apropiado" ocultaría estos detalles privados dentro del alcance de `doSomething(...)` , tales como:

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

Ahora, `b` y `doSomethingElse(..)` no son accesibles a ninguna influencia exterior, como cuando estaban controladas sólo por `doSomething(..)` . La funcionalidad y el resultado final no se han visto afectados, pero el diseño mantiene los detalles privados, lo que suele considerarse un mejor software.

Evitar colisiones

Otro beneficio de "ocultar" variables y funciones dentro de un ámbito es evitar la colisión no deseada entre dos identificadores diferentes con el mismo nombre pero con diferentes usos. Los resultados de la colisión a menudo producen una sobrescritura inesperada de los valores.

Por ejemplo:

```
function foo() {
    function bar(a) {
        i = 3; // Cambiar el `i` en el ámbito de inclusión for-loop
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // Oops, bucle infinito!
    }
}

foo();
```

La asignación `i = 3` dentro de `bar(..)` sobrescribe, inesperadamente, el `i` que se declaró en `foo(..)` en el bucle `for`. En este caso, resultará en un bucle infinito, porque `i` se establece en un valor fijo de `3` y que permanecerá para siempre `<10`.

La asignación dentro de la `bar(..)` necesita declarar una variable local para usar, independientemente del nombre de identificador elegido. `var i = 3;` solucionaría el problema (y crearía la declaración de "variable sombreada" anteriormente mencionada para `i`). Una opción adicional, no alternativa, es escoger otro nombre de identificador, tal como `var j = 3` ;. Sin embargo, su diseño de software, naturalmente, puede llamar el mismo nombre de identificador, por lo que la utilización de alcance para "ocultar" su declaración interna es la mejor/única opción en ese caso.

Global "Namespaces"

Un ejemplo particularmente fuerte de colisión variable (probable) ocurre en el ámbito global. Las bibliotecas múltiples cargadas en su programa pueden chocar muy fácilmente entre sí si no ocultan correctamente sus funciones y variables internas/privadas.

Dichas bibliotecas normalmente crearán una declaración de variable única, a menudo un objeto, con un nombre suficientemente único, en el ámbito global. Este objeto se utiliza a continuación como un "espacio de nombres" para esa biblioteca, donde todas las exposiciones específicas de la funcionalidad se hacen como propiedades fuera de ese objeto (espacio de nombres), en lugar de identificadores de nivel superior de ámbito léxico por sí mismos.

Por ejemplo:

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
    }
};
```

Gestión de módulos

Otra opción para evitar colisiones es el más moderno enfoque de "módulo", utilizando cualquiera de los distintos administradores de dependencia. Utilizando estas herramientas, ninguna biblioteca agrega identificadores al ámbito global, sino que se requiere que su identificador(es) sean explícitamente importados a otro ámbito específico mediante el uso de los diversos mecanismos del administrador de dependencias.

Debe observarse que estas herramientas no poseen funcionalidad "mágica" que está exenta de reglas de alcance léxico. Simplemente usan las reglas de alcance como se explica aquí para hacer cumplir que no se inyectan identificadores en ningún ámbito compartido y, en cambio, se mantienen en ámbitos privados no susceptibles a colisiones, lo que evita colisiones de alcance accidental.

Como tal, puede su código defensivamente, lograr los mismos resultados que los hacen los administradores de dependencias sin necesidad de utilizarlos, si así lo desea. Consulte el Capítulo 5 para obtener más información sobre el patrón del módulo.

3.3 Funciones como ámbitos

Hemos visto que podemos tomar cualquier fragmento de código y envolver una función alrededor de él, y que efectivamente "oculta" cualquier declaración de función o variable encerrada desde el ámbito externo dentro del ámbito interno de esa función.

Por ejemplo:

```
var a = 2;

function foo() { // <-- insert this

    var a = 3;
    console.log( a ); // 3

} // <-- and this
foo(); // <-- and this

console.log( a ); // 2
```

Si bien esta técnica "funciona", no es necesariamente ideal. Hay algunos problemas que introduce. El primero es que tenemos que declarar una función nombrada `foo()`, lo que significa que el nombre del identificador `foo` mismo "contamina" el ámbito de inclusión (global, en este caso). También tenemos que llamar explícitamente a la función por nombre (`foo()`) para que el código envuelto en realidad se ejecute.

Sería más ideal si la función no necesitaba un nombre (o, más bien, el nombre no contamine el ámbito de inclusión) y si la función podría ejecutarse automáticamente.

Afortunadamente, JavaScript ofrece una solución a ambos problemas.

```
var a = 2;

(function foo(){ // <-- insert this

    var a = 3;
    console.log( a ); // 3

})(); // <-- and this

console.log( a ); // 2
```

Vamos a desglosar lo que está sucediendo aquí.

En primer lugar, observe que la sentencia de la función wrapping comienza con `(function...` (con paréntesis al inicio) en lugar de simplemente `function...`. Si bien esto puede parecer un detalle menor, en realidad es un cambio importante. En lugar de tratar la función como una declaración estándar, la función se trata como una función-expresión.

Nota: La forma más fácil de distinguir entre declaración y expresión es la posición de la palabra " `function` " en la sentencia (no sólo una línea, sino una declaración distinta). Si " `function` " es la primera parte de la declaración, entonces es una declaración de función. De lo contrario, es una expresión de función.

La diferencia clave que podemos observar aquí entre una declaración de función y una expresión de función se refiere a donde su nombre está enlazado como un identificador.

Compare los dos fragmentos anteriores. En el primer fragmento, el nombre `foo` está enlazado en el ámbito de inclusión, y lo llamamos directamente con `foo()`. En el segundo snippet, el nombre `foo` no está enlazado en el ámbito de inclusión, sino que está limitado sólo dentro de su propia función.

En otras palabras, `(function foo() {..})` como una expresión significa que el identificador `foo` se encuentra sólo en el ámbito donde se indican los tres puntos `...`, no en el ámbito externo. Ocultar el nombre `foo` dentro de sí significa que no contamina el ámbito de inclusión innecesariamente.

Anónimo vs. Nombrado

Probablemente esté más familiarizado con las expresiones de función como parámetros de devolución de llamada, como:

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

Esto se denomina "expresión de función anónima", porque `function()...` no tiene identificador de nombre en ella.

Las expresiones de función pueden ser anónimas, pero las declaraciones de funciones no pueden omitir el nombre, que sería una gramática ilegal en JS.

Las expresiones de función anónimas son rápidas y fáciles de escribir, y muchas bibliotecas y herramientas tienden a fomentar este estilo de código idiomático.

Sin embargo, tienen varios inconvenientes a considerar:

1. Las funciones anónimas no tienen ningún nombre útil para mostrar en lo stack trace

(errores), lo que puede dificultar la depuración.
2. Sin un nombre, si la función necesita referirse a sí misma, para la recursión, etc., la referencia **deprecated** `arguments.callee` es desafortunadamente requerida. Otro ejemplo de la necesidad de autorreferencia es cuando una función de controlador de eventos desea desvincularse después de que se activa.
3. Las funciones anónimas omiten un nombre que es a menudo útil en proporcionar código más legible/comprensible. Un nombre descriptivo ayuda a auto-documentar el código en cuestión.

Las expresiones de la función en línea son poderosas y útiles - la cuestión de anónimo vs nombre no quita eso. Proporcionar un nombre para su expresión de función con bastante efectividad aborda todos estos draw-backs, pero no tiene desventajas tangibles. **La mejor práctica es nombrar siempre las expresiones de su función**:

```
setTimeout( function timeoutHandler(){ // <-- Look, I have a name!
    console.log( "I waited 1 second!" );
}, 1000 );
```

Invocando expresiones de función inmediatamente

```
var a = 2;

(function foo(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

Ahora que tenemos una función como una expresión en virtud de envolverla en un par `( )`, podemos ejecutar esa función añadiendo otro `( )` al final, como `(función foo() {..}) ()`. El primer par enclosing `( )` hace que la función sea una expresión, y la segunda `( )` que ejecute la función.

Este patrón es tan común, hace unos años la comunidad acordó un término para ello: `IIFE`, que significa expresión de función inmediatamente invocada.

Por supuesto, los `IIFE` no necesitan nombres, necesariamente - la forma más común de IIFE es usar una expresión de función anónima. Aunque ciertamente menos común, nombrar un IIFE tiene todos los beneficios antes mencionados sobre expresiones de función anónimas, por lo que es una buena práctica por adoptar.

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

Hay una ligera variación en el formulario tradicional IIFE, que algunos prefieren: `(function() {..}())`. Mira con atención para ver la diferencia. En la primera forma, la expresión de la función se envuelve en `( ),` y entonces se invoca con el par `( )` que está en el exterior justo después de él. En la segunda forma, el par que invoca `( )` se mueve al interior del par envolvente exterior `( )`.

Estas dos formas son idénticas en funcionalidad. Es puramente una elección estilística que usted prefiera.

Otra variación en IIFE que es bastante común es usar el hecho de que son, de hecho, sólo llamadas de función, y pasarlas en argumento(s).

Por ejemplo:

```
var a = 2;

(function IIFE( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

Pasamos en la referencia de objeto `window`, pero nombramos el parámetro `global`, de modo que tengamos una delineación estilística clara para referencias globales vs. no globales. Por supuesto, puede pasar cualquier cosa desde un ámbito incluido que desee, y puede nombrar el parámetro (s) cualquier cosa que se adapte a usted. Esto es en su mayoría sólo elección estilística.

Otra aplicación de este patrón aborda los concerns (nicho menor) de que el identificador indefinido por defecto podría tener su valor incorrectamente sobrescrito, causando resultados inesperados. Al nombrar un parámetro indefinido, pero sin pasar ningún valor

para ese argumento, podemos garantizar que el identificador indefinido es de hecho el valor indefinido en un bloque de código:

```
undefined = true; // El establecimiento de una mina de tierra para el otro código! ¡ev
itarlo!

(function IIFE( undefined ){

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

Otra variación del IIFE invierte el orden de las cosas, donde la función a ejecutar se da en segundo lugar, después de la invocación y los parámetros para pasarle. Este patrón se utiliza en el proyecto UMD (Universal Module Definition). Algunas personas lo encuentran un poco más limpio de entender, aunque es un poco más detallado.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

La expresión de la función `def` se define en la segunda mitad del fragmento y se pasa como parámetro (también llamado `def`) a la función IIFE definida en la primera mitad del fragmento. Finalmente, se invoca el parámetro `def` (la función), pasando `window` como parámetro global.

3.4 Bloques como ámbitos

Si bien las funciones son la unidad de ámbito más común, y ciertamente la más amplia de los enfoques de diseño en la mayoría de los códigos JS en circulación, otras unidades de alcance son posibles, y el uso de estas otras unidades de alcance puede conducir aún mejor, más limpio y código mantenible.

Muchos lenguajes distintos de JavaScript soportan Block Scope, por lo que los desarrolladores de esos lenguajes están acostumbrados a esa mentalidad, mientras que aquellos que sólo han trabajado principalmente en JavaScript pueden encontrar el concepto un poco extraño.

Pero incluso si usted nunca ha escrito una sola línea de código en forma de bloque de ámbito, usted todavía está probablemente familiarizado con este lenguaje muy común en JavaScript:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

Declaramos la variable `i` directamente dentro de la cabeza del bucle `for`, lo más probable es que nuestra intención sea usar `i` sólo dentro del contexto de ese bucle `for`, y esencialmente ignorar el hecho de que la variable realmente se alcanza al ámbito de inclusión (función o global).

Eso es a lo que se refiere el alcance de bloque. Declarar las variables lo más cerca posible, lo más local posible, a donde se van a utilizar. Otro ejemplo:

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

Estamos usando una variable `bar` sólo en el contexto de la sentencia `if`, por lo que hace sentido que lo declaremos dentro del `if-block`. Sin embargo, cuando declaramos variables no es relevante cuando se utiliza `var`, porque siempre pertenecerán al ámbito de inclusión. Este fragmento es esencialmente "falso" de bloque de alcance, por razones estilísticas, y confiar en la auto-ejecución no accidentalmente utilizar `bar` en otro lugar en ese ámbito.

El alcance del bloque es una herramienta para extender el "Principio de Exposición al Menor Privilegio" anterior desde ocultar información en funciones a ocultar información en bloques de nuestro código.

Considere nuevamente el ejemplo for-loop:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

¿Por qué contaminar todo el ámbito de una función con la variable `i` que sólo va a ser (o sólo debería ser, al menos) utilizada para el bucle `for` ?

Pero lo que es más importante, los desarrolladores pueden preferir revisarse contra el uso (re) accidental de variables ajenas a su propósito, como ser emitido un error sobre una variable desconocida si intenta utilizarlo en el lugar equivocado. Bloquear el ámbito (si fuera posible) para la variable `i` haría `i` disponible sólo para el bucle `for` , causando un error si se accede a otro lugar en la función. Esto ayuda a asegurar que las variables no se vuelvan a utilizar en formas confusas o difíciles de mantener.

Pero, la triste realidad es que, en la superficie, JavaScript no tiene ninguna facilidad para el alcance del bloque.

Es decir, hasta que hagas un poco más.

`with`

Hemos aprendido acerca de `with` en el Capítulo 2. Si bien es un fruncido a construir, es un ejemplo de (una forma de) ámbito de bloque, en el que el ámbito que se crea desde el objeto sólo existe para la vida de `with` con la declaración, y no en el ámbito circundante.

`try/catch`

Es un hecho muy poco conocido que JavaScript en ES3 especificó la declaración de variable en la cláusula `catch` de un `try/catch` para ser bloqueado al bloque `catch` .

Por ejemplo:

```
try {
    undefined(); // illegal operation to force an exception!
}
catch (err) {
    console.log( err ); // works!
}

console.log( err ); // ReferenceError: `err` not found
```

Como puede ver, `err` existe sólo en la cláusula `catch` y lanza un error cuando intenta hacer referencia a él en otro lugar.

Nota: Si bien este comportamiento se ha especificado y es cierto de prácticamente todos los entornos JS estándar (excepto quizás en un IE antiguo), muchos linters parecen todavía quejarse si tiene dos o más cláusulas de `catch` en el mismo ámbito que cada declaración de su variable de error con el mismo nombre del identificador. Esto no es en realidad una nueva definición, ya que las variables son de bloque con seguridad de alcance, pero los linters todavía parecen molestarse, y se quejan de este hecho.

Para evitar estas advertencias innecesarias, algunos devs nombrarán sus variables de `catch` `err1` , `err2` , etc. Otros devs simplemente desactivarán la comprobación de linting para nombres de variables duplicados.

La naturaleza de alcance de `catch` puede parecer un hecho académico inútil, pero vea el Apéndice B para más información sobre lo útil que podría ser.

`let`

Hasta ahora, hemos visto que JavaScript sólo tiene algunos comportamientos de nicho extraño que exponen la funcionalidad de ámbito de bloque. Si eso fuera todo lo que tenemos, y de hecho tuvimos por muchos, muchos años, entonces el alcance de bloque no sería útil para el desarrollador de JavaScript.

Afortunadamente, ES6 cambia eso, e introduce una nueva palabra clave `let` que se pone junto a `var` como otra forma de declarar variables.

La palabra clave `let` agrega la declaración de la variable al ámbito de cualquier bloque (comúnmente un par `{..}`) en el que está contenido. En otras palabras, deje implícitamente el alcance de cualquier bloque para su declaración de variable.

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

El uso de `let` para adjuntar una variable a un bloque existente es algo implícito. Puede confundirte si no estás prestando mucha atención a los bloques que tienen variables de alcance para ellos, y tienen el hábito de mover bloques alrededor, envolverlos en otros bloques, etc, a medida que desarrolla y evoluciona código.

La creación de bloques explícitos para el alcance de bloque puede solucionar algunas de estas preocupaciones, haciéndolo más evidente de dónde se adjuntan las variables y dónde no. Normalmente, el código explícito es preferible sobre código implícito o sutil. Este estilo de ámbito de bloque explícito es fácil de lograr y se adapta más naturalmente a cómo funciona el alcance de bloque en otros idiomas:

```
var foo = true;

if (foo) {
    { // <-- explicit block
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

Podemos crear un bloque arbitrario al que debemos enlazar simplemente incluyendo un par `{..}` dondequiera que una sentencia sea gramática válida. En este caso, hemos hecho un bloque explícito dentro de la instrucción `if`, que puede ser más fácil como un bloque entero para moverse más adelante en la refactorización, sin afectar la posición y la semántica de la instrucción `if` que encierra.

Nota: Para otra forma de expresar los ámbitos explícitos de bloque, consulte el Apéndice B.

En el capítulo 4, abordaremos el hoisting, que habla de que las declaraciones se toman como existentes para todo el ámbito en el que se producen.

Sin embargo, las declaraciones hechas con `let` no se "hoistean" a todo el alcance del bloque en el que aparecen. Tales declaraciones no "existirán" observables en el bloque hasta la declaración.

```
{
   console.log( bar ); // ReferenceError!
   let bar = 2;
}
```

Garbage Collection

Otra razón de porque bloquear el alcance es útil se refiere a los closures y la Garbage Collection (recolección de basura) para recuperar la memoria. Lo vamos a ilustrar brevemente aquí, pero el mecanismo de closure se explica en detalle en el capítulo 5.

Considere:

```
function process(data) {
    // do something interesting
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

La función `click` de devolución de llamada no necesita la variable `someReallyBigData` en absoluto. Eso significa, teóricamente, después de que el proceso `(..)` se ejecute, la estructura de datos de memoria grande podría ser Garbage Collection. Sin embargo, es bastante probable (aunque depende de la implementación) que el motor JS todavía tenga que mantener la estructura alrededor, ya que la función `click` tiene un closure en todo el ámbito.

Bloquear el alcance puede abordar esta preocupación, por lo que es más claro para el motor que no es necesario mantener `someReallyBigData` alrededor:

```
function process(data) {
    // do something interesting
}

// anything declared inside this block can go away after!
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

Declarar bloques explícitos para que las variables se enlacen localmente es una poderosa herramienta que puede agregar a su caja de herramientas de código.

Bucles `let`

Un caso particular donde `let` brilla está en el caso del ciclo `for` como ha sido discutido previamente.

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

No sólo deja en el encabezado del for-loop unido el `i` al cuerpo del for-loop, sino que de hecho, vuelve a enlazarlo a cada iteración del loop, asegurándose de volver a asignar el valor desde el final de la iteración del bucle anterior.

He aquí otra manera de ilustrar el comportamiento de vinculación por iteración que se produce:

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // re-bound for each iteration!
        console.log( i );
    }
}
```

La razón por la cual esta vinculación por iteración es interesante será aclarada en el Capítulo 5 cuando discutamos closures.

Debido a que las declaraciones se unen a bloques arbitrarios en lugar de al ámbito de la función adjunta (o global), puede haber esquemas en los que el código existente tiene una dependencia oculta en las declaraciones de `var` de la función, y reemplazar la `var` por `let` puede requerir atención adicional al refactorizar el código.

Considere:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

Este código es fácil re-factorizarlo como:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

Pero, tenga cuidado con estos cambios cuando se usan variables de alcance de bloque:

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- don't forget `bar` when moving!
        console.log( baz );
    }
}
```

Vea el Apéndice B para un estilo alterno (más explícito) de alcance de bloque que puede proporcionar un código más fácil de mantener/refactorizar que es más robusto a estos escenarios.

`const`

Además de `let`, ES6 introduce `const`, que también crea una variable de ámbito de bloque, pero cuyo valor es fijo (constante). Cualquier intento de cambiar ese valor en un momento posterior da como resultado un error.

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // block-scoped to the containing `if`

    a = 3; // just fine!
    b = 4; // error!
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

3.5 Revisión (TL; DR)

Las funciones son la unidad de scope más común en JavaScript. Las variables y las funciones que se declaran dentro de otra función están esencialmente "ocultas" de cualquiera de los "ámbitos" incluidos, que es un principio de diseño intencional de un buen software.

Pero las funciones no son de ninguna manera la única unidad de ámbito. Block-scope se refiere a la idea de que las variables y las funciones pueden pertenecer a un bloque arbitrario (generalmente, cualquier par `{..}`) de código, en lugar de sólo a la función de inclusión.

Comenzando con ES3, la estructura `try/catch` tiene block-scope en la cláusula `catch` .

En ES6, se introduce la palabra clave `let` (un primo de la palabra clave `var`) para permitir declaraciones de variables en cualquier bloque arbitrario de código. `if(..) {let a = 2; }` Declarará una variable `a` que esencialmente perteneciente a el alcance del bloque `{..}` de `if` y se adjunta allí.

Aunque algunos parecen creerlo así, el alcance del bloque no debe ser tomado como un reemplazo directo del alcance de la función `var` . Ambas funcionalidades coexisten, y los desarrolladores pueden y deben utilizar tanto las técnicas de alcance de función como de ámbito de bloque, donde son apropiadas para producir un código mejor, más legible y más fácil de mantener.

4- Hoisting

Por ahora, debe estar bastante cómodo con la idea de los scope, y cómo las variables se adjuntan a diferentes niveles de ámbito dependiendo de dónde y cómo se declaran. Tanto el ámbito de la función como el de bloque se comportan por las mismas reglas a este respecto: cualquier variable declarada dentro de un ámbito se adjunta a ese ámbito.

Pero hay un detalle sutil de cómo funciona la inclusión de un ámbito con declaraciones que aparecen en varios lugares dentro de un ámbito, y ese detalle es lo que examinaremos aquí.

4.1 ¿El Huevo o la Gallina?

Hay una tentación de pensar que todo el código que se ve en un programa JavaScript se interpreta línea por línea, de arriba hacia abajo en orden, tal y como se ejecuta el programa. Si bien eso es sustancialmente cierto, hay una parte de esa suposición que puede conducir a un pensamiento incorrecto sobre su programa.

Considere este código:

```
a = 2;

var a;

console.log( a );
```

¿Qué espera imprimir en la sentencia `console.log(..)` ?

Muchos desarrolladores esperan `undefined` , ya que la instrucción `var a` viene después de `a = 2` , y parece natural asumir que la variable es re-definida, y por lo tanto asignado el `undefined` . Sin embargo, la salida será `2` .

Considere otra pieza de código:

```
console.log( a );

var a = 2;
```

Es posible que se sienta tentado a asumir que, dado que el fragmento anterior mostraba un comportamiento de apariencia inferior a superior, tal vez en este fragmento, también se imprimirá `2` . Otros pueden pensar que ya que la variable `a` se utiliza antes de que se declara, esto debe resultar en un `ReferenceError` .

Desafortunadamente, ambas conjeturas son incorrectas. `undefined` es la salida.

Entonces, ¿qué está pasando aquí? Parece que tenemos una pregunta sobre tipo gallina o huevo. ¿Cuál viene primero, la declaración ("huevo"), o la asignación ("gallina")?

4.2 El compilador pega de nuevo

Para responder a esta pregunta, necesitamos referirnos al Capítulo 1, y nuestra discusión sobre los compiladores. Recuerde que el motor realmente compilará su código JavaScript antes de que lo interprete. Parte de la fase de compilación fue encontrar y asociar todas las declaraciones con sus alcances apropiados. El capítulo 2 nos mostró que este es el corazón del alcance léxico.

Por lo tanto, la mejor manera de pensar acerca de las cosas es que todas las declaraciones, tanto las variables como las funciones, se procesan primero, antes de ejecutar cualquier parte de su código.

Cuando ve `var a = 2 ;`, probablemente piensa en eso como una sentencia. Pero JavaScript realmente piensa en ello como dos declaraciones: `var a;` Y `a = 2 ;`. La primera declaración, la declaración, se procesa durante la fase de compilación. La segunda sentencia, la asignación, se deja en su lugar para la fase de ejecución.

Nuestro primer fragmento entonces debe ser pensado como si estuviera siendo manejado igual a esto:

```
var a;
```

```
a = 2;

console.log( a );
```

... donde la primera parte es la compilación y la segunda parte es la ejecución.

Del mismo modo, nuestro segundo fragmento se procesa realmente como:

```
var a;
```

```
console.log( a );

a = 2;
```

Por lo tanto, una forma de pensar, de manera metafórica, sobre este proceso, es que las declaraciones de variables y de funciones se "mueven" desde donde aparecen en el flujo del código hasta la parte superior del código. Esto da lugar al nombre "Hoisting".

En otras palabras, **el huevo (declaración) viene antes de la gallina (asignación).**

Nota: Solamente las declaraciones son subidas (hoisted), mientras que cualquier asignación u otra lógica ejecutable se deja en su lugar. Si la subida fuera a reorganizar la lógica ejecutable de nuestro código, eso podría causar estragos.

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

Se sube la declaración de la función `foo` (que en este caso incluye el valor implícito de ella como una función real), de modo que la llamada en la primera línea pueda ejecutarse.

También es importante tener en cuenta que la subida es por alcance. Así, mientras que nuestros fragmentos anteriores se simplificaron en que sólo incluían alcance global, la función `foo(..)` que ahora estamos examinando muestra que `var a` se sube a la parte superior de `foo(..)` (no, obviamente, a la parte superior del programa). Por lo tanto, el programa puede interpretarse de la siguiente manera:

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

Las declaraciones de funciones se suben, como acabamos de ver. Pero las expresiones funcionales no se suben.

```
foo(); // not ReferenceError, but TypeError!

var foo = function bar() {
    // ...
};
```

El identificador de variable `foo` se sube y se adjunta al ámbito de inclusión (global) de este programa, por lo que f `oo()` no falla como `ReferenceError`. Pero `foo` no tiene ningún valor todavía (como si hubiera sido una declaración de función verdadera en lugar de

expresión). Por tanto, `foo()` intenta invocar el valor indefinido, que es una operación `TypeError` ilegal.

También recuerde que aunque es una expresión de función nombrada, el identificador de nombre no está disponible en el ámbito de inclusión:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

Este fragmento se interpreta con mayor precisión (con hoisting) como:

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

4.3 Funciones Primero

Se sube tanto las declaraciones de funciones como las declaraciones de variables. Pero un detalle sutil (que puede aparecer en el código con varias declaraciones "duplicadas") es que las funciones se suben primero y luego las variables.

Considere:

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

`1` se imprime en lugar de `2` ! Este fragmento es interpretado por el Motor como:

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

Tenga en cuenta que `var foo` era la declaración duplicada (y por lo tanto ignorada), aunque llegó antes de la declaración `function foo() ...` , porque las declaraciones de función son subidas antes que las variables normales.

Mientras las declaraciones `var` multiples/duplicadas se ignoran de forma efectiva, las declaraciones de función posteriores reemplazan a las anteriores.

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

Si bien esto todo puede sonar como nada más que curiosidades académicas interesantes, pone de relieve el hecho de que las definiciones duplicadas en el mismo alcance son una idea realmente mala y, a menudo llevará a resultados confusos.

Las declaraciones de funciones que aparecen dentro de los bloques normales típicamente se elevan al ámbito de inclusión, en lugar de ser condicionales como implica este código:

```
foo(); // "b"

var a = true;
if (a) {
   function foo() { console.log( "a" ); }
}
else {
   function foo() { console.log( "b" ); }
}
```

Sin embargo, es importante tener en cuenta que este comportamiento no es fiable y está sujeto a cambios en futuras versiones de JavaScript, por lo que es mejor evitar declarar funciones en bloques.

4.4 Revisión (TL; DR)

Podemos ser tentados a mirar `var a = 2;` como una declaración, pero el Motor de JavaScript no lo ve así. Ve `var a` y `a = 2` como dos sentencias separadas, la primera es una tarea de fase-compilador y la segunda es una tarea de fase-ejecución.

Lo que esto lleva a que todas las declaraciones en un ámbito, independientemente de dónde aparezcan, se procesan primero antes de ejecutar el propio código. Se puede visualizar esto como declaraciones (variables y funciones) que se "mueven" a la parte superior de sus respectivos ámbitos, que llamamos "hoisting".

Las declaraciones mismas son elevadas, pero las asignaciones, incluso las asignaciones de las expresiones de la función, no se elevan.

Tenga cuidado con las declaraciones duplicadas, especialmente mezcladas entre las declaraciones normales de `var` y las declaraciones de funciones - ¡el peligro lo espera si lo hace!

5- Scope Closure

Llegamos a este punto con, esperanzadamente, una comprensión muy sana y sólida de cómo funciona el scope/ámbito/alcance.

Volvemos nuestra atención a una parte increíblemente importante, pero persistentemente esquiva, casi mitológica, del lenguaje: el closure. Si ha seguido nuestra discusión sobre el alcance léxico hasta el momento, la recompensa es que el closure va a ser, en gran medida, anticlimático, casi auto-obvio. Hay un hombre detrás de la cortina del mago, y estamos a punto de verlo. ¡No, su nombre no es Crockford!

Sin embargo, si usted tiene preguntas persistentes sobre el alcance léxico, ahora sería un buen momento para volver y revisar el capítulo 2 antes de proceder.

5.1 Ilustración

Para aquellos que tienen algo de experiencia en JavaScript, pero quizás nunca han comprendido completamente el concepto de closures, entender el closure puede parecer un nirvana especial que uno debe esforzarse y sacrificarse por alcanzar.

Recuerdo años atrás cuando tenía una comprensión firme de JavaScript, pero no tenía idea de qué era closures. La indirecta de que había ese otro lado del lenguaje, que me prometió aún más capacidad de la que ya poseía, me burlaba y me burlaba. Recuerdo leer a través del código fuente de los primeros frameworks tratando de entender cómo funciona realmente. Recuerdo la primera vez que algo del "patrón de módulo" comenzó a emerger en mi mente. ¡Recuerdo el a-ha! Momentos muy vívidos.

Lo que no sabía en ese entonces, lo que me llevó años entender, y lo que espero poder impartir, es el secreto: el closure está en todas partes de JavaScript, sólo hay que reconocerlo y abrazarlo. Closures no son una herramienta especial de opt-in que usted debe aprender nuevas sintaxis y patrones. No, los closures no son ni siquiera un arma que debes aprender a manejar y dominar como Luke entrenó en La Fuerza.

Los closures ocurren como resultado de escribir código que se basa en el alcance léxico. Simplemente suceden. Usted ni siquiera tiene que intencionalmente crear closures para tomar ventaja de ellos. Los closures se crean y se usan para usted en todo su código. Lo que le falta es el contexto mental apropiado para reconocer, abrazar y apalancar los closures por su propia voluntad.

El momento de la iluminación debe ser: oh, closures ya están apareciendo en todo mi código, finalmente puedo verlos ahora. Entender los closures es como cuando Neo ve la Matrix por primera vez.

5.2 Nitty Gritty

OK, suficiente hipérbole y referencias de películas desvergonzadas.

Aquí está una definición de lo que usted necesita saber para entender y reconocer los closures:

> El closure es cuando una función es capaz de recordar y acceder a su ámbito léxico incluso cuando esa función está ejecutándose fuera de su ámbito léxico.

Vamos a ver algo de código para ilustrar esa definición.

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a ); // 2
    }

    bar();
}

foo();
```

Este código debe parecer familiar a nuestras discusiones de Nested Scope. La funcion `bar()` tiene acceso a la variable `a` en el ámbito externo de inclusión debido a las reglas de búsqueda de alcance léxico (en este caso, es una referencia de referencia RHS).

¿Esto este un "closure"?

Bueno, técnicamente ... tal vez. Pero con nuestra definición de lo que necesitas saber arriba ... no exactamente. Creo que la manera más precisa de explicar la forma en que `bar()` hace referencia a `a` es a través de las reglas de búsqueda de alcance léxico, y esas reglas son sólo (una parte importante!) de lo que es el closure.

Desde una perspectiva puramente académica, lo que se dice del fragmento anterior es que la función `bar()` tiene un closure sobre el alcance de `foo()` (e incluso sobre el resto de los ámbitos a los que tiene acceso, como el alcance global en nuestro caso). Se dice que `bar()` se **cierra (closes)** sobre el alcance de `foo()`. ¿Por qué? porque `bar()` aparece anidado dentro de `foo()`. Llano y simple.

Pero, el closure definido de esta manera no es directamente observable, ni vemos el cierre ejercido en ese fragmento. Vemos claramente el alcance léxico, pero el closure sigue siendo una especie de misteriosa sombra detrás del código.

Consideremos entonces el código que pone el closure a plena luz:

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 -- Whoa, el cierre se acaba de observar, hombre!.
```

La funcion `bar()` tiene acceso de alcance léxico al ámbito interno de `foo()`. Pero entonces tomamos `bar()`, la función en sí, y la pasamos como un valor. En este caso, retornamos el objeto de función propiamente dicho que hace referencia a `bar`.

Después de ejecutar `foo()`, asignamos el valor que retorno (nuestra función interna `bar()`) a una variable llamada `baz`, y luego invocamos `baz()`, que por supuesto está invocando nuestra función interna `bar()`, sólo por una referencia de identificador diferente.

`bar()` se ejecuta, por supuesto. Pero en este caso, se ejecuta fuera de su ámbito léxico declarado.

Después de ejecutado `foo()`, normalmente esperamos que la totalidad del alcance interno de `foo()` se vaya, porque sabemos que el motor emplea un recolector de basura (Garnage Collector) que viene y libera memoria una vez que ya no está en uso. Puesto que parecería que el contenido de `foo()` ya no está en uso, parecería natural que lo consideren como desaparecido.

Pero la "magia" de los closures no deja que esto suceda. Ese alcance interior está, de hecho, todavía "en uso", y por lo tanto no desaparece. **¿Quién lo usa? La funcion `bar()` en sí.**

En virtud de donde fue declarado, `bar()` tiene un closure de alcance léxico sobre ese alcance interno de `foo()`, que mantiene ese alcance vivo para que `bar()` le haga una referencia en cualquier momento posterior.

`bar()` todavía tiene una referencia a ese ámbito, y esa referencia se llama closure.

Por lo tanto, unos pocos microsegundos más tarde, cuando se invoca la variable `baz` (invocando la función interna inicialmente etiquetada `bar`), tiene debidamente acceso al ámbito lexical de autor-tiempo, por lo que puede acceder a la variable tal como

esperábamos.

La función se invoca bien fuera de su alcance léxico de autor-tiempo. El closure permite que la función siga teniendo acceso al ámbito léxico en el que se definió en la hora de autor.

Por supuesto, cualquiera de las varias maneras en que las funciones pueden ser pasadas alrededor como valores, y de hecho invocadas en otros lugares, son todos ejemplos de observación/ejercicio de closures.

```
function foo() {
    var a = 2;

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // Mira ma, vi el closure!
}
```

Pasamos la función interna `baz` a la funcion `bar()`, y llamamos a esa función interna (etiquetada `fn` ahora), y cuando lo hacemos, su closure sobre el alcance interno de `foo()` se observa, accediendo a `a`.

Estas pasadas de funciones también pueden ser indirectas.

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // asigna `baz` a una variable global
}

function bar() {
    fn(); // Mira ma, vi el closure!
}

foo();

bar(); // 2
```

Cualquiera que sea la facilidad que utilicemos para ***transportar*** una función interna fuera de su ámbito léxico, mantendrá una referencia de alcance a donde originalmente fue declarada, y dondequiera que la ejecute, se ejercerá ese closure.

5.3 Ahora puedo ver

Los fragmentos de código anteriores son algo académicos y construidos artificialmente para ilustrar el uso del closures. Pero te prometí algo más que un juguete nuevo. Le prometí que el closure era algo que aparecía constantemente a su alrededor en su código existente. Veamos ahora esa verdad.

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

Tomamos una función interna (llamada `timer`) y la pasamos a `setTimeout(..)`. Pero el temporizador tiene un alcance de closure sobre el alcance de `wait(..)`, de hecho mantiene y utiliza una referencia a la variable `message`.

Mil milisegundos después de que hayamos ejecutado `wait(..)`, y su alcance interior de otra manera se hubiera ido mucho tiempo, ese temporizador de la función interna todavía tiene un closure sobre ese alcance.

En el fondo de las entrañas del Motor, la utilidad incorporada `setTimeout(...)` hace referencia a algún parámetro, probablemente llamado `fn` o `func` o algo así. El motor invoca esa función, que está invocando nuestra función de temporizador interno, y la referencia de alcance léxico está intacta.

Closure.

O, si usted esta persuadido por jQuery (o cualquier framework JS, para el caso):

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

No estoy seguro de qué tipo de código escribes, pero regularmente escribo código que es el responsable de controlar un ejército de drones global lleno de bots de closures, ¡así que esto es totalmente realista!

(Algunos) bromas aparte, esencialmente siempre y dondequiera que traten las funciones (que acceden a sus propios ámbitos léxicos respectivos) como valores de primera clase y los pasan alrededor, es probable que vean las funciones de ejercicio de closures. Sea que los temporizadores, los manejadores de eventos, las solicitudes de Ajax, la mensajería de ventanas cruzadas, los workers de la web, o cualquiera de las otras tareas asíncronas (o síncronas!), Al pasar en una función de devolución de llamada, prepárate para el sling alrededor de los closures!

Nota: El capítulo 3 introdujo el patrón IIFE. Aunque a menudo se dice que el IIFE (solo) es un ejemplo de closure observado, estaría algo en desacuerdo, por nuestra definición anterior.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

Este código "funciona", pero no es estrictamente una observación de como funcionan los closures. ¿Por qué? Debido a que la función (que denominamos "IIFE" aquí) no se ejecuta fuera de su alcance léxico. Todavía se invoca allí mismo en el mismo ámbito que se declaró (el ámbito global que también contiene `a`). `a` se encuentra a través de la búsqueda de alcance léxico normal, no realmente a través del closure.

Si bien el closure podría estar técnicamente ocurriendo a la hora de la declaración, no es estrictamente observable, y así, como dicen, es un árbol que cae en el bosque sin que nadie lo oiga.

Aunque un IIFE no es en sí mismo un ejemplo de closure, crea absolutamente el alcance, y es una de las herramientas más comunes que utilizamos para crear un ámbito que se puede cerrar. Por lo tanto, los IIFEs están fuertemente relacionados con los closures, incluso si no ejercen el closure ellos mismos.

Ponga este libro abajo en este momento, querido lector. Tengo una tarea para ti. Vaya a abrir parte de su código JavaScript reciente. Busque sus funciones como valores e identifique dónde está utilizando el closure y tal vez ni siquiera lo sabía antes.

Esperaré.

¡Ahora lo ves!

5.4 Loops + Closure

El ejemplo canónico más común usado para ilustrar el closure implica el humilde for-loop.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

Nota: Los linters a menudo se quejan al poner funciones dentro de bucles, porque los errores de no entender el closure son muy comunes entre los desarrolladores. Le explicamos cómo hacerlo correctamente aquí, aprovechando todo el poder de los closures. Pero esa sutileza se pierde a menudo en los linters y se quejarán de todos modos, asumiendo que usted no sabe realmente lo que usted está haciendo.

El espíritu de este fragmento de código es que normalmente esperamos que el comportamiento sea que los números " `1` ", " `2` ", .. " `5` " se imprimirían, uno a la vez, uno por segundo, respectivamente.

De hecho, si ejecuta este código, obtiene " `6` " impreso 5 veces, en los intervalos de un segundo.

Huh?

En primer lugar, vamos a explicar de dónde viene `6` . La condición de terminación del bucle es cuando `i` no es `<= 5` . La primera vez que es el caso es cuando `i` es `6` . Así pues, la salida está reflejando el valor final del `i` después de que el bucle termine.

Esto en realidad parece obvio a primera vista. Las devoluciones de llamada de la función `timer` están funcionando bien después de completar el bucle. De hecho, a medida que pasan los temporizadores, aunque se estableciera `SetTimeout(..., 0)` en cada iteración, todas esas devoluciones de función seguirían funcionando estrictamente después de completar el bucle, e imprimir así 6 cada vez.

Pero hay una pregunta más profunda en juego aquí. ¿Qué falta en nuestro código para que se comporte como lo hemos hecho implícito semánticamente?

Lo que falta es que estamos tratando de implicar que cada iteración del bucle "captura" su propia copia de `i` , en el momento de la iteración. Pero, la forma en que funciona el ámbito, todas las 5 de esas funciones, aunque se definen por separado en cada iteración

del bucle, todas están cerradas sobre el mismo ámbito global compartido, que tiene, de hecho, sólo un `i` en él.

Puesto de esa manera, por supuesto todas las funciones comparten una referencia a la misma `i`. Algo sobre la estructura del bucle tiende a confundirnos en pensar que hay algo más sofisticado en el trabajo. No hay. No hay diferencia que si cada uno de los 5 retornos de tiempo de espera se acaba de declarar uno después de la otra, sin bucle en absoluto.

OK, por lo tanto, de nuevo a nuestra pregunta caliente. ¿Qué falta? Necesitamos más alcance de closure. Específicamente, necesitamos un nuevo ámbito de closure para cada iteración del bucle.

En el Capítulo 3 aprendimos que el IIFE crea el ámbito declarando una función y ejecutándola inmediatamente.

Intentemos:

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

¿Eso funciona? Intentalo. Una vez más, voy a esperar.

Terminaré el suspenso por ti. Nope. ¿Pero por qué? Ahora obviamente tenemos más alcance léxico. Cada devolución de llamada de la función de tiempo de espera se está cerrando realmente sobre su propio alcance de iteración creado respectivamente por cada IIFE.

No es suficiente tener un alcance para cerrar si ese ámbito está vacío. Mira de cerca. Nuestro IIFE es sólo un vacío, no hace nada. Necesita algo en él para que sea útil para nosotros.

Necesita su propia variable, con una copia del valor `i` en cada iteración.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

¡Eureka! ¡Funciona!

Una pequeña variación que algunos prefieren es:

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

Por supuesto, puesto que estos IIFEs son sólo funciones, podemos pasar en `i`, y podemos llamarlo `j` si lo preferimos, o incluso podemos llamarlo `i` otra vez. De cualquier manera, el código funciona ahora.

> El uso de un IIFE dentro de cada iteración creó un nuevo alcance para cada iteración, lo que dio a nuestras devoluciones de llamada de función `timer` la oportunidad de cerrar sobre un nuevo ámbito para cada iteración, una que tenía una variable con el valor de iteración adecuado para nosotros acceder.

¡Problema resuelto!

Ámbito de bloque revisado

Mire atentamente nuestro análisis de la solución anterior. Utilizamos un IIFE para crear un nuevo ámbito por iteración. En otras palabras, en realidad necesitábamos un ámbito de bloque por iteración. El capítulo 3 nos mostró la declaración `let`, que secuestra un bloque y declara una variable allí mismo en el bloque.

Esencialmente convierte un bloque en un ámbito que podemos cerrar. Por lo tanto, el siguiente código impresionante "sólo funciona":

```
for (var i=1; i<=5; i++) {
    let j = i; // yay, ámbito de bloque para cerrar (el closure)
    setTimeout( function timer(){
        console.log( j );
    }, j*1000 );
}
```

¡Pero eso no es todo! (En mi mejor voz de Bob Barker). Hay un comportamiento especial definido para las declaraciones utilizadas en la cabeza de un bucle `for`. Este comportamiento indica que la variable se declarará no sólo una vez para el bucle, sino para cada iteración. Y, por último, se inicializará en cada iteración posterior con el valor desde el final de la iteración anterior.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

¿Cuan genial es eso? El ámbito de bloque y el closure trabajando mano a mano, resolviendo todos los problemas del mundo. No sé para de ti, pero eso me convierte en un feliz JavaScripter.

5.5 Módulos

Existen otros patrones de código que aprovechan la potencia de los closures, pero que no parecen estar sobre la superficie de los callbacks. Examinemos el más poderoso de ellos: el módulo.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

Como este código se encuentra en este momento, no hay un closure observable en curso. Simplemente tenemos algunas variables de datos privadas `something` y `another`, y un par de funciones internas `doSomething()` y `doAnother()`, que tienen alcance lexical (y, por tanto, closure!) Sobre el ámbito interno de `foo()`.

Pero ahora considere:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Este es el patrón en JavaScript que llamamos módulo. La forma más común de implementar el patrón de módulo es a menudo llamado "Módulo Revelador (Revealing Module)", y es la variación que presentamos aquí.

Vamos a examinar algunas cosas acerca de este código.

En primer lugar, `CoolModule()` es sólo una función, pero tiene que ser invocado para que haya una instancia de módulo creada. Sin la ejecución de la función externa, la creación del ámbito interno y los closures no se producirían.

En segundo lugar, la función `CoolModule()` devuelve un objeto, denotado por la sintaxis de objeto-literal `{key: value, ...}`. El objeto que devolvemos tiene referencias sobre él a nuestras funciones internas, pero no a nuestras variables de datos internas. Los mantenemos ocultos y privados. Es apropiado pensar en este valor de retorno de objeto como esencialmente una API pública para nuestro módulo.

Este valor de retorno del objeto se asigna en última instancia a la variable externa `foo`, y luego podemos acceder a los métodos de propiedad en la API, como `foo.doSomething()`.

Nota: No se requiere que devolvamos un objeto real (literal) de nuestro módulo. Podríamos regresar una función interna directamente. JQuery es en realidad un buen ejemplo de esto. Los identificadores `jQuery` y `$` son la API pública para el jQuery "module", pero son, ellos mismos, sólo una función (que puede tener propiedades, ya que todas las funciones son objetos).

Las funciones `doSomething()` y `doAnother()` tienen closures sobre el ámbito interno del módulo "instance" (llegado al invocar `CoolModule()`). Cuando transportamos esas funciones fuera del alcance léxico, a través de las referencias de propiedad sobre el objeto que devolvemos, hemos establecido ahora una condición por la cual el closure puede ser observado y ejercido.

Para decirlo más sencillamente, hay dos "requisitos" para que el patrón de módulo sea ejercido:

1. Debe haber una función envolvente externa, y debe invocarse al menos una vez (cada vez crea una instancia de módulo nuevo).
2. La función de inclusión debe devolver al menos una función interna, por lo que esta función interna tiene un closure sobre el ámbito privado, y puede acceder y/o modificar ese estado privado.

Un objeto con una propiedad de función en él solo no es realmente un módulo. Un objeto que es devuelto de una invocación de función que sólo tiene propiedades de datos en él y no hay funciones cerradas no es realmente un módulo, en el sentido observable.

El fragmento de código anterior muestra un creador de módulos autónomo denominado `CoolModule()` que puede invocarse cualquier número de veces, cada vez que se crea una nueva instancia de módulo. Una ligera variación en este patrón es cuando sólo te interesa tener una instancia, un "singleton" de clases:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Aquí, convirtió nuestra función de módulo en un IIFE (véase el capítulo 3), e inmediatamente lo invocó y asignó su valor de retorno directamente a nuestro módulo único identificador de instancia `foo`.

Los módulos son sólo funciones, por lo que pueden recibir parámetros:

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

Otra variación ligera pero potente en el patrón de módulo es nombrar el objeto que estás regresando como tu API pública:

```
var foo = (function CoolModule(id) {
    function change() {
        // modifying the public API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

Al conservar una referencia interna al objeto API público dentro de la instancia del módulo, puede modificar la instancia del módulo desde el interior, incluida la adición y eliminación de métodos, propiedades y el cambio de sus valores.

Módulos modernos

Varios cargadores/administradores de dependencias de módulos terminan esencialmente con este patrón de definición de módulos en una API amigable. En lugar de examinar cualquier biblioteca en particular, permítanme presentar una prueba de concepto muy simple para fines (sólo) de ilustración:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

La parte clave de este código es `modules[name] = impl.apply(impl, deps)`. Esto invoca la función de contenedor de definición de un módulo (pasando en cualquier dependencia) y almacena el valor de retorno, la API del módulo, en una lista interna de módulos rastreados por nombre.

Y aquí es cómo podría utilizarlo para definir algunos módulos:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Los módulos " `foo` " y " `bar` " se definen con una función que devuelve una API pública. " `foo` " incluso recibe la instancia de " `bar` " como un parámetro de dependencia, y puede usarlo en consecuencia.

Pasa algún tiempo examinando estos fragmentos de código para comprender completamente el poder de los closures puestos a usar para nuestros propios propósitos. La clave para llevar es que no hay realmente ninguna "magia" particular para los administradores de módulos. Reúnen ambas características del patrón de módulo I enumerado anteriormente: invocando un contenedor de definición de función y manteniendo su valor de retorno como API para ese módulo.

En otras palabras, los módulos son sólo módulos, incluso si usted pone una herramienta de envoltura amigable encima de ellos.

Módulos Futuros

ES6 agrega soporte de sintaxis de primera clase para el concepto de módulos. Cuando se carga a través del sistema de módulos, ES6 trata un archivo como un módulo separado. Cada módulo puede importar otros módulos o miembros específicos de la API, así como exportar sus propios miembros API públicos.

Nota: Los módulos basados en funciones no son un patrón estáticamente reconocido (algo que el compilador conoce), por lo que su API semántica no se consideran hasta el momento de la ejecución. Es decir, puede modificar realmente la API de un módulo durante el tiempo de ejecución (véase la anterior discusión publicAPI).

Por el contrario, las API del módulo ES6 son estáticas (las API no cambian en tiempo de ejecución). Dado que el compilador lo sabe, puede (y lo hace!) comprobar durante la compilación (carga de archivos y) que existe una referencia a un miembro de la API de un módulo importado. Si no existe la referencia de API, el compilador produce un error "anticipado" en tiempo de compilación, en lugar de esperar la resolución dinámica en tiempo de ejecución tradicional (y los errores, si los hay).

Los módulos ES6 no tienen un formato "en línea", sino que deben definirse en archivos separados (uno por módulo). Los navegadores / motores tienen un "cargador de módulos" por defecto (que es sobre-escribible, pero eso está bien más allá de nuestra discusión aquí) que sincronizadamente carga un archivo de módulo cuando se importa.

Considere:

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// import only `hello()` from the "bar" module
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

```
// import the entire "foo" and "bar" modules
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Nota: Deberían crearse archivos separados " `foo.js` " y " `bar.js` ", con el contenido como se muestra en los dos primeros fragmentos, respectivamente. A continuación, su programa cargará / importará esos módulos para usarlos, como se muestra en el tercer fragmento.

`import` importa uno o más miembros de la API de un módulo en el ámbito actual, cada uno a una variable enlazada (`hello` en nuestro caso). `module` importa una API de módulo completa a una variable enlazada (`foo` , `bar` en nuestro caso). `export` exporta un identificador (variable, función) a la API pública para el módulo actual. Estos operadores pueden utilizarse tantas veces como sea necesario en la definición de un módulo.

El contenido dentro del archivo de módulo se trata como si estuviera encerrado en un closure de ámbito, al igual que con los módulos de función de closures vistos anteriormente.

5.6 Revisión (TL; DR)

El closure parece a los no ilustrados como un mundo místico separado dentro de JavaScript que sólo las pocas almas más valientes pueden alcanzar. Pero en realidad es sólo un hecho estándar y casi obvio de cómo escribimos código en un entorno de ámbito léxico, donde las funciones son valores y se pueden transmitir a voluntad.

El closure es cuando una función puede recordar y acceder a su alcance léxico incluso cuando se invoca fuera de su alcance léxico.

Los closure pueden hacernos tropezar, por ejemplo con bucles, si no tenemos cuidado de reconocerlos y cómo funcionan. Pero también son una herramienta inmensamente poderosa, permitiendo patrones como módulos en sus diversas formas.

Los módulos requieren dos características clave: 1) invocación de una función de envoltura externa, para crear el ámbito de inclusión y 2) el valor de retorno de la función de envoltura debe incluir referencia a al menos una función interna que tiene closure sobre el ámbito interno privado del envoltorio.

Ahora podemos ver closures alrededor de nuestro código existente, y tenemos la capacidad de reconocer y aprovecharlos para nuestro propio beneficio!

6- Scope Dinámico

En el capítulo 2, hablamos de "Ámbito dinámico" como un contraste con el modelo "Ámbito Léxico", que es cómo funciona el ámbito en JavaScript (y de hecho, la mayoría de los otros lenguajes).

Vamos a examinar brevemente el alcance dinámico, para martillar sobre las diferencias. Pero, lo que es más importante, el alcance dinámico es realmente un primo cercano a otro mecanismo (`this`) en JavaScript, que cubrimos en el título "This y los Prototipos de Objetos" de esta serie de libros.

Como vimos en el capítulo 2, el ámbito léxico es el conjunto de reglas sobre cómo el motor puede buscar una variable y dónde la encontrará. La característica clave del alcance léxico es que se define a tiempo de autor, cuando se escribe el código (suponiendo que no lo engañe con `eval()` o `with`).

El alcance dinámico parece implicar, y por buena razón, que hay un modelo por el cual el alcance se puede determinar dinámicamente en tiempo de ejecución, en vez de estaticamente a tiempo de autor. Ése es de hecho el caso. Vamos a ilustrarlo a través del código:

```
function foo() {
    console.log( a ); // 2
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

El ámbito léxico sostiene que la referencia RHS a un in `foo()` será resuelta a la variable global `a` , lo que dará como resultado el valor `2` .

Por el contrario, **el ámbito dinámico no se ocupa de cómo y dónde se declaran las funciones y los ámbitos, sino de dónde se llaman**. En otras palabras, la cadena de alcance se basa en la pila de llamadas (call-stack), no en la anidación de ámbitos en el código.

Por lo tanto, si JavaScript tiene un ámbito dinámico, cuando se ejecuta `foo()`, teóricamente el código siguiente resultará en `3` como salida.

```
function foo() {
    console.log( a ); // 3  (not 2!)
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

¿Cómo puede ser posible esto? Debido a que cuando `foo()` no puede resolver la referencia de variable para `a`, en lugar de intensificar la cadena de alcance anidada (lexical), va hasta la pila de llamadas, para encontrar dónde se llamó `foo()`. Desde que `foo()` fue llamado desde `bar()`, comprueba las variables en el ámbito de `bar()`, y encuentra un `a` con valor `3`.

¿Extraño? Probablemente pienses así, por el momento.

Pero eso es sólo porque probablemente sólo has trabajado en (o al menos profundamente considerado) código que tiene un alcance léxico. Por lo tanto, el alcance dinámico parece extraño. Si sólo hubieras escrito código en un lenguaje con un alcance dinámico, parecería natural, y el ámbito léxico sería la cosa extraña.

Para ser claro, J**avaScript no tiene, de hecho, un ámbito dinámico**. Tiene alcance léxico. Llano y simple. Pero el mecanismo de este tipo es como un ámbito dinámico.

El contraste clave: **el ámbito léxico es write-time, mientras que el ámbito dinámico (y `this`!) son runtime**. El alcance lexical cuida donde una función *fue declarada*, pero los concerns dinámicas del alcance de donde una función *fue llamada*.

Por último: `this` le importa cómo se llamó una función, lo que demuestra lo estrechamente relacionado que este mecanismo esta con la idea del alcance dinámico. Para profundizar más en `this`, lea el título "this & Object Prototipos".

7- Ámbito de bloque de Polyfilling

En el Capítulo 3, exploramos el Ámbito de Bloque. Vimos que `with` y la cláusula `catch` son ejemplos minúsculos de ámbito de bloque que han existido en JavaScript al menos desde la introducción de ES3.

Pero es la introducción de ES6 de `let` que finalmente brinda capacidad de alcance de bloque completo y sin restricciones a nuestro código. Hay muchas cosas interesantes, tanto funcionales como de estilo de código, que permitirán el alcance de bloque.

Pero, ¿qué pasaría si quisieramos utilizar el ámbito de bloque en los entornos pre-ES6?

Considere este código:

```
{
    let a = 2;
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

Esto funcionará muy bien en entornos ES6. Pero, ¿podemos hacerlo con las versiones anteriores de ES6? `catch` es la respuesta.

```
try{throw 2}catch(a){
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

Whoa! Ese es un código feo, extraño. Vemos un try / catch que parece arrojar un error forzosamente, pero el "error" que lanza es sólo un valor `2`, y luego la declaración de variables que lo recibe está en la cláusula `catch(a)`. Mente: explota!.

Así es, la cláusula `catch` tiene un alcance de bloque, lo que significa que puede usarse como un polyfill para el ámbito de bloque en entornos pre-ES6.

"Pero ...", dices. "... nadie quiere escribir un código así de feo!" Es verdad. Nadie escribe (algunos) el código generado por el compilador de CoffeeScript, tampoco. Ese no es el punto.

El punto es que las herramientas pueden transpilar el código ES6 para trabajar en entornos pre-ES6. Puede escribir código utilizando el ámbito de bloque y beneficiarse de dicha funcionalidad y dejar que una herramienta de creación de pasos se encargue de producir código que funcionará de verdad cuando se implementa.

Éste es realmente el camino de migración preferido para todos (ahem, la mayoría) de ES6: usar un transpilador de código para tomar código ES6 y producir código compatible con ES5 durante la transición de pre-ES6 a ES6.

7.1 Traceur

Google mantiene un proyecto llamado "Traceur" [^ note-traceur], que tiene la tarea de transponer las funciones de ES6 en pre-ES6 (en su mayoría ES5, pero no todas!) Para uso general. El comité TC39 se basa en esta herramienta (y otros) para probar la semántica de las características que especifican.

¿Qué produce Traceur a partir de nuestro fragmento? ¡Lo adivinaste!

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

Por lo tanto, con el uso de estas herramientas, podemos empezar a aprovechar el ámbito del bloque sin importar si estamos apuntando a ES6 o no, porque `try` / `catch` ha estado alrededor (y trabajado de esta manera) desde los días de ES3.

7.2 Bloques implícitos vs. explícitos

En el capítulo 3, hemos identificado algunos peligros potenciales para la capacidad de mantenimiento / refactorabilidad del código cuando introducimos el ámbito de bloque. ¿Hay otra manera de aprovechar el ámbito de bloque pero para reducir este inconveniente?

Considere esta forma alternativa de `let` , llamada "let block" o "let statement" (en contraste con "let declarations" de antes).

```
let (a = 2) {
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

En lugar de implícitamente secuestrar un bloque existente, la instrucción `let` crea un bloque explícito para su vinculación de ámbito. No sólo el bloque explícito se destaca más, y tal vez más robusto en la refactorización de código, produce un código más limpio, gramaticamente, forzando todas las declaraciones a la parte superior del bloque. Esto hace que sea más fácil mirar a cualquier bloque y saber cuál es scoped a ella y cual no.

Como patrón, refleja el enfoque que muchas personas adoptan en el scope de funciones cuando mueven / elevan manualmente todas sus declaraciones `var` a la parte superior de la función. El let-statement los coloca allí en la parte superior del bloque por intención, y si no usas las declaraciones diseminadas, tus declaraciones de alcance de bloque son algo más fáciles de identificar y mantener.

Pero, hay un problema. El formulario let-statement no está incluido en ES6. Tampoco el compilador oficial de Traceur acepta esa forma de código.

Tenemos dos opciones. Podemos formatearlo usando la sintaxis válida de ES6 y un poco de disciplina en el código:

```
/*let*/ { let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

Pero, las herramientas están destinadas a resolver nuestros problemas. Por lo tanto, la otra opción es escribir bloques explícitos de instrucciones y dejar que una herramienta los convierta en código válido de trabajo.

Por lo tanto, he construido una herramienta llamada "let-er" [^ note-let_er] para abordar sólo este problema. Let-er es un transpilador de código de build-step, pero su única tarea es encontrar formularios let-statement y transpilarlos. Dejará solo cualquiera del resto de su código, incluyendo cualquier let-declaraciones. Puede usar let-er como el primer paso del transpilador ES6 y, a continuación, pasar su código a través de algo como Traceur si es necesario.

Además, let-er tiene un indicador de configuración --es6, que cuando se enciende (desactivado por defecto), cambia el tipo de código producido. En lugar del `try` / `catch` ES3 polyfill hack, let-er tomaría nuestro fragmento y produce uno totalmente compatible con ES6, no hacky:

```
{
    let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

Por lo tanto, puede empezar a utilizar let-er de inmediato y orientar todos los entornos pre-ES6, y cuando sólo se preocupan por ES6, puede agregar el indicador y orientar instantáneamente sólo ES6.

Y lo más importante, puede utilizar el formulario de declaración de instrucciones más preferible y más explícito aunque no sea parte oficial de ninguna versión de ES (todavía).

7.3 Rendimiento

Permítanme añadir una última nota rápida sobre el rendimiento de `try` / `catch`, y / o para abordar la pregunta, "¿por qué no sólo utilizar un IIFE para crear el scope?"

En primer lugar, el rendimiento de `try` / `catch` es más lento, pero no hay supuesto razonable de que tiene que ser de esa manera, o incluso que siempre será de esa manera. Dado que el transportista oficial de ES6 aprobado por TC39 utiliza `try` / `catch`, el equipo de Traceur ha pedido a Chrome que mejore el rendimiento de `try` / `catch`, y obviamente están motivados para hacerlo.

En segundo lugar, IIFE no es una comparación de manzanas a manzanas justa con `try` / `catch`, porque una función envuelta alrededor de cualquier código arbitrario cambia el significado, dentro de ese código, de esto, `return`, `break` y `continue`. IIFE no es un sustituto general adecuado. Sólo se puede utilizar manualmente en ciertos casos.

La pregunta realmente se convierte en: ¿quieres bloquear el ámbito, o no. Si lo hace, estas herramientas le proporcionan esa opción. Si no es así, sigue usando `var` y continúa tu codificación!

[^note-traceur]:Google Traceur

[^note-let_er]:let-er

8- Lexical-this

Aunque este título no aborda este mecanismo `this` en detalle, hay un tema ES6 que relaciona esto con el alcance léxico de una manera importante, que examinaremos rápidamente.

ES6 agrega una forma sintáctica especial de declaración de función llamada "función de flecha (arrow function)". Se parece a esto:

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

La llamada "flecha gorda (fat arrow)" se menciona a menudo como una palabra corta para la palabra clave de `function` tediosamente verbosa (sarcasmo).

Pero hay algo mucho más importante en las funciones de flecha que no tiene nada que ver con el ahorro de pulsaciones de teclado en su declaración.

En pocas palabras, este código sufre un problema:

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome";

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome
```

El problema es la pérdida de `this` binding en la función `cool()`. Hay varias maneras de abordar ese problema, pero una solución frecuentemente repetida es `var self = this ;`.

Que podría parecer:

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

Sin llegar demasiado a las malas hierbas aquí, la "solución" `var self = this` simplemente dispensa todo el problema de la comprensión y el uso adecuado de `this` binding, y en su lugar vuelve a algo con lo que estamos quizás más cómodos: ámbito léxico. El `self` se convierte en un simple identificador que puede ser resuelto a través del alcance léxico y el closure, y no se preocupa de lo que le pasó a el binding de `this` a lo largo del camino.

A la gente no le gusta escribir cosas verbosas, especialmente cuando lo hacen una y otra vez. Por lo tanto, una motivación de ES6 es ayudar a aliviar estos escenarios, y, de hecho, corregir problemas del lenguaje común, como este.

La solución ES6, la función de flecha, introduce un comportamiento llamado "lexical this".

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // arrow-function ftw?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

La breve explicación es que las funciones de flecha no se comportan en absoluto como las funciones normales cuando se trata de la vinculación de `this` . Descartan todas las reglas normales para la vinculación de `this` , y en cambio asumen el valor de `this` y el alcance inmediato de su inclusión léxica, cualquiera que sea.

Por lo tanto, en ese fragmento, la función de flecha no obtiene su `this` sin consolidarlo de alguna manera impredecible, simplemente "hereda" el `this` de la función `cool()` (que es correcto si lo invocamos como se muestra!).

Mientras que esto hace para el código más corto, mi perspectiva es que las funciones de la flecha son realmente apenas las que codifican en la sintaxis del lenguaje un error común de los reveladores, que es confundir y combinar reglas "que vinculan" con las reglas de "alcance lexical".

Dicho de otra manera: ¿por qué ir al problema y la verbosidad de usar el paradigma del estilo `this` de codificación, sólo para cortar en las rodillas, mezclándolo con referencias léxicas. Parece natural abrazar un enfoque o el otro para cualquier pieza de código, y no mezclarlos en el mismo código.

Nota: otra detracción de las funciones de flecha es que son anónimas, no nombradas. Consulte el Capítulo 3 para ver las razones por las cuales las funciones anónimas son menos deseables que las funciones con nombre.

Un enfoque más apropiado, en mi perspectiva, a este "problema", es usar y abrazar correctamente el mecanismo `this`.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // `this` is safe because of `bind(..)`
                console.log( "more awesome" );
            }.bind( this ), 100 ); // look, `bind()`!
        }
    }
};

obj.cool(); // more awesome
```

Ya sea que prefiera el nuevo vocabulario léxico-this comportamiento de las funciones de las flechas, o prefiere el `bind()` probado-y-`true()`, es importante tener en cuenta que las funciones flecha no son solo para escribir abreviadamente "function".

Tienen una diferencia de comportamiento intencional que debemos aprender y entender, y si así lo deseamos, aprovechar.

Ahora que comprendemos plenamente el alcance léxico (y el closure!), La comprensión léxico-this debe ser una brisa!

III- this & Object Prototypes

Prefacio

Al leer este libro en preparación para escribir este prólogo, me vi obligado a reflexionar sobre cómo aprendí JavaScript y cuánto ha cambiado en los últimos 15 años que he estado programando y desarrollando con él.

Cuando empecé a usar JavaScript hace 15 años, la práctica de usar tecnologías no HTML como CSS y JS en sus páginas web se llamaba DHTML o HTML dinámico. En aquel entonces, la utilidad de JavaScript variaba enormemente y parecía estar inclinada hacia la adición de copos de nieve animados a sus páginas web o relojes dinámicos que decía la hora en la barra de estado. Basta con decir que realmente no presté mucha atención a JavaScript en la primera parte de mi carrera debido a la novedad de las implementaciones que he encontrado a menudo en Internet.

No fue hasta 2005 que redescubrí por primera vez JavaScript como un lenguaje de programación real al que necesitaba prestar más atención. Después de excavar en la primera versión beta de Google Maps, me enganchó el potencial que tenía. En ese momento, Google Maps era una aplicación de primera clase que le permitía mover un mapa con el ratón, acercar y alejar, y hacer solicitudes de servidor sin recargar la página, todo ello con JavaScript. ¡Parecía mágico!

Cuando algo parece mágico, suele ser una buena indicación de que estás en el amanecer de una nueva forma de hacer las cosas. Y muchacho, no me equivoqué - el rápido crecimiento a hoy, yo diría que Javascript es uno de los lenguajes primarios que utilizo para la programación del lado del cliente y del servidor, y no lo haría de cualquier otra manera.

Uno de mis arrepentimientos al mirar en los últimos 15 años es que no le di a JavaScript más de una oportunidad antes de 2005, o más exactamente, que me faltó la previsión para ver JavaScript como un verdadero lenguaje de programación que es tan útil como C++, C #, Java, y muchos otros.

Si tuviera esta serie de libros de You Do not Know JS al comienzo de mi carrera, mi historia se vería muy diferente de lo que es hoy. Y esa es una de las cosas que me encanta acerca de esta serie: explica JS a un nivel que construye su comprensión a medida que avanza la serie, pero de una manera divertida e informativa.

this & Object Prototypes es una continuación maravillosa a la serie. Hace un trabajo grande y natural de construir en el libro anterior, scope y closures, y extender ese conocimiento a una parte muy importante del lenguaje de JS, la palabra clave `this` y prototipos. Estas dos cosas simples son fundamentales para lo que aprenderás en los libros futuros, porque son fundamentales para hacer una programación real con JavaScript. El concepto de cómo crear objetos, relacionarlos y ampliarlos para representar cosas en su aplicación es necesario para crear aplicaciones grandes y complejas en JavaScript. Y sin ellos, la creación de aplicaciones complejas (como Google Maps) no sería posible en JavaScript.

Yo diría que la gran mayoría de los desarrolladores web probablemente nunca han construido un objeto JavaScript y sólo tratan el lenguaje como pegamento vinculante de eventos entre los botones y las solicitudes AJAX. Yo estaba en ese campo en un momento de mi carrera, pero después de aprender a dominar prototipos y crear objetos en JavaScript, un mundo de posibilidades se abría para mí. Si usted cae en la categoría de sólo la creación de eventos vinculante y código de pegamento, este libro es una lectura obligada; Si sólo necesita un refresco, este libro será un recurso para usted. De cualquier manera, no te decepcionará. ¡Créeme!

Nick Berardi

Nickberardi.com, @nberardi

0- Prefacio

Estoy seguro de que se dio cuenta, pero "JS" en el título de la serie de libros no es una abreviatura de palabras utilizadas para maldecir sobre JavaScript, aunque maldecir a las peculiaridades del lenguaje es algo con lo que probablemente todos se pueden identificar.

Desde los primeros días de la web, JavaScript ha sido una tecnología fundamental que ha impulsado la experiencia interactiva en torno al contenido que consumimos. Mientras que el apuntador parpadeante y los molestos avisos emergentes podrían ser donde comenzó JavaScript, casi dos décadas después, la tecnología y la capacidad de JavaScript ha crecido en muchos ámbitos de magnitud, y pocos dudan de su importancia en el corazón de la plataforma de software más ampliamente disponible: La web.

Pero como lenguaje, ha sido perpetuamente un blanco para mucha crítica, debido en parte a su "herencia", pero más aún debido a su filosofía de diseño. Incluso el nombre evoca, como Brendan Eich una vez lo dijo, "el hermano bebe estúpido" comparado junto a su hermano mayor más maduro "Java". Pero el nombre es simplemente un accidente de la política y el marketing. Los dos idiomas son muy diferentes en muchos aspectos importantes. "JavaScript" está relacionado con "Java" como "Carnival" es a "Car".

Debido a que JavaScript toma conceptos y sintaxis idiomáticos de varios idiomas, incluyendo orgullosas raíces procedimentales de estilo C, así como raíces funcionales sutiles, menos obvias de estilo Scheme/Lisp, es sumamente accesible a una amplia audiencia de desarrolladores, incluso aquellos con poca o sin experiencia en programación. El "Hello World" de JavaScript es tan simple que el idioma se hace atractivo y es fácil de ponerse cómodo con la comprensión temprana.

Mientras que JavaScript es quizás uno de los idiomas más fáciles de poner en funcionamiento, sus excentricidades hacen que el dominio sólido del lenguaje sea una ocurrencia mucho menos común que en muchos otros idiomas. Cuando se necesita un conocimiento bastante profundo de un lenguaje como C o C++ para escribir un programa a gran escala, la producción a gran escala de JavaScript puede, y con frecuencia lo es, apenas se raya con la superficie de lo que el lenguaje puede realmente hacer.

Los conceptos sofisticados, que están profundamente arraigados en el lenguaje, tienden a aparecer en formas aparentemente simplistas, como pasar las funciones como devoluciones de llamada, lo que anima al desarrollador de JavaScript a utilizar el lenguaje tal como está y ha no preocuparse demasiado por lo que está pasando bajo la capucha.

Es simultáneamente un lenguaje sencillo y fácil de usar que tiene un amplio atractivo y una colección compleja y matizada de mecánica del lenguaje que sin un estudio cuidadoso escapará a la verdadera comprensión, incluso de los más experimentados desarrolladores de JavaScript.

Ahí radica la paradoja de JavaScript, el talón de Aquiles de la lengua, el desafío que estamos abordando actualmente. Debido a que JavaScript se puede utilizar sin comprender, la comprensión del lenguaje a menudo nunca se logra.

Misión

Si en cada punto que encuentres una sorpresa o frustración en JavaScript, tu respuesta es añadirlo a la lista negra, como algunos están acostumbrados a hacer, pronto serás relegado a una concha hueca de la riqueza de JavaScript.

Si bien este subconjunto ha sido conocido como "The Good Parts", le pediría a usted, querido lector, que lo considere "The Easy Parts", "The Safe Parts" o incluso "The Incomplete Parts".

Esta serie de libros de Javascript ofrece un desafío contrario: aprenda y entienda profundamente todo Javascript, incluso y especialmente "las piezas resistentes (The Tough Parts)".

Aquí, nos dirigimos a los desarrolladores de JS que tienen en la cabeza la mentalidad de aprender "lo suficiente" para continuar, sin nunca obligarse a aprender exactamente cómo y por qué el lenguaje se comporta de la manera que lo hace. Además, evitamos el consejo común de retirarse cuando el camino se vuelve áspero.

No estoy contento, ni debes estar, en parar una vez que algo funciona correctamente, y no saber realmente por qué. Le desafío a viajar por ese "camino costoso" y abrazar todo lo que JavaScript es y puede hacer. Con ese conocimiento, ninguna técnica, ningún framework, ningún acrónimo popular de moda, estará más allá de su comprensión.

Estos libros toman partes específicas del lenguaje que son las comúnmente mal entendidas o no comprendidas, y se sumerge muy profundamente y exhaustivamente en ellas. Usted debe termina la lectura con una firme confianza en su comprensión, no sólo de lo teórico, sino lo práctico "lo que necesita saber".

El JavaScript que usted conoce ahora mismo es probablemente la partes dada a usted por otros que han sido "quemados" por una comprensión incompleta. Ese JavaScript es sólo una sombra del verdadero lenguaje. Realmente no sabes JavaScript, pero si crees en esta serie, lo harás. Sigue leyendo, amigos. JavaScript te espera.

Resumen

JavaScript es impresionante. Es fácil de aprender en parte, y mucho más difícil de aprender por completo (o incluso lo suficiente). Cuando los desarrolladores encuentran confusión, suelen culpar al idioma en lugar de su falta de comprensión. Estos libros apuntan a arreglar eso, inspirando una apreciación fuerte del lenguaje para que usted pueda entenderlo ahora, y deba, profundamente entenderlo.

Nota: Muchos de los ejemplos en este libro asumen los entornos de motor de JavaScript modernos (y futuros), como ES6. Es posible que algunos códigos no funcionen como se describe si se ejecutan en motores anteriores (pre-ES6).

1- `this` o That?

Uno de los mecanismos más confusos en JavaScript es la palabra clave `this`. Es una palabra clave de identificador especial que se define automáticamente en el ámbito de cada función, pero a lo que se refiere exactamente podría molestar incluso a los desarrolladores de JavaScript más experimentados.

> Cualquier tecnología suficientemente avanzada es indistinguible de la magia. - Arthur C. Clarke

El mecanismo `this` en JavaScript no es realmente tan avanzado, pero los desarrolladores a menudo parafrasean esa cita en su propia mente insertando un aura de "complejo" o "confuso", y no hay duda de que sin una falta de comprensión clara, `this` puede parecer francamente mágico en su confusión.

Nota: La palabra " `this` " es un pronombre terriblemente común en el discurso general. Por lo tanto, puede ser muy difícil, especialmente verbalmente, determinar si estamos utilizando "este" como un pronombre o usarlo para referirnos al identificador de palabra clave real. Para mayor claridad, utilizaré siempre `this` para referirme a la palabra clave especial, y "this" o *this* o de otra manera.

1.1 ¿Porque `this` ?

Si el mecanismo `this` es tan confuso, incluso para los desarrolladores de JavaScript experimentados, uno puede preguntarse por qué es incluso útil? ¿Es más problemático de lo que vale? Antes de saltar en el cómo, debemos examinar el por qué.

Intentemos ilustrar la motivación y la utilidad de `this` :

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

Si el cómo de este fragmento te confunde, ¡no te preocupes! Llegaremos a eso en breve. Simplemente deja esas preguntas de lado por ahora para que podamos estudiar el porqué más claramente.

Este fragmento de código permite que las funciones `identifier()` y `speak()` sean reutilizadas en contextos múltiples (`me` y `you`), en lugar de necesitar una versión separada de la función para cada objeto.

En lugar de confiar en `this` , usted podría haberlo pasado explícitamente en un objeto de contexto tanto para `identifier()` como para `speak()` .

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

Sin embargo, el mecanismo `this` proporciona una forma más elegante de "pasar" implícitamente una referencia de objeto, lo que conduce a un diseño de API más limpio y una reutilización más fácil.

Cuanto más complejo es su patrón de uso, más claramente verá que pasar el contexto alrededor como un parámetro explícito es a menudo más desordenado que pasar alrededor del contexto de `this`. Cuando exploramos objetos y prototipos, veremos la utilidad de una colección de funciones que pueden referenciar automáticamente al objeto de contexto apropiado.

1.2 Confusiones

Pronto comenzaremos a explicar cómo `this` funciona realmente, pero primero debemos disipar algunos conceptos erróneos sobre cómo realmente NO funciona.

El nombre "this" crea confusión cuando los desarrolladores tratan de pensarlo demasiado literalmente. Hay dos significados a menudo asumidos, pero ambos son incorrectos.

Itself

La primera tentación común es asumir que `this` se refiere a la función misma. Esa es una inferencia gramatical razonable, por lo menos.

¿Por qué quieres referirte a una función desde dentro de sí mismo? Las razones más comunes serían cosas como la recursión (llamando a una función desde dentro de sí misma) o tener un manejador de eventos que pueda desvincularse cuando se llama por primera vez.

Los desarrolladores nuevos en los mecanismos de JS a menudo piensan que hacer referencia a la función como un objeto (todas las funciones en JavaScript son objetos!) le permite almacenar el estado (valores en las propiedades) entre las llamadas de función. Aunque esto es ciertamente posible y tiene algunos usos limitados, el resto del libro explicará en muchos otros patrones para mejores lugares para almacenar el estado además del objeto de función.

Pero por un momento exploraremos ese patrón, para ilustrar cómo `this` no permite que una función obtenga una referencia a sí misma tal como podríamos haber asumido.

Considere el siguiente código, donde intentamos rastrear cuántas veces se llamó una función (`foo`):

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 0 -- WTF?
```

`foo.count` sigue siendo `0` , aunque las cuatro sentencias `console.log` indican claramente que `foo(...)` se llamó en realidad cuatro veces. La frustración se deriva de una interpretación demasiado literal de lo que significa (en `this.count++`).

Cuando el código ejecuta `foo.count = 0` , de hecho agrega una propiedad `count` al objeto de función `foo` . Sin embargo, para la referencia `this.count` dentro de la función, `this` no apunta en absoluto a ese objeto de función, y así, aunque los nombres de propiedad son los mismos, los objetos raíz son diferentes y se produce confusión.

Nota: Un desarrollador responsable debe preguntar en este momento: "Si estaba incrementando una propiedad `count` pero no era la que esperaba, ¿cuál fue el `count` que incrementé?" De hecho, si cavara más profundamente, descubriría que había creado accidentalmente una variable global `count` (véase el Capítulo 2 para saber cómo sucedió!), Y actualmente tiene el valor `NaN` . Por supuesto, una vez que identifica este resultado peculiar, entonces tiene un conjunto de preguntas más: "¿Cómo era global, y por qué terminó `NaN` en lugar de algún valor `count` adecuado?" (Véase el capítulo 2).

En lugar de detenerse en este punto y cavar en por qué la referencia `this` no parece estar comportándose como se esperaba, y responder a esas preguntas difíciles pero importantes, muchos desarrolladores simplemente evitan el problema en conjunto, y hackean hacia otra solución, como la creación de otro Objeto para contener la propiedad `count` :

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( data.count ); // 4
```

Si bien es cierto que este enfoque "resuelve" el problema, desafortunadamente simplemente ignora el problema real -la falta de comprensión de lo que `this` significa y cómo funciona- y vuelve a caer en la zona de confort de un mecanismo más familiar: alcance léxico .

Nota: El alcance léxico es un mecanismo perfectamente fino y útil; No estoy menospreciando el uso de la misma, por cualquier medio (ver el título "Scope & Closures" de esta serie de libros). Pero constantemente adivinar cómo usar `this` , y por lo general estar equivocado, no es una buena razón para retroceder de nuevo al ámbito léxico y nunca aprender por qué esto le escapa.

Para hacer referencia a un objeto de función desde dentro de sí mismo, `this` por sí mismo suele ser insuficiente. Generalmente se necesita una referencia al objeto de función a través de un identificador léxico (variable) que apunte hacia él.

Considere estas dos funciones:

```
function foo() {
    foo.count = 4; // `foo` refers to itself
}

setTimeout( function(){
    // anonymous function (no name), cannot
    // refer to itself
}, 10 );
```

En la primera función, llamada "función nombrada", `foo` es una referencia que se puede utilizar para referirse a la función desde dentro de sí misma.

Pero en el segundo ejemplo, la función de devolución de llamada pasada a `setTimeout(..)` no tiene ningún identificador de nombre (llamada "función anónima"), por lo que no hay forma adecuada de referirse al objeto de función en sí misma.

Nota: La referencia a la vieja escuela, pero ahora desaprobada y con el ceño fruncido `arguments.callee`. La referencia dentro de una función también apunta al objeto de función de la función en ejecución. Esta referencia es típicamente la única manera de acceder al objeto de una función anónima desde dentro de sí misma. El mejor enfoque, sin embargo, es evitar el uso de funciones anónimas por completo, al menos para aquellos que requieren una auto-referencia, y en su lugar utilizar una función nombrada (expresión). `arguments.callee` está obsoleto y no debe utilizarse.

Por lo tanto, otra solución a nuestro ejemplo de ejecución habría sido usar el identificador `foo` como una referencia de objeto de función en cada lugar, y no usar `this` en absoluto, lo cual funciona:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

Sin embargo, ese enfoque de manera similar a los pasos laterales de la comprensión real de `this` y se basa enteramente en el ámbito léxico de la variable `foo`.

Sin embargo, otra forma de abordar el problema es forzar `this` a apuntar realmente al objeto de función `foo`:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    // Note: `this` IS actually `foo` now, based on
    // how `foo` is called (see below)
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // using `call(..)`, we ensure the `this`
        // points at the function object (`foo`) itself
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

En lugar de evitar `this`, lo abrazamos. Vamos a explicar en un momento cómo funcionan estas técnicas de trabajo mucho más completamente, por lo que no te preocupes si todavía estás un poco confundido!

Su alcance

El siguiente error más común sobre el significado de `this` es que de alguna manera se refiere al alcance de la función. Es una pregunta difícil, porque en cierto sentido hay algo de verdad, pero en el otro sentido, es bastante equivocado.

Para ser claro, `this` no se refiere, en modo alguno, al **ámbito léxico** de una función. Es cierto que internamente, el alcance es como un objeto con propiedades para cada uno de los identificadores disponibles. Pero el "objeto" de alcance no es accesible al código JavaScript. Es una parte interna de la implementación del Motor.

Considere el código que intenta (y falla!) cruzar el límite y usa `this` para referirse implícitamente al ámbito léxico de una función:

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); //undefined
```

Hay más de un error en este fragmento. Aunque puede parecer artificial, el código que ves es una destilación del código real del mundo real que se ha intercambiado en los foros públicos de ayuda de la comunidad. Es una ilustración maravillosa (si no triste) de lo equivocados que pueden ser estos supuestos sobre `this`.

En primer lugar, se intenta hacer referencia a la función `bar()` a través de `this.bar()`. Es casi ciertamente un accidente que funcione, pero vamos a explicar el cómo en breve. La forma más natural de haber invocado `bar()` habría sido omitir la declaración `this`. Y sólo hacer una referencia léxica al identificador.

Sin embargo, el desarrollador que escribe este código intenta usar `this` para crear un puente entre los ámbitos léxicos de `foo()` y `bar()`, de modo que `bar()` tenga acceso a la variable `a` en el ámbito interno de `foo()`. **Ningún puente es posible**. No puedes usar esta referencia para buscar algo en un ámbito léxico. No es posible.

Cada vez que se sientan tratando de mezclar perspectivas de alcance léxico con `this`, recuerde: no hay puente.

1.3 ¿Que es `this`?

Después de apartar varias suposiciones incorrectas, volvamos ahora nuestra atención a cómo funciona realmente el mecanismo `this`.

Hemos dicho anteriormente que `this` no es una vinculación autor-tiempo sino un enlace de tiempo de ejecución. Es contextual basado en las condiciones de la invocación de la función. El alcance `this` no tiene nada que ver con el lugar donde se declara una función, sino que tiene en cambio que ver con la forma en que se llama a la función.

Cuando se invoca una función, se crea un registro de activación, también conocido como contexto de ejecución. Este registro contiene información sobre dónde se llamó la función (la pila de llamadas), cómo se invocó la función, qué parámetros se pasaron, etc. Una de las propiedades de este registro es la referencia de `this` que se utilizará durante la duración de la ejecución de esa función.

En el próximo capítulo, aprenderemos a encontrar el **sitio de llamada** de una función para determinar cómo su ejecución vinculará a `this`.

1.4 Revisión (TL; DR)

El enlace `this` es una fuente constante de confusión para el desarrollador de JavaScript que no se toma el tiempo para aprender cómo funciona realmente el mecanismo. Las conjeturas, el ensayo y el error y el copia y pegue de las respuestas de StackOverflow no son una forma efectiva o adecuada de aprovechar este importante mecanismo.

Para aprender `this`, primero tiene que aprender lo que no es `this`, a pesar de cualquier suposición o concepto erróneo que pueden llevarlo por esos caminos. `this` no es ni una referencia a la función en sí, ni una referencia al ámbito léxico de la función.

`this` es en realidad un enlace que se realiza cuando se invoca una función y lo que hace referencia se determina totalmente por el sitio de llamada donde se llama la función.

2- `this`, todo tiene sentido ahora!

En el capítulo 1, hemos descartado varias ideas erróneas sobre `this` y aprendimos en su lugar que `this` se trata de una vinculación hecha para cada invocación de función, basada totalmente en su **call-site** (cómo se llama la función).

2.1 Sitio de llamada

Para entender la vinculación `this`, tenemos que entender el call-site: la ubicación en el código donde se llama una función (**no donde se declara**). Tenemos que inspeccionar el sitio de llamada para responder a la pregunta: ¿es `this` una referencia?

Encontrar el sitio de llamada se trata generalmente: "de ir a localizar desde donde se llama una función", pero no siempre es tan fácil, ya que ciertos patrones de codificación pueden oscurecer el verdadero sitio de llamada.

Lo importante es pensar en la pila de llamadas (la pila de funciones que se han llamado para llegar al momento actual en ejecución). El sitio de llamada que nos interesa es en la invocación antes de la función en ejecución.

Vamos a demostrar call-stack y call-site:

```
function baz() {
    // call-stack es: `baz`
    // así, nuestro sitio de llamada está en el ámbito global

    console.log( "baz" );
    bar(); // <-- call-site para `bar`
}

function bar() {
    // call-stack es: `baz` -> `bar`
    // así, nuestro sitio de llamada está en `baz`

    console.log( "bar" );
    foo(); // <-- call-site para `foo`
}

function foo() {
    // call-stack es: `baz` -> `bar` -> `foo`
    // asi, nuestro sitio de llamada esta en `bar`

    console.log( "foo" );
}

baz(); // <-- call-site para `baz`
```

Tenga cuidado al analizar el código para encontrar el sitio de llamada real (desde la pila de llamadas), porque es lo único que importa para el enlace de `this`.

Nota: Puede visualizar una pila de llamadas en su mente mirando la cadena de llamadas de función en orden, como hicimos con los comentarios en el fragmento anterior. Pero esto es laborioso y propenso a errores. Otra forma de ver la pila de llamadas es usar una herramienta de depuración en su navegador. La mayoría de los navegadores de escritorio modernos incorporan herramientas de desarrollo, que incluyen un depurador JS. En el fragmento anterior, podría haber establecido un punto de interrupción en las herramientas para la primera línea de la función `foo()` o simplemente insertado el `debugger` ; en la declaración de esa primera línea. Al ejecutar la página, el depurador se detendrá en esta ubicación y le mostrará una lista de las funciones que se han llamado para llegar a esa línea, que será la pila de llamadas. Por lo tanto, si está intentando diagnosticar la vinculación de `this` , utilice las herramientas de desarrollador para obtener la pila de llamadas y, a continuación, busque el segundo elemento desde la parte superior y se mostrará el sitio de llamada real.

2.2 Nada más que reglas

Volvamos nuestra atención ahora a cómo el sitio de llamada determina dónde apuntará a `this` durante la ejecución de una función.

Debe inspeccionar el sitio de llamada y determinar cuál de las 4 reglas se aplica. Primero explicaremos cada una de estas 4 reglas de forma independiente, y luego ilustraremos su orden de precedencia, si se pudieran aplicar múltiples reglas al sitio de llamada.

Vinculación por defecto

La primera regla que examinaremos viene del caso más común de las llamadas de función: invocación de función autónoma. Piense en esta regla `this` como la regla de captura por defecto cuando no se aplica ninguna otra regla.

Considere este código:

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo(); // 2
```

Lo primero que hay que tener en cuenta es que si las variables declaradas en el ámbito global, como `var a = 2` , son sinónimas con propiedades de objeto global del mismo nombre. No son copias el uno del otro, son el uno al otro. Piense en ello como dos caras de la misma moneda.

En segundo lugar, vemos que cuando `foo()` se llama, `this.a` se resuelve a nuestra variable global `a` . ¿Por qué? Porque en este caso, el enlace por defecto para `this` se aplica a la llamada de la función, y así apunta `this` al objeto global.

¿Cómo sabemos que la regla de vinculación predeterminada se aplica aquí? Examinamos el sitio de llamada para ver cómo se llama `foo()` . En nuestro fragmento, `foo()` se llama con una referencia de función simple, no decorada. Ninguna de las otras reglas que demostraremos se aplicará aquí, por lo que se aplica la vinculación por defecto.

Si el `strict mode` está en efecto, el objeto global no es elegible para el enlace predeterminado, por lo que `this` se establece como `undefined` .

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

Un detalle sutil pero importante es: aunque el conjunto de reglas de vinculación de `this` se basan totalmente en el call-site, el objeto global sólo es elegible para el enlace por defecto si el contenido de `foo()` no se ejecuta en modo estricto; El estado de modo estricto del sitio de llamada de `foo()` es irrelevante.

```
function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

Nota: La mezcla intencional de modo estricto y modo no estricto en su propio código generalmente es mal visto. Su programa entero debe ser probablemente Estricto o no Estricto. Sin embargo, a veces se incluye una biblioteca de terceros que tiene diferente Strict'ness en su propio código, por lo que hay que tener cuidado con estos sutiles detalles de compatibilidad.

Vinculación implícita

Otra regla a considerar es: el sitio de llamada tiene un objeto de contexto, también conocido como un objeto propietario o que contiene, aunque estos términos alternativos podrían ser un poco engañosos.

Considere:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

En primer lugar, observe la manera en que `foo()` se declara y luego se agrega como una propiedad de referencia en `obj`. Independientemente de si `foo()` se declara inicialmente en `obj`, o se agrega como una referencia posterior (como muestra este fragmento), en ninguno de los casos la función es realmente una "propiedad" o está "contenida" por el objeto `obj`.

Sin embargo, el sitio de llamada utiliza el contexto `obj` para referenciar la función, por lo que podría decirse que el objeto `obj` "posee" o "contiene" la referencia de función en el momento en que se llama a la función.

Lo que usted decida para llamar a este patrón, en el punto que `foo()` se llama, es precedido por una referencia de objeto a `obj`. Cuando hay un objeto de contexto para una referencia de función, la regla de vinculación implícita dice que es el objeto que se debe utilizar para la llamada de función de la vinculación de `this`.

Debido a que `obj` es lo mismo que `this` para la llamada `foo()`, `this.a` es sinónimo de `obj.a`.

Sólo el nivel superior / último de una cadena de referencia de propiedad de objeto es importante para el sitio de llamada. Por ejemplo:

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Implícitamente perdido

Una de las frustraciones más comunes que crea la vinculación de `this` es cuando una función implícitamente vinculada pierde esa vinculación, lo que generalmente significa que se vuelve a la vinculación por defecto, ya sea del objeto global o `undefined`, dependiendo del `strict mode`.

Considere:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

A pesar de que `bar` parece ser una referencia a `obj.foo`, de hecho, es realmente sólo otra referencia a la propia `foo`. Además, el call-site es lo que importa, y el call-site es `bar()`, que es una llamada simple, no decorada y por lo tanto se aplica el enlace por defecto.

La forma más sutil, más común y más inesperada en que esto ocurre es cuando consideramos pasar una función de devolución de llamada:

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` is just another reference to `foo`

    fn(); // <-- call-site!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

doFoo( obj.foo ); // "oops, global"
```

El paso de parámetros es simplemente en una asignación implícita, y puesto que estamos pasando una función, es una asignación implícita de referencia, de modo que el resultado final es el mismo que el fragmento anterior.

¿Qué pasa si la función a la que está pasando su devolución de llamada no es suya, sino integrada en el lenguaje? No hay diferencia, da el mismo resultado.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

setTimeout( obj.foo, 100 ); // "oops, global"
```

Piense en esta pseudo-implementación teórica cruda de `setTimeout()` proporcionada como un built-in del entorno JavaScript:

```
function setTimeout(fn,delay) {
    // wait (somehow) for `delay` milliseconds
    fn(); // <-- call-site!
}
```

Es bastante común que nuestras devoluciones de funciones pierdan su vinculación `this`, como acabamos de ver. Pero otra manera en que `this` puede sorprendernos es cuando la función que hemos pasado a nuestra devolución de llamada a intencionalmente cambia `this` para la llamada. Los manejadores de eventos en las bibliotecas de JavaScript populares son muy aficionados a forzar su devolución de llamada a tener un `this` al que apunta, por ejemplo, al elemento DOM que activó el evento. Mientras que a veces puede ser útil, otras veces puede ser francamente exasperante. Lamentablemente, estas herramientas rara vez le permiten elegir.

De cualquier manera `this` se cambia inesperadamente, usted no está realmente en control de cómo su referencia de función de devolución de llamada se ejecutará, así que usted no tiene ninguna manera (aún) de controlar el call-site para dar su vinculación deseada. Veremos pronto una manera de "arreglar" ese problema arreglando `this`.

Vinculación explícita

Con la vinculación implícita como acabamos de ver, tuvimos que mutar el objeto en cuestión para incluir una referencia sobre sí misma a la función, y usar esta referencia de función de propiedad para indirectamente (implícitamente) vincular `this` con el objeto.

¿Pero, qué pasa si usted quiere forzar una llamada de la función para utilizar un objeto particular para este enlace, sin poner una referencia de la función de la característica en el objeto?

Todas las funciones del lenguaje tienen algunas utilidades a su disposición (a través de su [[Prototype]]) - más sobre esto más adelante) que pueden ser útiles para esta tarea. Específicamente, las funciones tienen métodos de `call(..)` y de `apply(..)`. Técnicamente, los ambientes de host JavaScript a veces proporcionan funciones que son lo suficientemente especiales (una forma amable de decirlo!) Que no tienen dicha funcionalidad. Pero son pocos. La gran mayoría de las funciones proporcionadas, y ciertamente todas las funciones que va a crear, tienen acceso a `call(..)` y `apply(..)`.

¿Cómo funcionan estas utilidades? Ambos toman, como su primer parámetro, un objeto para usar para `this`, y luego invocan la función con el `this` especificado. Puesto que usted está indicando directamente lo que usted quiere que `this` sea, lo llamamos enlace explícito.

Considere:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Invocando `foo` con una vinculación explícita por `foo.call(..)` nos permite forzar a su `this` a ser `obj`.

Si pasa un valor primitivo simple (de tipo `string`, `boolean` o `number`) como este enlace, el valor primitivo se envuelve en su objeto-forma (`new String(..)`, `new Boolean(..)` o `new Number(..)`, respectivamente). Esto se refiere a menudo como "boxing".

Nota: Con respecto a la vinculación de `this`, el `call(..)` y `apply(..)` son idénticas. Ellos se comportan de manera diferente con sus parámetros adicionales, pero eso no es algo que nos importa en este momento.

Desafortunadamente, la vinculación explícita por sí sola todavía no ofrece ninguna solución al problema mencionado anteriormente, de una función "perder" su intención de vinculación `this`, o simplemente tenerla pavimentada por un framework, etc.

Vinculación dura (Hard Binding)

Pero un patrón de variación en torno a la vinculación explícita realmente hace el truco. Considere:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// `bar` hard binds `foo`'s `this` to `obj`
// so that it cannot be overriden
bar.call( window ); // 2
```

Veamos cómo funciona esta variación. Creamos una función `bar()` que, internamente, llama manualmente a `foo.call(obj)`, invocando forzosamente `foo` con la vinculación `obj` para `this`. No importa cómo invoque más tarde la función `bar()`, siempre se invocará manualmente `foo` con `obj`. Esta vinculación es tanto explícita como fuerte (hard), por lo que lo llamamos vinculación dura (hard binding).

La forma más típica de envolver una función con una vinculación dura crea un paso a través de cualquier argumento pasado y cualquier valor de retorno recibido:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Otra forma de expresar este patrón es crear un ayudante reutilizable:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// simple `bind` helper
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Dado que la vinculación dura es un patrón tan común, se proporciona con una utilidad integrada como ES5: `Function.prototype.bind` , y se utiliza de esta manera:

```
unction foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

`bind(..)` devuelve una nueva función que está codificada para llamar a la función original con el contexto `this` como se especificó.

Nota: A partir de ES6, la función hard-bound producida por `bind(..)` tiene una propiedad `.name` que deriva de la función de destino original. Por ejemplo: `bar = foo.bind(..)` debe tener un valor `bar.name` de " `bound foo` ", que es el nombre de la llamada a la función que debe aparecer en un stack trace.

"Contextos" de API Call

Muchas funciones de las bibliotecas, y de hecho muchas nuevas funciones incorporadas en el lenguaje JavaScript y el entorno del host, proporcionan un parámetro opcional, generalmente llamado "context", que está diseñado como una solución para no tener que usar `bind(..)` Para asegurar que su función de devolución de llamada utilice un `this` en particular.

Por ejemplo:

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// use `obj` como `this` para la llamada `foo(..)`
[1, 2, 3].forEach( foo, obj ); // 1 awesome  2 awesome  3 awesome
```

Internamente, estas diversas funciones casi seguramente usan una vinculación explícito vía `call(..)` o `apply(..)` , ahorrándole el problema.

`new` Binding

La cuarta y última regla para la vinculación `this` nos obliga a repensar un concepto erróneo muy común sobre las funciones y los objetos en JavaScript.

En los lenguajes tradicionales orientados a clases, los "constructores" son métodos especiales asociados a las clases, que cuando la clase se instancia con un operador `new` , se llama al constructor de esa clase. Esto normalmente se parece a algo así:

```
something = new MyClass(..);
```

JavaScript tiene un operador `new` , y el patrón de código para usarlo parece idéntico a lo que vemos en esos lenguajes orientados a clases; La mayoría de los desarrolladores asumen que el mecanismo de JavaScript está haciendo algo similar. Sin embargo, realmente no hay ninguna conexión a la funcionalidad orientada a clases implicada por el uso de `new` en JS.

En primer lugar, vamos a volver a definir lo que es un "constructor" en JavaScript. En JS, los constructores son sólo funciones que pasan a ser llamadas con el operador `new` delante de ellas. No están unidos a las clases, ni están instanciando una clase. Ni siquiera son tipos especiales de funciones. Son sólo funciones regulares que, en esencia, son secuestradas por el uso de `new` en su invocación.

Por ejemplo, la función `Number(..)` que actúa como constructor, citando desde la especificación ES5.1:

> 15.7.2 El Constructor de Números
>
> Cuando Number se llama como parte de una expresión `new`, es un constructor: inicializa el objeto recién creado.

Por lo tanto, casi cualquier función, incluyendo las funciones de objetos integradas como `Number(..)` (véase el capítulo 3) se puede llamar con `new` delante de ella, y hace que la función llamada sea una llamada al constructor. Esta es una distinción importante pero sutil: realmente no hay tal cosa como "funciones constructoras", sino más bien llamadas de construcción de funciones.

Cuando se invoca una función con `new` delante de ella, también conocida como llamada del constructor, se ejecutan automáticamente las siguientes acciones:

1. Un nuevo objeto se crea (aka, construido) en el aire
2. El objeto de nueva construcción es [[Prototype]] - linked
3. El objeto de nueva construcción se establece como el enlace de esa llamada de función
4. A menos que la función devuelva su propio **objeto** alternativo, la llamada de función invocada automáticamente devolverá el objeto recién construido.

Los pasos 1, 3 y 4 se aplican a nuestra discusión actual. Vamos a saltar el paso 2 por ahora y volver a él en el capítulo 5.

Considere este código:

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

Al llamar a `foo(...)` con un `new` delante de él, hemos construido un nuevo objeto y establecemos ese nuevo objeto como el de la llamada de `foo(..)`. Por tanto `new` es la manera final en que una llamada a la función `this` puede ser vinculada. Llamaremos a esto nueva vinculación (new banding).

2.3 Todo en orden

Por lo tanto, ahora hemos descubierto las 4 reglas para vincular `this` en las llamadas de función. Todo lo que necesita hacer es encontrar el sitio de llamada e inspeccionarlo para ver qué regla se aplica. Pero, ¿qué pasa si el sitio de llamada tiene múltiples reglas elegibles? Debe haber un orden de precedencia a estas reglas, por lo que demostraremos a continuación cuál es el orden para aplicar las reglas.

Debe quedar claro que la vinculación por defecto es la regla de prioridad más baja de las 4. Por lo tanto, vamos a dejarla de lado.

¿Cuál es más precedente, vinculación implícita o vinculación explícita? Vamos a probarlo:

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

Por lo tanto, la vinculación explicita tiene prioridad sobre la vinculacion implicita, lo que significa que debe preguntar primero si se aplica la vinculación explicita antes de comprobar la vinculación implícita.

Ahora, sólo necesitamos averiguar dónde se ajusta el nuevo vinculo en la precedencia.

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

OK, la nueva vinculación es más precedente que la vinculación implícita. Pero ¿crees que la nueva vinculación es más o menos precedente que la vinculación explícita?

Nota: `new` y `call` / `apply` no se pueden usar juntos, por lo que no se permite `new foo.call(obj1)` , para probar el nuevo vinculo directamente contra el vinculo explícito. Pero todavía podemos usar una vinculación dura (hard binding) para probar la precedencia de las dos reglas.

Antes de explorar eso en un listado de código, piense en cómo funciona físicamente el vinculo duro, que es `Function.prototype.bind(..)` crea una nueva función de encapsulamiento que está codificada para ignorar su propia vinculación (lo que sea), y utilizar un manual que proporcionamos.

Por ese razonamiento, parecería obvio asumir que la vinculación dura (que es una forma de vinculación explícita) es más precedente que una nueva vinculación, y por lo tanto no puede ser reemplazada por una nueva.

Vamos a revisar:

```
function foo(something) {
    this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

Whoa! `bar` está limitado contra `obj1`, pero `new bar(3)` no cambió `obj1.a` para ser `3` como habríamos esperado. En su lugar, la llamada a `bar(..)` con límite rígido (a `obj1`) puede ser reemplazada por `new`. Desde que se aplicó `new`, obtuvimos el objeto recién creado, que denominamos `baz`, y vemos de hecho que `baz.a` tiene el valor `3`.

Esto debería ser sorprendente si vuelves a nuestro "falso" `bind` helper:

```
function bind(fn, obj) {
    return function() {
        fn.apply( obj, arguments );
    };
}
```

Si razonas sobre cómo funciona el código del ayudante, no tiene una forma para que una llamada `new` de operador anule la vinculación de hard-obj como acabamos de observar.

Pero el built-in `Function.prototype.bind(...)` como ES5 es más sofisticado, un poco así, de hecho. Aquí está el polyfill (ligeramente reformateado) proporcionado por la página MDN para `bind(..)`:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // closest thing possible to the ECMAScript 5
            // internal IsCallable function
            throw new TypeError( "Function.prototype.bind - what " +
                "is trying to be bound is not callable"
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat( Array.prototype.slice.call( arguments ) )
                );
            }
        ;

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
    };
}
```

Nota: El `bind()` polyfill mostrado anteriormente difiere del built-in `bind(..)` en ES5 con respecto a las funciones enlazadas que se usarán con `new` (vea más adelante por qué es útil). Debido a que el polyfill no puede crear una función sin un `.prototype` como lo hace la utilidad incorporada, hay alguna indirección matizada para aproximar el mismo comportamiento. Pise con cuidado si va a utilizar el `new` con una función de hard-bound y confía en este polyfill.

La parte que está permitiendo el `new` overriding es:

```
this instanceof fNOP &&
oThis ? this : oThis

// ... and:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

vamos aqui 2 de septiembre 2017