Los fundamentos de JavaScript que todo informático debería conocer

EL GRAN LIBRO DE HTML5, CSS3 Y JAVASCRIPT

Capítulo 4

Javascript

4.1 Breve Introducción a Javascript

HTML5 puede ser considerado como un edificio con tres columnas: HTML, CSS y Javascript. Ya hemos estudiado los elementos incorporados en HTML y las nuevas propiedades que convierten a CSS en una herramienta ideal para diseñadores. Es tiempo de abordar lo que puede ser considerado como uno de los pilares más importantes de esta especificación: Javascript.

Javascript es un lenguaje interpretado que puede ser utilizado para diversos propósitos, pero que había sido considerado hasta ahora como solo un complemento de HTML y CSS. Una de las innovaciones que ayudó a cambiar esta imagen fueron los nuevos motores de procesamiento incorporados por las recientes versiones de los navegadores más populares. Estos motores fueron desarrollados para acelerar el procesamiento del código Javascript convirtiéndolo en código máquina y así alcanzar velocidades de ejecución similares a las aplicaciones de escritorio. Esta nueva capacidad ayudó al lenguaje a superar limitaciones previas y confirmarlo como la mejor opción de codificación para la web.

Javascript necesita un ambiente donde ser ejecutado, pero gracias a los nuevos intérpretes y motores de procesamiento como V8 de Google, se ha convertido en un lenguaje más poderoso e independiente. Desde ahora, puede ser ejecutado casi en cualquier parte y sus nuevas características lo han hecho responsable de la revolución de Internet en estos últimos años así como el surgimiento del nuevo mercado de dispositivos móviles.

Para aprovechar esta prometedora arquitectura, Javascript ha sido mejorado con el objetivo de lograr mayor portabilidad e integración. Además, interfaces de programación de aplicaciones completas (APIs) han sido incorporadas por defecto en cada navegador para asistir al lenguaje en características fundamentales. Estas nuevas APIs (como Web Storage, Canvas y otras) son interfaces de librarías de código incluidas en los navegadores. La idea es hacer accesible en todo momento poderosas herramientas a través de técnicas simples y estándar de programación, expandiendo el alcance del lenguaje y facilitando la creación de poderosas aplicaciones para la web.

En este momento, Javascript es el lenguaje más popular y prometedor de todos, y HTML5 lo está convirtiendo en una herramienta esencial para desarrolladores de aplicaciones web y móviles. En este capítulo vamos a estudiar conceptos básicos de Javascript y cómo incorporar códigos en nuestros documentos HTML. También vamos a introducir recientes adiciones al lenguaje a modo de preparación para afrontar el resto del libro.

IMPORTANTE: Nuestra aproximación a Javascript en este libro es introductoria. Trabajamos con características básicas pero solo aplicamos los conceptos elementales necesarios para aprovechar las innovaciones introducidas en

HTML5. Para expandir su conocimiento del lenguaje, visite nuestro sitio web y siga los enlaces de este capítulo. Si usted está familiarizado con esta información, siéntase libre de saltear las partes que ya conoce.

El Lenguaje

Javascript es un lenguaje de programación, algo completamente diferente de HTML y CSS. HTML es un lenguaje de etiquetas, como un código críptico que los navegadores interpretan para organizar la información, y CSS puede ser considerado simplemente como una hoja de estilos (aunque la nueva especificación lo ha convertido en una herramienta más dinámica). Javascript, por otro lado, es un lenguaje de secuencia de comandos, muy similar a cualquier otro lenguaje profesional como C++ o Java. La única diferencia significativa entre la mayoría de los lenguajes de programación y Javascript es su naturaleza. Mientras que otros lenguajes son orientados a objetos, Javascript es un lenguaje basado en prototipos, lo que, paradójicamente, lo relaciona más con objetos que cualquier otro, como veremos pronto.

Javascript puede realizar numerosas tareas, desde proveer instrucciones simples hasta calcular complejos algoritmos, pero la característica más importante es, como en la mayoría de los lenguajes de programación, la capacidad de almacenar y procesar información, lo cual se logra a través de variables.

Variables

La memoria de un ordenador o dispositivo móvil es como un panal gigante con millones y millones de celdas disponibles para almacenar información. Esas celdas tienen una dirección, un número consecutivo que identifica a cada una. Las celdas tienen un espacio limitado, y generalmente es necesaria la combinación de varias de ellas para almacenar grandes cantidades de datos. Debido a la complejidad del proceso de manipulación de este espacio de almacenamiento, los lenguajes de programación incorporaron el concepto de variables para facilitar la identificación de cada valor almacenado en memoria.

Las variables son solo nombres asignados a una celda o un grupo de celdas donde los datos serán almacenados. Por ejemplo, si queremos almacenar el valor 5 en memoria, necesitamos saber dónde el número fue almacenado para poder leerlo luego. Crear una variable nos permite identificar ese espacio en particular por un nombre y recuperar el contenido de esa celda usando el nombre correspondiente.

Para declarar una variable, Javascript usa la palabra clave **`var`**.

```
<script>
  var minumero = 2;
</script>
```

Listado 4-1: *Declarando una variable en Javascript*

Conceptos Básicos: Las etiquetas **`<script>`** son usadas para informar al navegador que el código fuente entre ellas fue escrito en Javascript. Esta es una de varias posibilidades con las que contamos para incorporar código Javascript en un documento HTML. Estudiaremos otras alternativas pronto.

En el Listado 4-1, creamos la variable **minumero** y almacenamos el valor **2** en el espacio de memoria que representa. Este proceso es normalmente llamado **asignar**. Bajo estos términos, lo que hacemos es "asignar el valor **2** a la variable **minumero**".

Javascript reserva un espacio en memoria, almacena el número **2**, crea una referencia a ese espacio y le asigna el nombre **minumero**. De este modo, cada vez que usamos esa referencia (el nombre **minumero**), obtenemos el número **2.**

```
<script>
  var minumero = 2;
  alert(minumero);
</script>
```

Listado 4-2: *Usando el contenido de una variable*

En el nuevo ejemplo del Listado 4-2 creamos la variable y le asignamos nuevamente el valor **2**. Desde ese momento el contenido de la variable **minumero** es **2**. Usando el método **alert()**, el contenido de esta variable es mostrado en pantalla.

Conceptos Básicos: El método **alert()** es un método tradicional de Javascript. Es usado para generar una ventana emergente con la que mostrar información en pantalla. El método mostrará en la ventana cualquier valor declarado entre paréntesis, desde textos hasta contenidos de variables, como en el código del Listado 4-2 (por ejemplo, **alert("Hola Mundo");**). Más adelante, estudiaremos con mayor profundidad éste y otros métodos.

Hágalo Usted Mismo: Si nuestro propósito es solamente probar el código Javascript, no necesitamos crear la estructura HTML completa. El código fuente del Listado 4-2 es más que suficiente para que los navegadores puedan interpretar los ejemplos correctamente. Copie este código fuente en una archivo de texto vacío, grábelo con el nombre de su preferencia y la extensión**.html** (por ejemplo, **jscodigo.html**), y abra el archivo en su navegador. El navegador generará de inmediato una ventana conteniendo el número **2**.

Las variables son llamadas *variables* porque no son constantes. Podemos cambiar el valor que les asignamos cada vez que lo necesitemos. Esa es, de hecho, su característica más importante.

```
<script>
  var minumero = 2;
  minumero = 3;
  alert(minumero);
</script>
```

Listado 4-3: *Asignando un nuevo valor para la variable*

En el Listado 4-3, luego de la creación de la variable **minumero**, accedemos nuevamente a la misma para asignarle otro valor. Ahora el método **alert()** mostrará el número **3**. En esta segunda asignación no necesitamos utilizar más la palabra clave **var**. La variable ya había sido creada, por lo que su nombre y el signo = (igual) son suficientes para asignar el nuevo valor.

En una situación más práctica, probablemente usaríamos el valor inicial de la variable para realizar alguna operación y luego asignaríamos el resultado a la misma variable:

```
<script>
  var minumero = 2;
  minumero = minumero + 1;
  alert(minumero);
</script>
```

Listado 4-4: *Usando el valor almacenado en la variable*

En este ejemplo el valor **1** es sumado al valor actual de **minumero** y el resultado es asignado a la misma variable. Esto es lo mismo que sumar 2 + 1, con la diferencia de que al usar una variable en lugar de un número su valor puede cambiar constantemente. Esto es debido a que durante la ejecución del código, los valores de las variables pueden ser modificados. Podríamos haber tenido el número **5** en lugar del **2** almacenado en la variable como resultado de un procesamiento previo y entonces el resultado de la suma hubiera sido **6** en lugar de **3**.

Esto nos ofrece una perspectiva del poder de las variables. Podemos realizar cualquier operación matemática o concatenar texto, no importa, las variables siempre mantendrán el último valor que les asignamos.

Conceptos Básicos: Además del operador **+**, hay numerosos operadores disponibles en Javascript. Los más comunes son **+** (suma), **–** (resta), ***** (multiplicación) y **/** (división), pero el lenguaje también ofrece operadores lógicos, como **&&** (y) o **||** (o), operadores de comparación, como **==** (igual), **!=** (diferente), **<** (menor que), **>** (mayor que), y otros. Estudiaremos y aplicaremos varios de ellos a lo largo del libro. Para obtener una lista completa, visite nuestro sitio web y siga los enlaces de este capítulo.

Las variables pueden almacenar números, textos, valores predefinidos, como valores Booleanos (verdadero o falso), o valores elementales como **null** (nulo) o **undefined** (indefinido). Los números son expresados como lo hicimos en anteriores ejemplos, pero el texto tiene que ser encerrado entre comillas dobles o simples.

```
<script>
  var mitexto = "Hola Mundo!";
  alert(mitexto);
</script>
```

Listado 4-5: *Asignando un texto a una variable*

Las variables pueden también almacenar varios valores al mismo tiempo en una estructura conocida como **matriz** (array). Las matrices pueden ser descriptas como variables multidimensionales, y pueden ser creadas usando una notación simple que incluye corchetes para agrupar los valores y comas para diferenciarlos. Los valores de una matriz son luego identificados con un índice que comienza por el valor 0.

```
<script>
  var mimatriz = ["rojo", "verde", "azul"];
  alert(mimatriz[0]);
</script>
```

Listado 4-6: *Creando variables multidimensionales*

En el Listado 4-6 creamos una matriz llamada **mimatriz** conteniendo tres valores en forma de texto: "rojo", "verde" y "azul". Javascript asigna automáticamente el índice **0** para el primer valor, **1** para el segundo y **2** para el tercero. Para leer esta información, cada vez que queremos acceder a los valores de la matriz tenemos que mencionar su índice. Esto se logra encerrando el número de índice entre corchetes luego del nombre de la variable. Por ejemplo, para obtener el primer valor de **mimatriz** tenemos que escribir **mimatriz[0]**, como hicimos en nuestro ejemplo.

Las matrices, al igual que las variables, pueden tomar cualquier tipo de valor. Podemos crear una matriz como la del Listado 4-6 usando números, textos, valores Booleanos o incluso declarar valores como **null** o **undefined**.

```
<script>
  var mimatriz = ["rojo", , 32, null, "HTML5 es genial!"];
  alert(mimatriz[1]);
</script>
```

Listado 4-7: *Usando diferentes tipos de valores*

Conceptos Básicos: Los valores **null** y **undefined** son valores nativos de Javascript. Básicamente, ambos representan la ausencia de un valor, pero **undefined** también es retornado cuando una propiedad de un objeto o el elemento de una matriz no fueron definidos. En el ejemplo del Listado 4-7, el segundo elemento de la matriz no fue definido. Si intentamos leer este elemento, el valor retornado es **undefined**.

Hágalo Usted Mismo: Copie el código del Listado 4-7 en un archivo de texto vacío, grábelo con el nombre de su preferencia y la extensión **.html** (por ejemplo, **jscodigo.html**), y abra el archivo en su navegador. Cambie el índice en el método **alert()** para mostrar cada uno de los valores de la matriz.

Por supuesto, también podemos realizar operaciones con los valores de una matriz y almacenar el resultado, como hicimos antes con variables simples.

```
<script>
  var mimatriz = ["rojo", 32];
  alert(mimatriz[0]);
  mimatriz[0] = "color " + mimatriz[0];
  mimatriz[1] = mimatriz[1] + 10;
  alert(mimatriz[0] + " " + mimatriz[1]);
</script>
```

Listado 4-8: *Trabajando con matrices*

Con el código del Listado 4-8 podemos aprender no solo cómo modificar el valor de una matriz sino también cómo concatenar texto. Matrices trabajan del mismo modo que las variables simples, con la excepción de que tenemos que mencionar el índice cada vez que queremos usarlas. Por medio de la línea **mimatriz[1] = mimatriz[1] + 10** le ordenamos al intérprete de Javascript que lea el valor actual de **mimatriz** en el índice **1** (32), sume 10 a ese valor y almacene el resultado (42) en la misma matriz y en el mismo índice; por lo que ahora el valor de **mimatriz[1]** será 42.

El operador **+** trabaja con números y textos, pero en el caso de textos lo que hace es concatenar los valores y generar un nuevo texto que incluye a ambos. En nuestro ejemplo, el

valor de **mimatriz[0]** era "rojo", pero luego concatenamos este valor con el texto "color " y asignamos el resultado nuevamente a **mimatriz[0]**; su nuevo valor será "color rojo".

Conceptos Básicos: Al final del último código, el método **alert()** muestra los dos valores de la matriz concatenados con un espacio. El operador **+** puede ser usado para realizar operaciones con el objetivo de obtener nuevos valores que puedan ser procesados y almacenados o con propósitos de diseño (cuando el proceso incluye números y texto, los números son convertidos en texto automáticamente).

Hasta el momento hemos modificado los valores ya existentes en una matriz. Javascript ofrece una forma simple de interactuar con matrices a un nivel más profundo. Usando los siguientes métodos, podemos extraer o agregar nuevos valores:

push(valor)—Este método agrega un nuevo valor al final de la matriz. El atributo **valor** es el valor a ser agregado.

shift()—Este método remueve el primer valor de la matriz y lo retorna para ser procesado.

pop()—Este método remueve el último valor de la matriz y lo retorna para ser procesado.

```
<script>
  var mimatriz = ["rojo", 32];
  mimatriz.push('coche');
  alert(mimatriz[2]);
  alert(mimatriz.shift());
</script>
```

Listado 4-9: *Agregando y extrayendo valores de una matriz*

El valor agregado por el método **push()** es asignado al siguiente índice disponible en la matriz. En el Listado 4-9, el valor insertado por **push()** recibe el índice **2**. En consecuencia, usando este índice, podemos mostrar el valor en pantalla.

Al final del código, extraemos y mostramos en pantalla el primer valor de la matriz. Los valores extraídos por los métodos **shift()** y **pop()** ya no son parte de la matriz. Luego de ejecutar el código de este ejemplo, la matriz contendrá solo dos valores: **32** y **coche**.

Condicionales y Bucles

Un programa de computación no es un programa si no puede establecer condiciones y generar complejos ciclos de procesamiento. Javascript ofrece un total de cuatro declaraciones con el objeto de ejecutar código de acuerdo a condiciones establecidas por el programador: **if**, **switch**, **for** y **while**.

```
<script>
  var mivariable = 9;
  if(mivariable < 10) {
    alert("El valor es menor que 10");
  }
</script>
```

Listado 4-10: *Comprobando una condición con* `if`

Conceptos Básicos: En Javascript es recomendable cerrar siempre las líneas de código con un punto y coma, pero esto no es necesario luego de un bloque de declaraciones (un bloque de código encerrado en llaves).

La declaración **`if`** controla la expresión entre paréntesis y procesa las instrucciones entre llaves si la condición es verdadera. En el código del Listado 4-10, el valor 9 es asignado a **`mivariable,`** y luego, usando **`if`** comparamos el valor de esta variable con el número **`10`**. Si el valor actual de la variable es menor que **`10`**, el método **`alert()`** es ejecutado para mostrar el mensaje en pantalla.

Por medio de una adición simple podemos ejecutar una acción en caso de que la condición sea verdadera y también cuando es falsa. La construcción **`if else`** controlará la condición entre paréntesis y ejecutar una acción diferente en cada caso.

```
<script>
  var mivariable = 9;
  if(mivariable < 10) {
    alert("El valor es menor que 10");
  }else{
    alert("El valor es igual que 10 o mayor");
  }
</script>
```

Listado 4-11: *Comprobando dos condiciones con* `if else`

En este ejemplo el código comprueba la condición entre paréntesis. Si es verdadera ejecuta el primer grupo de instrucciones y si es falsa el segundo (dentro del bloque **`else`**). En consecuencia, la declaración está comprobando dos condiciones: cuando el número es menor que 10 y cuando es igual o más grande que **`10`**. Para comprobar múltiples condiciones, Javascript incluye la declaración **`switch`** .

```
<script>
  var mivariable = 9;
  switch(mivariable) {
    case 5:
      alert("El valor es cinco");
      break;
    case 8:
      alert("El valor es ocho");
      break;
    case 10:
      alert("El valor es diez");
      break;
    default:
      alert("El valor es " + mivariable);
  }
</script>
```

Listado 4-12: *Usando la declaración* `switch`

La declaración **`switch`** evalúa una expresión (generalmente una variable simple), compara el resultado con las declaraciones **`case`** dentro de las llaves y, en caso de éxito, ejecuta las instrucciones declaradas para ese caso. En el ejemplo del Listado 4-12, **`switch`** lee la variable **`mivariable`** para obtener su valor. Luego, compara este valor con cada caso (**`case`**). Si el valor

es **5**, por ejemplo, el control será transferido al primer caso y el método **alert()** mostrará en la pantalla el texto "el número es cinco". Si el valor de este primer caso no concuerda con el valor de la variable, el segundo caso será evaluado, y así sucesivamente. Si ningún caso coincide con el valor de la variable, las instrucciones que se ejecutan son las que se encuentran en el caso llamado **default** (por defecto).

El código dentro de cada caso debe finalizar con la instrucción **break**. Esta instrucción informa al intérprete Javascript que no hay necesidad de continuar evaluando el resto de los casos.

Las declaraciones **switch** y **if** son útiles pero realizan una tarea simple: evalúan una expresión, ejecutan un grupo de instrucciones de acuerdo al resultado y al final retornan el control al código siguiente. En algunas situaciones, esto no será suficiente. Podríamos, por ejemplo, necesitar ejecutar un grupo de instrucciones varias veces para la misma condición, o evaluar una expresión una vez más luego de que un cambio se produce durante el proceso. Javascript ofrece dos declaraciones para estos casos: **for** y **while**.

```
<script>
  var mivariable = 9;
  for(var f = 0; f < mivariable; f++) {
    alert("El valor actual es " + f);
  }
</script>
```

Listado 4-13: *Usando la declaración* `for`

La declaración **for** ejecutará el código entre llaves mientras la condición declarada en el segundo parámetro sea verdadera. Utiliza la sintaxis **for(inicialización; condición; incremento)**. El primer parámetro establece los valores iniciales para el bucle, el segundo parámetro es la condición a ser comprobada, y el tercer parámetro es una instrucción que determinará cómo los valores iniciales evolucionarán en cada ciclo.

En el código del Listado 4-13, declaramos una variable llamada **f** y le asignamos el valor **0** para utilizarlo como valor de inicialización. La condición en este ejemplo determina si el valor de la variable **f** es menor que el de la variable **mivariable**. En caso de que la condición sea verdadera, el código entre llaves es ejecutado. Al finalizar, el intérprete Javascript ejecuta el tercer parámetro de **for** y comprueba la condición nuevamente. Si la condición es verdadera, las instrucciones son ejecutadas una vez más. El bucle continúa hasta que la condición es falsa.

La variable **f** es inicializada con el valor **0**. Luego de que el primer ciclo es completado, la instrucción **f++** incrementa el valor de **f** en una unidad (1), y la condición es comprobada nuevamente. En cada ciclo, el valor de **f** es incrementado una unidad. Este proceso continúa hasta que el valor de **f** es **9**, en cuyo caso la condición será falsa (9 no es menor que 9), y el bucle es interrumpido.

Conceptos Básicos: El operador **++** es otro operador Javascript para suma. Lo que hace es sumar el valor 1 al valor actual de la variable y almacenar el resultado en la misma variable. Es una forma de abreviar la operación **variable = variable + 1**.

La declaración **for** es útil cuando podemos predeterminar ciertas condiciones, como el valor inicial para el bucle o cómo ese valor evolucionará. Cuando las condiciones no son claras, podemos aplicar la declaración **while** .

```
<script>
  var mivariable = 9;
  while(mivariable < 100) {
    mivariable++;
  }
  alert("El valor final de mivariable es " + mivariable);
</script>
```

Listado 4-14: *Usando la declaración* `while`

La declaración **`while`** solo requiere la declaración de la condición entre paréntesis y el código a ser ejecutado entre llaves. El bucle se ejecutará hasta que la condición sea falsa. Si la primera evaluación de la condición resulta falsa, el código nunca es ejecutado. Si lo que necesitamos es que las instrucciones sean ejecutadas al menos una vez, podemos usar la declaración adicional **`do while`**.

```
<script>
  var mivariable = 9;
  do{
    mivariable++;
  }while(mivariable > 100);
  alert("El valor final de mivariable es " + mivariable);
</script>
```

Listado 4-15: *Usando la declaración* `do while`

En el código del Listado 4-15 cambiamos la condición a "**`mivariable`** mayor que 100". Como el valor inicial asignado a la variable fue 9, esta condición es falsa, pero como estamos usando la declaración **`do while`**, el código dentro del bucle es ejecutado una vez antes de que la condición sea evaluada. Al final, la ventana emergente mostrará el valor **`10`** (9 + 1 = 10).

Funciones y Funciones Anónimas

Hasta el momento hemos trabajado en el espacio global. Este es el espacio principal donde el código es procesado al momento de ser leído por el intérprete. Las instrucciones son ejecutadas secuencialmente en el orden en el que fueron listadas dentro del código, pero podemos especificar un flujo diferente usando funciones.

Básicamente, una función es un bloque de código identificado por un nombre. Varias son las ventajas al declarar el código dentro de una función en lugar de hacerlo en el espacio global. La más importante es que las funciones son declaradas pero no ejecutadas hasta que las llamamos por su nombre.

```
<script>
  function mifuncion(){
    alert("Soy una función");
  }
  mifuncion();
</script>
```

Listado 4-16: *Declarando funciones*

Conceptos Básicos: Las comillas en Javascript son intercambiables. Un texto puede ser declarado usando comillas dobles o simples (por ejemplo, **"Hola"** o **'Hola'**). Si un texto incluye en su interior el mismo tipo de comillas que usamos para declararlo, debemos marcarlas como carácter especial usando una barra invertida (por ejemplo, **'Ella dijo \'hola\''**).

Las funciones son declaradas usando la palabra clave **function**, un nombre, y el código entre llaves. Para llamar a una función (ejecutarla), solo tenemos que usar su nombre con un par de paréntesis al final, como hicimos en el Listado 4-16.

Del mismo modo que con las variables, el intérprete Javascript lee la función, almacena su contenido en memoria y asigna una referencia a su nombre. Cuando llamamos la función, el intérprete utiliza esta referencia para leer la función en memoria. Esto nos permite llamar a la función una y otra vez siempre que la necesitemos. Esta característica convierte a las funciones en poderosas unidades de procesamiento.

```
<script>
  var mivariable = 5;
  function mifuncion(){
    mivariable = mivariable * 2;
  }
  for(var f = 0; f < 10; f++){
    mifuncion();
  }
  alert("El valor de mivariable es " + mivariable);
</script>
```

Listado 4-17: *Procesando datos con funciones*

El ejemplo del Listado 4-17 combina varias declaraciones ya estudiadas. En primer lugar, declaramos una variable y le asignamos el valor **5**. Luego, la función **mifuncion()** es declarada (pero no ejecutada). En el siguiente paso usamos una declaración **for** para crear un bucle que funcionará mientras el valor de la variable **f** sea menor a 10. Finalmente, mostramos en pantalla el último valor almacenado en la variable **mivariable**.

Todo lo que hace el código dentro del bucle **for** es llamar a la función, por lo que **mifuncion()** será ejecutada en cada ciclo. Esta función multiplica el valor de la variable **mivariable** por **2** y almacena el resultado en la misma variable. Cada vez que la función es ejecutada, el valor de la variable **mivariable** es incrementado. Mediante este ejemplo podemos ver el poder de las funciones como unidades de procesamiento: las declaramos una vez, les damos un nombre como referencia, y luego podemos usarlas cada vez que las necesitamos.

Otra forma de declarar una función es usar una función anónima y asignarla a una variable. El nombre de la variable será el nombre que usaremos luego para llamar a la función. Esta técnica es usada para definir funciones dentro de otras funciones, convertir funciones en constructores de objetos y crear patrones complejos de programación, como veremos más adelante.

```
<script>
  var mivariable = 5;
  var mifuncion = function(){
    mivariable = mivariable * 2;
  };
```

```
  for(var f = 0; f < 10; f++){
    mifuncion();
  }
  alert("El valor de mivariable es " + mivariable);
</script>
```

Listado 4-18: *Declarando funciones con funciones anónimas*

Funciones anónimas son simplemente funciones sin un nombre o identificador (por eso son llamadas "anónimas"). Podemos asignar estas funciones a variables, usarlas para crear métodos para objetos, pasarlas como argumento a otras funciones (o retornarlas desde otras funciones), etc. Este tipo de funciones son extremadamente útiles en Javascript. Patrones complejos de programación no serían posibles sin este tipo de construcciones.

IMPORTANTE: El estudio de funciones anónimas y patrones complejos de programación no forman parte del propósito de este libro, pero veremos algunos ejemplos de ambos en éste y próximos capítulos.

En ejemplos anteriores usamos una variable definida en el espacio global, pero las variables pueden tener diferentes alcances. Las que son definidas en el espacio global tienen un alcance global, lo que significa que pueden ser usadas en cualquier lugar del código. Por otro lado, las variables declaradas dentro de funciones tienen un alcance local, lo que significa que solo pueden ser usadas dentro de la función en donde fueron declaradas. Esta es otra ventaja de las funciones, son lugares especiales en el código donde podemos almacenar datos privados que no serán accesibles por otras partes del código. Esta segregación nos ayuda a evitar duplicados que podrían conducir a errores o a sobre escribir accidentalmente otras variables y valores. Dos variables con el mismo nombre, una en el espacio global y otra dentro de una función, serán consideradas como dos variables diferentes.

```
<script>
  var mivariable = 5;
  function mifuncion(){
    var mivariable = "Esta es una variable de la función";
    alert(mivariable);
  }
  mifuncion();
  alert(mivariable);
</script>
```

Listado 4-19: *Declarando variables globales y locales*

En el último ejemplo declaramos dos variables llamadas **`mivariable`**, una en el espacio global y otra dentro de la función **`mifuncion()`**. También incluimos en el código dos métodos **`alert()`**, uno dentro de la función y otro en el espacio global al final del código. Ambos métodos muestran el contenido de **`mivariable`**, pero, como verá si ejecuta el ejemplo en su navegador, estas son dos variables diferentes. La variable dentro de la función está referenciando un espacio en memoria que contiene el texto "Esta es una variable de la función", mientras que la variable en el espacio global (declarada en primer lugar) está referenciando un espacio en memoria que contiene el valor 5.

Las variables globales son útiles cuando varias funciones tienen que compartir valores comunes. Este tipo de variables son accesibles desde cada función en el código, pero como

resultado, existe la posibilidad de que sus valores sean borrados accidentalmente desde otras partes del código o incluso desde otros códigos (todos los archivos Javascript cargados en el documento comparten el mismo espacio global). Por esta razón, es recomendable evitar su uso cuando sea posible.

IMPORTANTE: Variables globales pueden también ser creadas desde el interior de las funciones. Omitir la palabra clave **`var`** al declarar una variable dentro de una función es todo lo que necesitamos para declarar esa variable como global.

Conceptos Básicos: Una práctica común para evitar el uso de variables globales es declarar un objeto global y luego crear nuestra aplicación dentro de este objeto. Estudiaremos objetos Javascript en la siguiente sección de este capítulo.

Podemos leer variables globales desde el interior de las funciones pero no variables locales desde el espacio global. Esta es la razón por la que Javascript ofrece diferentes alternativas de comunicación entre una parte y otra. Mediante estos mecanismos podemos enviar datos a una función a través de sus atributos o podemos retornar datos desde una función usando la instrucción **`return`**.

```
<script>
  var mivariable = 5;
  var miresultado;
  function mifuncion(contador){
    contador = contador + 12;
    return contador;
  }
  miresultado = mifuncion(mivariable);
  alert(miresultado);
</script>
```

Listado 4-20: *Comunicándose con funciones*

Los argumentos son variables que se definen entre los paréntesis de la función y pueden ser usados dentro de la función como lo haríamos con cualquier otra variable. Estas variables toman los valores que son enviados a la función cuando es llamada. En el código del Listado 4-20, la función **`mifuncion()`** es declarado con un argumento llamado **`contador`**. Más adelante, llamamos a la función con la variable **`mivariable`** entre paréntesis. Este es valor que la función recibirá y asignará a la variable **`contador`**. Cuando la función es llamada, el valor inicial de **`contador`** es **`5`** (porque ese es el valor inicial asignado a **`mivariable`**). Dentro de **`mifuncion()`**, sumamos **`12`** al **`contador`** y retornamos el resultado por medio de la instrucción **`return`**. De regreso en el espacio global, el valor retornado por la función (**`17`**) es asignado a la variable **`miresultado`**, y el contenido de esta variable es mostrado en pantalla.

Al inicio declaramos la variable **`miresultado`** pero no le asignamos ningún valor. Esto es perfectamente aceptable e incluso recomendable. Es buena práctica el declarar todas las variables con las que vamos a trabajar al comienzo, y así evitar confusión y poder identificarlas fácilmente desde otras partes del código.

IMPORTANTE: Más adelante estudiaremos otros aspectos de funciones y practicaremos cada una de las técnicas presentadas en este capítulo.

Objetos

Los objetos son como grandes variables capaces de contener otras variables (llamadas **propiedades**) así como funciones (llamadas **métodos**). Para declarar objetos, podemos optar por varias alternativas, aunque lo más simple es usar notación literal.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30
  };
</script>
```

Listado 4-21: *Creando objetos*

El objeto es declarado usando la palabra clave **`var`** y unas llaves para agrupar su contenido. Las propiedades y los métodos requieren un nombre y son declarados usando dos puntos luego del nombre y una coma para separar cada declaración (la última declaración no requiere coma).

En el ejemplo del Listado 4-21 declaramos el objeto **`miobjeto`** y sus dos propiedades: **`nombre`** y **`edad`**. El valor de la propiedad **`nombre`** es **`Juan`** y el valor de la propiedad **`edad`** es **`30`**. Al contrario de las variables, no podemos acceder a los valores de las propiedades usando directamente su nombre. Antes del nombre debemos especificar el objeto al que pertenecen, para esto contamos con dos tipos de notación: usando un punto o corchetes.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30
  };
  alert("Mi nombre es " + miobjeto.nombre);
  alert("Yo tengo " + miobjeto['edad'] + " años de edad");
</script>
```

Listado 4-22: *Accediendo propiedades*

En el Listado 4-22 usamos ambas notaciones para acceder a las propiedades del objeto. Al final de la ejecución de este código, dos ventanas de alerta serán mostradas con los mensajes: "Mi nombre es Juan" y "Yo tengo 30 años de edad". Es irrelevante cuál notación utilizamos, excepto en algunas circunstancias. Por ejemplo, cuando necesitamos acceder a una propiedad por medio del valor de una variable tenemos que hacerlo usando corchetes.

```
<script>
  var mivariable = "nombre";
  var miobjeto = {
    nombre: "Juan",
    edad: 30
  };
  alert(miobjeto[mivariable]);
</script>
```

Listado 4-23: *Accediendo propiedades por medio de variables*

En este último ejemplo no podríamos haber accedido a la propiedad usando un punto (**miobjeto.mivariable**) porque el intérprete hubiera tratado de encontrar una propiedad llamada **mivariable** que no existe. Usando corchetes, el intérprete lee primero la variable y luego accede al objeto con el valor de la variable (**nombre**) en lugar de su nombre. El resultado mostrado en pantalla será "Juan".

También es necesario acceder a las propiedades de este modo cuando su nombre es considerado inválido para una variable (incluye caracteres inválidos, un espacio, o comienza con un número). En el siguiente ejemplo, el objeto incluye una propiedad declarada con un texto entre comillas. Está permitido declarar nombres de propiedades de este modo, pero debido a que el texto incluye un espacio, el código **miobjeto.mi edad** retornaría un error. Para acceder a esta propiedad debemos una vez más usar corchetes.

```
<script>
  var mivariable = "nombre";
  var miobjeto = {
    nombre: "Juan",
    'mi edad': 30
  };
  alert(miobjeto['mi edad']);
</script>
```

Listado 4-24: *Propiedades con nombres inválidos*

Además de leer los valores de las propiedades, podemos también crearlas o modificarlas usando ambos tipos de notación. En el siguiente ejemplo vamos a modificar el valor de la propiedad **nombre** y a agregar una nueva propiedad llamada **trabajo**. De este modo, el objeto tendrá tres propiedades: **nombre**, **edad** y **trabajo**. Al final del código, mostramos todos los valores en pantalla.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30
  };
  miobjeto.nombre = "Jorge";
  miobjeto.trabajo = "Programador";
  alert(miobjeto.nombre + " " + miobjeto.edad + " " + miobjeto.trabajo);
</script>
```

Listado 4-25: *Actualizando valores y agregando nuevas propiedades al objeto*

Como mencionamos anteriormente, los objetos pueden también incluir funciones. Las funciones dentro de un objeto son llamadas **métodos**. Métodos convierten a un objeto en una unidad capaz de contener y procesar información de forma independiente.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30,
    mostrarnombre: function(){
        alert(miobjeto.nombre);
    },
```

```
    cambiarnombre: function(nuevonombre){
      miobjeto.nombre = nuevonombre;
    }
  };
  miobjeto.mostrarnombre();
  miobjeto.cambiarnombre('Jorge');
  miobjeto.mostrarnombre();
</script>
```

Listado 4-26: *Incluyendo funciones dentro de objetos*

El código del Listado 4-26 muestra no solo cómo incluir métodos en un objeto sino también cómo usarlos. Los métodos tienen la misma sintaxis que las propiedades: requieren dos puntos luego del nombre y una coma para separar cada declaración. La diferencia con propiedades es que los métodos no toman valores directamente, son construidos usando funciones anónimas.

En este ejemplo creamos dos métodos: **mostrarnombre()** y **cambiarnombre()**. El método **mostrarnombre()** genera una ventana emergente con el valor de la propiedad **nombre**, mientras que el método **cambiarnombre()** asigna a la propiedad **nombre** el valor recibido por medio del atributo **nuevonombre**. Estos son dos métodos independientes que trabajan sobre la misma propiedad. Para ejecutarlos, usamos un punto y paréntesis luego del nombre. Al igual que las funciones, los métodos pueden tomar valores por medio de sus atributos y retornar valores usando la instrucción **return** (en este ejemplo, **cambiarnombre()** requiere un valor con el nuevo nombre).

Los objetos también pueden contener otros objetos. La estructura puede adquirir más y más complejidad de acuerdo a lo que necesitemos. Para acceder a las propiedades y métodos de estos objetos internos, debemos concatenar sus nombres usando puntos.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30,
    moto: {
      modelo: "Susuki",
      fecha: 1981
    },
    mostrarcoche: function(){
      alert(miobjeto.nombre + " posee una " + miobjeto.moto.modelo);
    }
  };
  miobjeto.mostrarcoche();
</script>
```

Listado 4-27: *Creando objetos dentro de objetos*

Como probablemente ya se habrá dado cuenta, los objetos son unidades de procesamiento poderosas e independientes. Su importancia es tan relevante en Javascript que el lenguaje no solo ofrece diferentes alternativas para crear nuestros propios objetos sino que además aporta los suyos propios. De hecho, prácticamente todo en Javascript es un objeto. Podemos usar objetos nativos, crear nuestros propios objetos desde cero, y modificarlos o expandirlos, pero una de las características más importantes es la posibilidad de crear nuevos objetos que heredan las propiedades y métodos de los objetos en los cuales se basan.

Herencia en Javascript (cómo objetos obtienen sus propiedades y métodos de otros objetos) se logra a través de prototipos. Cada objeto tiene un prototipo, y nuevos objetos son creados a partir de ese prototipo .

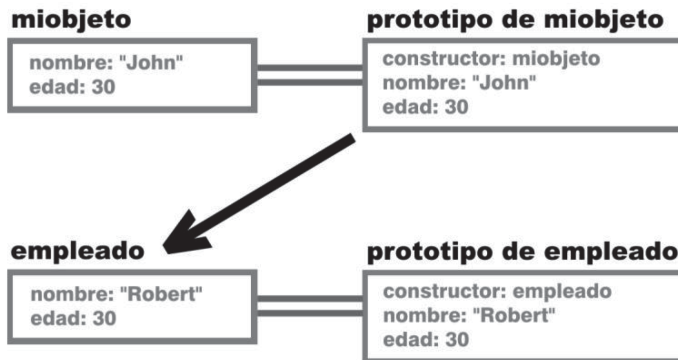

Figura 4-1: *Herencia de prototipos*

El objeto **`miobjeto`** creado en ejemplos anteriores obtuvo su propio prototipo automáticamente al ser creado. Si decidimos crear un nuevo objeto basado en **`miobjeto`**, el nuevo será creado desde el prototipo de **`miobjeto`**, no desde el objeto mismo. En la Figura 4-1, el objeto **`empleado`** es creado desde el prototipo de **`miobjeto`**. El nuevo objeto también obtiene su propio prototipo. Si más adelante nuevos objetos son creados desde **`empleado`**, estarán basados en el prototipo de **`empleado`**. Esta construcción puede seguir por siempre, creando una cadena de objetos llamada **cadena de prototipos**. Las modificaciones introducidas en un prototipo afectarán todos los objetos hacia abajo en la cadena. Por ejemplo, si introducimos cambios al prototipo de **`miobjeto`**, **`empleado`** y todos los objetos bajados en el prototipo de **`empleado`** heredarán estos cambios.

Las técnicas disponibles para lograr este tipo de herencia eran confusas e inconsistentes con las provistas por otros lenguajes de programación orientados a objetos, lo cual disuadió a algunos programadores a adoptar Javascript durante un tiempo. El nuevo método **`create()`** mejora esta situación. Este método es parte de un objeto nativo llamado precisamente Object. Lo que hace el método es usar un objeto existente como prototipo para crear uno nuevo, por lo que no tenemos que preocuparnos más sobre cómo trabajar con prototipos.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30,
    mostrarnombre: function(){
        alert(this.nombre);
    },

    cambiarnombre: function(nuevonombre){
      this.nombre = nuevonombre;
    }
  };
  var empleado = Object.create(miobjeto);
  empleado.cambiarnombre('Roberto');
  empleado.mostrarnombre();
  miobjeto.mostrarnombre();
</script>
```

Listado 4-28: *Generando nuevas instancias*

Algo importante para resaltar del Listado 4-28 es cómo referenciamos el objeto con el que estamos trabajando. Los objetos pueden referenciarse a sí mismos usando la palabra clave **this**. En este ejemplo usamos **this** en lugar del nombre del objeto (**miobjeto**) para poder acceder a la propiedad **nombre** desde dentro del objeto. Lo que hace la palabra clave es declarar algo como "este objeto" en lugar de proveer el nombre del objeto. La razón por la que usamos **this** es porque el objeto creado actúa como un plano para crear nuevos objetos (llamados instancias). Estas copias tendrán su propio nombre, el cuál no será el mismo que el del objeto original. El uso de **this** es necesario en estos casos para representar el nombre del objeto con el que estamos trabajando, sin importar si es el original o una de las copias. Por ejemplo, en el objeto **empleado**, creado por el código del Listado 4-28 a partir de **miobjeto**, la palabra clave **this** representa al objeto **empleado**, por lo que la construcción **this.nombre** en este objeto retornará el mismo valor que **empleado.nombre**.

El objeto **empleado** es creado usando el método **create()**. Este método solo requiere el nombre del objeto que va a actuar de prototipo para el nuevo. El método retorna un nuevo objeto que podemos asignar a una variable. En el Listado 4-28, la instancia es creada primero con **Object.create()** y es luego asignada a la variable **empleado**. Una vez que tenemos esta instancia, podemos actualizar sus valores. Usando el método **cambiarnombre()** cambiamos el nombre de **empleado** a "Roberto" y luego mostramos en la pantalla el valor de cada una de las propiedades **nombre**.

IMPORTANTE: El método **create()** es un nuevo método introducido en las versiones más recientes del lenguaje. Para simular este método en navegadores viejos, Douglas Crockford, en su sitio web (www.crockford.com) y también en su libro "Javascript: The Good Parts" (publicado por O'Reilly Media/Yahoo Press), recomienda el uso del siguiente código:

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

En el último código tenemos dos objetos individuales, **miobjeto** y **empleado**, con sus propias propiedades, métodos y valores, pero conectados a través de la cadena de prototipos. El nuevo objeto **empleado** no es solo una copia, es una instancia, un objeto que mantiene una conexión con su prototipo. Cuando el prototipo cambia, sus instancias heredan estos cambios.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30,
    mostrarnombre: function(){
        alert(this.nombre);
    },
    cambiarnombre: function(nuevonombre){
      this.nombre = nuevonombre;
    }
  };
  var empleado = Object.create(miobjeto);
  empleado.edad = 24;
```

```
  miobjeto.mostraredad = function(){
    alert(this.edad);
  };

  empleado.mostraredad();
</script>
```

Listado 4-29: *Agregando un nuevo método a un prototipo*

En el Listado 4-29, el objeto **empleado** es creado usando **miobjeto** como prototipo, del mismo modo que lo hemos hecho anteriormente. Luego de esto, la propiedad **edad** de este nuevo objeto es actualizada con el valor 24. Al final, agregamos al objeto que actúa como prototipo (**miobjeto**) un método llamado **mostraredad()**. Debido a la cadena de prototipos, este nuevo método es accesible también desde las instancias creadas por medio de este prototipo. Cuando llamamos al método **mostraredad()** de **empleado** al final del código, el intérprete busca el método dentro del objeto **empleado** y continúa buscando en la cadena de prototipos hasta que lo encuentra en **miobjeto**. Luego de que el método es finalmente encontrado y ejecutado, el valor 24 es mostrado en pantalla. Esto se debe a que, a pesar de que el método **mostraredad()** pertenece a **miobjeto**, la palabra clave **this** en este método apunta al objeto con el que estamos trabajando (**empleado**). Debido a la cadena de prototipos, el método **mostraredad()** puede ser invocado desde **empleado**, y debido a la palabra clave **this**, la propiedad **edad** mostrada en la pantalla es la que pertenece a **empleado**.

El método **create()** es tan simple como poderoso. Toma un objeto y lo convierte en el prototipo de uno nuevo. Esto nos permite construir una cadena de objetos en la que cada uno hereda propiedades y métodos de su predecesor. En el siguiente ejemplo, vamos a demostrar esta característica creando un total de cuatro objetos. El objeto inicial **miobjeto** es el prototipo de **empleado1**, pero luego usamos **empleado1** como el prototipo de **empleado2** y **empleado2** como el prototipo de **empleado3**. Estos son todos objetos independientes pero conectados entre sí por la cadena de prototipos. Cuando nuevos métodos y propiedades son agregados a uno de los objetos (un prototipo), el resto de los objetos a continuación en la cadena también tendrán acceso a los mismos.

```
<script>
  var miobjeto = {
    nombre: "Juan",
    edad: 30,
    mostrarnombre: function(){
        alert(this.nombre);
    },
    cambiarnombre: function(nuevonombre){
      this.nombre = nuevonombre;
    }
  };
  var empleado1 = Object.create(miobjeto);
  var empleado2 = Object.create(empleado1);
  var empleado3 = Object.create(empleado2);

  empleado2.mostraredad = function(){
    alert(this.edad);
  };
  empleado3.edad = 24;
  empleado3.mostraredad();
</script>
```

Listado 4-30: *Probando la cadena de prototipos*

En el Listado 4-30, la cadena de prototipos es demostrada agregando el método **mostraredad()** a **empleado2**. Ahora, **empleado2** y **empleado3** (y también cualquier otro objeto creado luego a partir de cualquiera de estos dos objetos) tendrán acceso a este método, pero si intentamos llamar al método desde **empleado1** no funcionará. Esto es debido a que el proceso de herencia afecta los objetos como en la vida real; hacia abajo de la cadena, no hacia arriba.

A pesar de la simpleza de **Object.create()**, a veces necesita ser complementado con otros patrones más complejos y funciones auxiliares para activar determinadas características en nuestro código, como creación dinámica de objetos, inicialización de valores o creación de propiedades y métodos privados.

Dependiendo de nuestras necesidades, puede resultar una ventaja crear nuestros objetos con una técnica alternativa que combina notación literal, funciones anónimas y la instrucción **return**.

```
<script>
  var constructor = function(nombreinicial){
    var obj = {
      nombre: nombreinicial,
      edad: 30,
      mostrarnombre: function(){
          alert(this.nombre);
      },
      cambiarnombre: function(nuevonombre){
        this.nombre = nuevonombre;
      }
    };
    return obj;
  };
  empleado = constructor("Juan");
  empleado.mostrarnombre();
</script>
```

***Listado 4-31:** Usando funciones anónimas para crear objetos*

El en ejemplo del Listado 4-31 usamos una función anónima para crear un objeto. La función anónima es asignada a la variable **constructor** y a través de esta variable llamamos a la función con el valor inicial "**Juan**". La función asigna este valor a su atributo **nombreinicial** y luego le asigna el valor de este atributo a la propiedad **nombre**. Finalmente, el objeto llamado **obj** es retornado por la instrucción **return** y es asignado a la variable **empleado**.

Usando esta técnica podemos crear nuevos objetos de forma dinámica, asignar valores iniciales o incluso realizar algunas operaciones al comienzo de la función y retornar un objeto creado a partir de este procesamiento.

Como podemos apreciar, esto es mucho más que simplemente crear y duplicar objetos. La ventaja más importante que ofrece esta técnica es la posibilidad de definir propiedades y métodos privados. Todos los objetos e instancias que hemos creado hasta ahora contienen propiedades y métodos públicos, lo cual significa que el contenido de los objetos puede ser accedido y modificado desde cualquier parte del código. Para evitar esto y permitir el acceso a las propiedades y métodos solo para el objeto que los creó, debemos volverlos privados usando una técnica llamada **closure** (cierre).

Como explicamos anteriormente, las variables creadas en el espacio global son accesibles desde cualquier parte del código, mientras que las variables creadas dentro de las funciones son solo accesibles desde las mismas funciones en las que fueron declaradas. Lo que no

mencionamos es que las funciones, y en consecuencia los métodos, siempre mantienen una conexión con el entorno en donde fueron creadas, y siempre mantienen una conexión con sus variables. Cuando retornamos un objeto desde una función, sus métodos podrán acceder las variables de la función, incluso cuando ya no se encuentran en el mismo entorno. Veamos un ejemplo:

```
<script>
  var constructor = function(){
    var nombre = "Juan";
    var edad = 30;
    var obj = {
      mostrarnombre: function(){
          alert(nombre);
      },
      cambiarnombre: function(nuevonombre){
        nombre = nuevonombre;
      }
    };
    return obj;
  };
  empleado = constructor();
  empleado.mostrarnombre();
</script>
```

Listado 4-32: *Creando propiedades privadas para un objeto*

El código del Listado 4-32 es exactamente el mismo usado en ejemplos previos excepto que en lugar de declarar **nombre** y **edad** como propiedades del objeto los declaramos como variables de la función que está retornando el objeto. El objeto retornado recordará estas variables, pero ese objeto será el único que tendrá acceso a ellas. No hay forma de acceder a estos valores desde otras partes del código, solo podemos hacerlo por medio de los métodos del objeto retornado (en este caso, **mostrarnombre()** y **cambiarnombre()**). Esto es llamado **closure**. La función es cerrada y su entorno ya no puede ser accedido, pero mantenemos un elemento conectado a este entorno (un objeto en nuestro ejemplo).

Por supuesto, cada nueva instancia creada a partir de esta función tendrá la misma estructura y tendrá acceso a sus propias propiedades privadas.

Hágalo Usted Mismo: Cree nuevos objetos a partir de la función del Listado 4-32 y use los métodos **cambiarnombre()** y **mostrarnombre()** para acceder a sus propiedades.

Constructores

Las funciones que usamos en los ejemplos anteriores para crear objetos son llamadas **constructores**. Javascript incluye una clase especial de constructor que trabaja con el operador **new** para obtener un nuevo objeto. Este método no es parte de la programación de prototipos pero fue introducido en Javascript para ser consistente con otros lenguajes orientados a objetos (especialmente Java).

Esta clase de constructores no crean un objeto, son como planos a partir de los cuales los objetos son creados. Por este motivo, el uso de la palabra clave **this** es requerido para declarar o acceder a sus propiedades y métodos.

```
<script>
  function Miobjeto(){
    this.nombre = "Juan";
    this.edad = 30;
    this.mostrarnombre = function(){
        alert(this.nombre);
    };
    this.cambiarnombre = function(nuevonombre){
      this.nombre = nuevonombre;
    };
  }
  var empleado = new Miobjeto();
  empleado.mostrarnombre();
</script>
```

Listado 4-33: *Usando constructores*

Excepto por el uso de **`this`**, constructores y funciones son muy similares. Para evitar confusión, es recomendable escribir su nombre con la letra inicial en mayúsculas, como lo hicimos en el ejemplo del Listado 4-33.

Para crear un objeto a partir de un constructor, tenemos que usar el operador **`new`**. Este operador toma el nombre del constructor y retorna un objeto basado en el mismo.

Un constructor puede incluir atributos con los cuales proveer valores iniciales para el nuevo objeto.

```
<script>
  function Miobjeto(nombreinicial, edadinicial){
    this.nombre = nombreinicial;
    this.edad = edadinicial;
    this.mostrarnombre = function(){
        alert(this.nombre);
    };
    this.cambiarnombre = function(nuevonombre){
      this.nombre = nuevonombre;
    };
  }
  var empleado = new Miobjeto("Roberto", 55);
  empleado.mostrarnombre();
</script>
```

Listado 4-34: *Definiendo valores iniciales para el objeto*

Este constructor no convierte a un objeto en prototipo de otro, como lo hace el método **`create()`**. En su lugar, crea nuevas instancias a partir del prototipo del constructor (**`Miobjeto.prototype`**). Debido a esto, si queremos introducir modificaciones para afectar esas instancias no podemos trabajar directamente con el objeto inicial, como lo hicimos antes, debemos acceder al prototipo del constructor usando la propiedad **`prototype`** .

```
<script>
  function Miobjeto(nombreinicial, edadinicial){
    this.nombre = nombreinicial;
    this.edad = edadinicial;
    this.mostrarnombre = function(){
        alert(this.nombre);
    };
```

```
    this.cambiarnombre = function(nuevonombre){
      this.nombre = nuevonombre;
    };
  }
  Miobjeto.prototype.mostraredad = function(){
    alert(this.edad);
  };
  var empleado = new Miobjeto("Roberto", 55);
  empleado.mostraredad();
</script>
```

Listado 4-35: *Expandiendo el prototipo*

El constructor del Listado 4-35 es el mismo que el del ejemplo anterior. Lo que hicimos esta vez es agregar un nuevo método llamado **`mostraredad()`** al prototipo de **`Miobjeto`**. De ahora en más, cada nueva instancia incluirá este método.

IMPORTANTE: Los prototipos son esenciales en Javascript pero son difíciles de asimilar. Si nos acostumbramos a trabajar con el método **`create()`** y la notación literal estudiada al comienzo de esta parte del capítulo, en la mayoría de los casos no necesitaremos trabajar directamente con prototipos, constructores o el operador **`new`**. De hecho, aplicando el método **`create()`** es la forma adecuada de trabajar en Javascript, creando objetos e instancias sin involucrarnos en la complejidad de un lenguaje de prototipos como éste. Sin embargo, estas técnicas son aún necesarias en algunas circunstancias. Librerías básicas y APIs requieren su uso todo el tiempo, por lo que es bueno que nos familiaricemos con ellas. Las opciones ofrecidas por Javascript a este respecto son casi ilimitadas. Para aprender más acerca de patrones de programación y cómo trabajar con objetos, visite nuestro sitio web y siga los enlaces de este capítulo.

El Objeto Window

Cada vez que el navegador arranca, un objeto global llamado Window es creado para referenciar la ventana del navegador y ofrecer algunos métodos y propiedades básicas. Todo es incluido dentro de este objeto, desde códigos hasta el documento completo.

El objeto Window puede ser accedido desde código Javascript por medio de la propiedad **`window`**. Esta propiedad debería ser mencionada cada vez que queremos acceder propiedades y métodos Javascript, pero no es algo estrictamente necesario. Debido a que éste es el objeto global, el intérprete infiere que todo aquello que no tenga una referencia le pertenece. Por ejemplo, podemos escribir el método del objeto Window **`alert()`** como lo hicimos en anteriores ejemplos o como **`window.alert()`**.

La siguiente es una lista de las propiedades y los métodos de uso más frecuente ofrecidos por este objeto:

alert(texto)—Este método muestra una ventana emergente en la pantalla conteniendo los valores entre paréntesis (ver los ejemplos incluidos en este capítulo).

confirm(texto)—Este método es similar a **`alert()`**, pero ofrece dos botones: Si y No, para que el usuario pueda elegir. Retorna el valor **`true`** (verdadero) o **`false`** (falso) de acuerdo a la respuesta del usuario (vea el ejemplo del Capítulo 14, Listado 14-6).

setTimeout(función, milisegundos)—Este método ejecuta la función especificada en el primer argumento luego del tiempo en milisegundos especificado por el segundo argumento. Podemos asignarle una variable y luego cancelar este proceso usando el método **`clearTimeout()`** con el nombre de la variable entre paréntesis (por ejemplo, **`var tiempo = setTimeout(function(){ alert("Hola"); }, 5000);`**). Un ejemplo del uso de este método es ofrecido en el Capítulo 11, Listado 11-13.

setInterval(función, milisegundos)—Este método es similar a **`setTimeout()`** pero llamará a la función sin parar hasta que el proceso es cancelado por el método **`clearInterval()`** (por ejemplo, **`var tiempo = setInterval(function(){ alert("Hola"); }, 2000);`**). Algunos ejemplos de este método son ofrecidos en los Capítulos 6 y 10.

location—Este objeto incluye varias propiedades para retornar información sobre la dirección URL del documento actual. Puede también ser usado como una propiedad para declarar o retornar la URL del documento (por ejemplo, **`window.location = "http://www.formasterminds.com"`**).

history—Este objeto incluye métodos para navegar a través del historial de la ventana. Algunos de estos métodos son **`back()`**, para cargar el documento anterior, **`forward()`**, para cargar un documento más reciente, y **`go()`**, para cargar cualquier documento en el historial de la ventana. Estudiaremos estos métodos junto con la API History en el Capítulo 18.

El Objeto Document

Como ya mencionamos, casi todo en Javascript es un objeto, desde los objetos que creamos en los ejemplos de este capítulo hasta funciones, matrices, textos o incluso números, por lo que no debería sorprenderle que el documento HTML por completo y cada uno de sus elementos también son objetos.

Dentro del navegador, una estructura interna es creada para procesar el documento HTML. Esta estructura es llamada DOM (Document Object Model) y está compuesta por objetos que representan cada elemento HTML. El DOM comienza con el objeto Document, accesible desde código Javascript a través de la propiedad **`document`**. Usando esta propiedad podemos acceder o modificar cualquier elemento HTML en el documento.

El objeto Document es parte del objeto Window y ofrece varias propiedades y métodos adicionales para trabajar con el documento. Estudiaremos y aplicaremos varios de estos métodos más adelante en este capítulo.

Este contenido forma parte del libro *EL GRAN LIBRO DE HTML5, CSS3 y JAVASCRIPT* de la Editorial Marcombo y del autor L.D Gauchat.
Está prohibida su reproducción y utilización con fines comerciales.