Chapter 9. Style

# Capítulo 9. Estilo

¡He aquí un tonto estilo señorial!

-WilliamShakespeare, *The First Part of Henry the Sixth*

Los programas informáticos son las cosas más complejas que fabrica el ser humano. Los programas se componen de un enorme número de partes, expresadas como funciones, sentencias y expresiones que se ordenan en secuencias que deben estar prácticamente libres de errores. El comportamiento en tiempo de ejecución se parece muy poco al programa que lo implementa. Normalmente se espera que el software se modifique a lo largo de su vida productiva. El proceso de convertir un programa correcto en otro programa correcto diferente es extremadamente difícil.

Los buenos programas tienen una estructura que anticipa -pero no se ve sobrecargada por- las posibles modificaciones que serán necesarias en el futuro. Los buenos programas también tienen una presentación clara. Si un programa está bien expresado, tenemos más posibilidades de entenderlo para poder modificarlo o repararlo con éxito.

Estas preocupaciones son válidas para todos los lenguajes de programación, y especialmente para JavaScript. La tipificación poco precisa y la excesiva tolerancia a los errores de JavaScript ofrecen pocas garantías en tiempo de compilación sobre la calidad de nuestros programas, por lo que, para compensar, debemos codificar con una disciplina estricta.

JavaScript contiene un amplio conjunto de características débiles o problemáticas que pueden socavar nuestros intentos de escribir buenos programas. Evidentemente, debemos evitar las peores características de JavaScript. Sorprendentemente, quizá también debamos evitar las características que suelen ser útiles pero ocasionalmente peligrosas. Dichas funciones son molestias atractivas y, al evitarlas, se evita una gran clase de errores potenciales.

El valor a largo plazo del software para una organización está en proporción directa con la calidad del código base. A lo largo de su vida, un programa será manejado por muchos pares de manos y ojos. Si un programa es capaz de comunicar claramente su estructura y características, es menos probable que se rompa cuando se modifique en un futuro nunca demasiado lejano.

El código JavaScript a menudo se envía directamente al público. Siempre debe tener calidad de publicación. La pulcritud cuenta. Si escribes con un estilo claro y coherente, tus programas serán más fáciles de leer.

Los programadores pueden debatir interminablemente sobre lo que constituye un buen estilo. La mayoría de los programadores están firmemente arraigados a lo que están acostumbrados, como el estilo imperante en el lugar donde fueron a la escuela, o en su primer trabajo. Algunos han tenido carreras provechosas sin ningún sentido del estilo. ¿No es eso una prueba de que el estilo no importa? E incluso si el estilo no importa, ¿no es un estilo tan bueno como cualquier otro?

Resulta que el estilo importa en la programación por la misma razón que importa en la escritura. Se lee mejor.

A veces se piensa que los programas de ordenador son un medio sólo de escritura, por lo que importa poco cómo esté escrito mientras funcione. Pero resulta que la probabilidad de que un programa funcione aumenta significativamente con nuestra capacidad de leerlo, lo que también aumenta la probabilidad de que realmente funcione según lo previsto. Además, la naturaleza del software es modificarse ampliamente a lo largo de su vida productiva. Si podemos leerlo y comprenderlo, podemos esperar modificarlo y mejorarlo.

A lo largo de este libro he utilizado un estilo coherente. Mi intención era que los ejemplos de código fueran lo más fáciles de leer posible. He utilizado los espacios en blanco de forma coherente para darte más pistas sobre el significado de mis programas.

He sangrado el contenido de los bloques y los literales de los objetos cuatro espacios. Coloqué un espacio entre if y (para que if no pareciera una invocación a una función. Sólo en las invocaciones hice que ( fuera adyacente con el símbolo precedente. Pongo espacios alrededor de todos los operadores infijos excepto en . y [, que no reciben espacios porque tienen mayor precedencia. Pongo un espacio después de cada coma y dos puntos.

Pongo como máximo una sentencia en cada línea. Si una frase no cabe en una línea, la rompo después de una coma o de un operador binario. Eso ofrece más protección contra los errores de copiar y pegar que se enmascaran con la inserción del punto y coma. (La tragedia de la inserción del punto y coma se desvelará en [el Apéndice A](apa.html).) Pongo cuatro espacios más en el resto de la sentencia, u ocho espacios si cuatro es ambiguo (como un salto de línea en la parte de la condición de una sentencia if ).

I *always* utilizo bloques con sentencias estructuradas como if y while porque es menos propenso a errores. Lo he visto:

if (a)  
 b( );

convertirse:

if (a)  
 b( );  
 c( );

que es un error muy difícil de detectar. Se parece a

if (a) {  
 b( );  
 c( );  
}

pero significa

if (a) {  
 b( );  
}  
c( );

El código que parece significar una cosa pero en realidad significa otra es probable que cause errores. Un par de llaves es una protección realmente barata contra errores que pueden ser caros de encontrar.

Siempre utilizo el estilo K&R, poniendo el { al final de una línea en lugar de delante, porque evita un horrible error de diseño en la declaración return de JavaScript.

He incluido algunos comentarios. Me gusta poner comentarios en mis programas para dejar información que leerán más adelante personas (posiblemente yo mismo) que necesitarán entender lo que estaba pensando. A veces pienso en los comentarios como en una máquina del tiempo que utilizo para enviar mensajes importantes a mi yo futuro.

Me esfuerzo por mantener los comentarios actualizados. Los comentarios erróneos pueden hacer que los programas sean aún más difíciles de leer y entender. No puedo permitirme eso.

He intentado no hacerte perder el tiempo con comentarios inútiles como éste:

i = 0; // Set i to zero.

En JavaScript, prefiero utilizar comentarios de línea. Reservo los comentarios de bloque para la documentación formal y para comentar.

Prefiero que la estructura de mis programas sea autoiluminadora, eliminando la necesidad de comentarios. No siempre tengo éxito, así que mientras mis programas esperan la perfección, escribo comentarios.

JavaScript tiene sintaxis C, pero sus bloques no tienen ámbito. Por tanto, la convención de que las variables deben declararse en su primer uso es un mal consejo en JavaScript. JavaScript tiene ámbito de función, pero no de bloque, así que declaro todas mis variables al principio de cada función. JavaScript permite declarar las variables después de utilizarlas. Eso me parece un error, y no quiero escribir programas que parezcan errores. Quiero que mis errores destaquen. Del mismo modo, nunca utilizo una expresión de asignación en la parte de la condición de un ifporque:

if (a = b) { ... }

probablemente sea la intención:

if (a === b) { ... }

Quiero evitar frases hechas que parezcan errores.

Nunca permito que los casos de conmutación pasen al caso siguiente. Una vez encontré un error en mi código provocado por un "fall through" involuntario inmediatamente después de haber hecho un enérgico discurso sobre por qué el "fall through" era útil a veces. Tuve la suerte de poder aprender de la experiencia. Cuando reviso las características de un lenguaje, ahora presto especial atención a las características que a veces son útiles pero ocasionalmente peligrosas. Ésas son las *worst parts* porque es difícil saber si se utilizan correctamente. Es un lugar donde se esconden los bugs.

La calidad no fue una preocupación motivadora en el diseño, implementación o estandarización de JavaScript. Eso pone una mayor carga en los usuarios del lenguaje para resistir las debilidades del mismo.

JavaScript ofrece soporte para programas grandes, pero también proporciona formas y modismos que van en contra de los programas grandes. Por ejemplo, JavaScript facilita el uso de variables globales, pero éstas resultan cada vez más problemáticas a medida que aumenta la complejidad de los programas.

Yo utilizo una única variable global para contener una aplicación o biblioteca. Cada objeto tiene su propio espacio de nombres, por lo que es fácil utilizar objetos para organizar mi código. El uso del cierre proporciona una mayor ocultación de la información, aumentando la solidez de mis módulos.