Invocation

# Invocación

La invocación de una función suspende la ejecución de la función actual, pasando el control y los parámetros a la nueva función. Además de los parámetros declarados, toda función recibe dos parámetros adicionales: this y arguments. El parámetro this es muy importante en la programación orientada a objetos, y su valor viene determinado por el parámetro *invocation pattern*. Existen cuatro patrones de invocación en JavaScript: el patrón de invocación a métodos, el patrón de invocación a funciones, el patrón de invocación a constructores y el patrón de invocación a aplicaciones. Los patrones difieren en cómo se inicializa el parámetro de bonificación this.

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

## El patrón de invocación de métodos

Cuando una función se almacena como una propiedad de un objeto, la llamamos una función *method*. Cuando se invoca un método, this se vincula a ese objeto. Si una expresión de invocación contiene un refinamiento (es decir, una expresión . punto o [*subscript*] ), se invoca como un método:

// Create myObject. It has a value and an increment  
// method. The increment method takes an optional  
// parameter. If the argument is not a number, then 1  
// is used as the default.  
  
var myObject = {  
 value: 0,  
 increment: function (inc) {  
 this.value += typeof inc === 'number' ? inc : 1;  
 }  
};  
  
myObject.increment( );  
document.writeln(myObject.value); // 1  
  
myObject.increment(2);  
document.writeln(myObject.value); // 3

Un método puede utilizar this para acceder al objeto, de modo que pueda recuperar valores del objeto o modificarlo. La vinculación de this al objeto se produce en el momento de la invocación. Esta vinculación tan tardía hace que las funciones que utilizan this sean altamente reutilizables. Los métodos que obtienen su contexto de objeto de this se llaman *public methods*.

## El patrón de invocación de funciones

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

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

Cuando se invoca una función con este patrón, this se vincula al objeto global. Esto fue un error en el diseño del lenguaje. Si el lenguaje se hubiera diseñado correctamente, al invocar la función interna, this seguiría estando ligada a la variable this de la función externa. Una consecuencia de este error es que un método no puede emplear una función interna para que le ayude a hacer su trabajo, porque la función interna no comparte el acceso del método al objeto, ya que su this está ligado al valor incorrecto. Afortunadamente, existe una solución fácil. Si el método define una variable y le asigna el valor de this, la función interna tendrá acceso a this a través de esa variable. Por convención, el nombre de esa variable es that:

// Augment myObject with a double method.  
  
myObject.double = function ( ) {  
 var that = this; // Workaround.  
  
 var helper = function ( ) {  
 that.value = add(that.value, that.value);  
 };  
  
 helper( ); // Invoke helper as a function.  
};  
  
// Invoke double as a method.  
  
myObject.double( );  
document.writeln(myObject.value); // 6

## El patrón de invocación al constructor

JavaScript es un *prototypal* lenguaje de herencia. Eso significa que los objetos pueden heredar propiedades directamente de otros objetos. El lenguaje no tiene clases.

Esto supone un cambio radical respecto a la moda actual. La mayoría de los lenguajes actuales son *classical*. La herencia prototípica es muy expresiva, pero no se entiende muy bien. El propio JavaScript no confía en su naturaleza prototípica, por lo que ofrece una sintaxis de creación de objetos que recuerda a la de los lenguajes clásicos. Pocos programadores clásicos consideraron aceptable la herencia prototípica, y la sintaxis de inspiración clásica oscurece la verdadera naturaleza prototípica del lenguaje. Es lo peor de ambos mundos.

Si se invoca una función con el prefijo new, se creará un nuevo objeto con un enlace oculto al valor del miembro prototype de la función, y this se vinculará a ese nuevo objeto.

El prefijo new también cambia el comportamiento de la sentencia return. Veremos más sobre esto a continuación.

// Create a constructor function called Quo.  
// It makes an object with a status property.  
  
var Quo = function (string) {  
 this.status = string;  
};  
  
// Give all instances of Quo a public method  
// called get\_status.  
  
Quo.prototype.get\_status = function ( ) {  
 return this.status;  
};  
  
// Make an instance of Quo.  
  
var myQuo = new Quo("confused");  
  
document.writeln(myQuo.get\_status( )); // confused

Las funciones destinadas a ser utilizadas con el prefijo new se denominan *constructors*. Por convención, se guardan en variables con nombre en mayúsculas. Si se llama a un constructor sin el prefijo new, pueden ocurrir cosas muy malas sin que se produzca una advertencia en tiempo de compilación o de ejecución, por lo que la convención sobre el uso de mayúsculas es realmente importante.

No se recomienda utilizar este estilo de funciones constructoras. Veremos alternativas mejores en el próximo capítulo.

## El patrón de invocación Apply

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

El método apply nos permite construir una matriz de argumentos que utilizaremos para invocar una función. También nos permite elegir el valor de this. El método apply recibe dos parámetros. El primero es el valor que debe vincularse a this. El segundo es una matriz de parámetros.

// Make an array of 2 numbers and add them.  
  
var array = [3, 4];  
var sum = add.apply(null, array); // sum is 7  
  
// Make an object with a status member.  
  
var statusObject = {  
 status: 'A-OK'  
};  
  
// statusObject does not inherit from Quo.prototype,  
// but we can invoke the get\_status method on  
// statusObject even though statusObject does not have  
// a get\_status method.  
  
var status = Quo.prototype.get\_status.apply(statusObject);  
 // status is 'A-OK'