Analyzing JavaScript

# Analizando JavaScript

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

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

Las funciones de JavaScript son objetos de primera clase con alcance (mayoritariamente) léxico. JavaScript es el primer lenguaje lambda que se generaliza. En el fondo, JavaScript tiene más en común con Lisp y Scheme que con Java. Es Lisp vestido de C. Esto hace de JavaScript un lenguaje extraordinariamente potente.

La moda actual en la mayoría de los lenguajes de programación exige un tipado fuerte. La teoría es que el tipado fuerte permite al compilador detectar una gran clase de errores en tiempo de compilación. Cuanto antes podamos detectar y reparar los errores, menos nos costarán. JavaScript es un lenguaje poco tipado, por lo que los compiladores de JavaScript son incapaces de detectar errores de tipo. Esto puede ser alarmante para las personas que llegan a JavaScript desde lenguajes fuertemente tipados. Pero resulta que la tipificación fuerte no elimina la necesidad de pruebas cuidadosas. Y he descubierto en mi trabajo que los tipos de errores que encuentra la comprobación de tipos fuertes no son los errores que me preocupan. Por otra parte, la tipificación libre me parece liberadora. No necesito formar complejas jerarquías de clases. Y nunca tengo que hacer castings ni pelearme con el sistema de tipos para obtener el comportamiento que quiero.

JavaScript tiene una notación literal de objeto muy potente. Los objetos pueden crearse simplemente enumerando sus componentes. Esta notación fue la inspiración de JSON, el popular formato de intercambio de datos. (En [el Apéndice](ape.html) E encontrarás más información sobre JSON).

Una característica controvertida de JavaScript es la herencia prototípica. JavaScript tiene un sistema de objetos sin clases en el que los objetos heredan propiedades directamente de otros objetos. Esto es realmente potente, pero resulta desconocido para los programadores con formación clásica. Si intentas aplicar patrones de diseño clásicos directamente a JavaScript, te sentirás frustrado. Pero si aprendes a trabajar con la naturaleza prototípica de JavaScript, tus esfuerzos se verán recompensados.

JavaScript es muy denostado por su selección de ideas clave. Sin embargo, en su mayor parte, esas elecciones fueron buenas, aunque inusuales. Pero hubo una elección que fue especialmente mala: JavaScript depende de las variables globales para la vinculación. Todas las variables de nivel superior de todas las unidades de compilación se juntan en un espacio de nombres común llamado *the global object*. Esto es malo porque las variables globales son malvadas, y en JavaScript son fundamentales. Afortunadamente, como veremos, JavaScript también nos proporciona herramientas para mitigar este problema.

En algunos casos, no podemos ignorar las partes malas. Hay algunas partes horribles inevitables, que se irán señalando a medida que ocurran. También se resumirán en [el Apéndice A](apa.html). Pero en este libro conseguiremos evitar la mayoría de las partes malas, resumiendo gran parte de lo que se omitió en el Apéndice [B](apb.html). Si quieres saber más sobre las partes malas y cómo utilizarlas mal, consulta cualquier otro libro de JavaScript.

El estándar que define JavaScript (también conocido como JScript) es la tercera edición de *The ECMAScript Programming Language*que está disponible en [http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf.](http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf) El lenguaje descrito en este libro es un subconjunto propio de ECMAScript. Este libro no describe todo el lenguaje porque deja fuera las partes malas. El tratamiento que aquí se hace no es exhaustivo. Evita los casos extremos. Tú también deberías hacerlo. Hay peligro y miseria en los bordes.

[El Apéndice C](apc.html) describe una herramienta de programación llamada JSLint, un analizador sintáctico de JavaScript que puede analizar un programa JavaScript e informar sobre las partes malas que contiene. JSLint proporciona un grado de rigor que suele faltar en el desarrollo de JavaScript. Puede darte la seguridad de que tus programas contienen sólo las partes buenas.

JavaScript es un lenguaje de muchos contrastes. Contiene muchos errores y aristas afiladas, por lo que podrías preguntarte: "¿Por qué debería utilizar JavaScript?". Hay dos respuestas: la primera es que no tienes elección. La Web se ha convertido en una importante plataforma para el desarrollo de aplicaciones, y JavaScript es el único lenguaje que se encuentra en todos los navegadores. Es una pena que Java fracasara en ese entorno; si no lo hubiera hecho, podría haber una opción para las personas que desean un lenguaje clásico fuertemente tipado. Pero Java fracasó y JavaScript está floreciendo, así que hay pruebas de que JavaScript hizo algo bien.

La otra respuesta es que, a pesar de sus deficiencias, *JavaScript is really good*. es ligero y expresivo. Y una vez que le coges el truco, la programación funcional es muy divertida.

Pero para utilizar bien el lenguaje, debes estar bien informado sobre sus limitaciones. Te hablaré de ellas con cierta brutalidad. No dejes que eso te desanime. Las partes buenas son lo suficientemente buenas como para compensar las malas.