IRINA ALFARO
RESUMEN JAVASCRIPT
CETAV 2015

Que es JavaScript?

‘An object-oriented computer programming language commonly used to create interactive effects within web browsers.’
JavaScript es un lenguaje de programación que permite a los desarrolladores crear acciones en sus páginas web.
JavaScript tiene la ventaja de ser incorporado en cualquier página web, puede ser ejecutado sin la necesidad de instalar otro programa para ser visualizado.
Es un lenguaje fácil de emplear, entender y trabajar.
Gran parte de la programación en este lenguaje está centrada en describir objetos, escribir funciones que respondan a movimientos del mouse, aperturas, utilización de teclas, cargas de páginas entre otros.

Cómo nació JavaScript?

JavaScript nació con la necesidad de permitir a los autores de sitio web crear páginas que permitan intercambiar con los usuarios, ya que se necesitaba crear webs de mayor complejidad. El HTML solo permitía crear páginas estáticas donde se podía mostrar textos con estilos, pero se necesitaba interactuar con los usuarios.
Gracias a su compatibilidad con todos los navegadores modernos se ha convertido en un estándar como lenguaje de programación del lado del cliente.
Del lado del cliente: las extensiones del lado del cliente permiten a una aplicación colocar elementos en un formulario HTML y responder a los eventos del usuario como clicks del mouse, ingresos en formularios, y navegación en la página.
Del lado del servidor: las extensiones del lado del servidor permiten a las aplicaciones comunicarse con una base de datos relacional, proveyendo un flujo constante de información, o realizar movimientos de archivos en el servidor.

Diferencias entre JavaScript y Java

Java es un lenguaje de programación multiplataforma orientado a objetos. Este lenguaje, por su propiedad de multiplataforma, se puede implementar en cualquier entorno (linux, windows, unix, etc) ya que lo único necesario para que ese lenguaje sea interpretado es que el sistema sobre el cual se van a correr las aplicaciones, webs, etc. posean la JVM (Java Virtual Machine), que es el interpretador de lenguaje java. Los archivos fuente se guardan como .java y los "compilados" se guardan como .class, que son los archivos interpretados por la JVM. Otra cosa es que java es un lenguaje de clases.

Por otro lado, JavaScript es un lenguaje que se implementa sobre plataforma web, el cual sirve para diversas funciones sobre la misma. Contrariamente a lo que informaron en la respuesta, no es solo para realizar retoques en páginas web, si no que JavaScript también cuenta con programación de objetos para interactuar en las páginas (no es solo scripting interpretado). Esto hace que se puedan programar juegos en JavaScript (obviamente interpretados sobre una web). Al mismo tiempo, al igual que java, JavaScript cuenta con diversos frameworks, como prototype, jquery, moo, etc. con los cuales se pueden programar aplicaciones web con muchas utilidades y robustez.
Diferencias:

Compilado

Para programar en Java necesitamos UN Kit de desarrollo y UN compilador. Sin embargo, JavaScript no es UN lenguaje que necesite que sus programas se compilen, sino que éstos se interpretan por parte del navegador cuando éste lee la página.

Orientado a objetos

Java es UN lenguaje de programación orientado a objetos. (Más tarde veremos que quiere decir orientado a objetos, para el que no lo sepa todavía) JavaScript no es orientado a objetos, esto quiere decir que podremos programar sin necesidad de crear clases, Tal como se realiza en los lenguajes de programación estructurada como C o Pascal.

Propósito

Java es mucho más potente que JavaScript, esto es debido a que Java es un lenguaje de propósito general, con el que se pueden hacer aplicaciones de lo más variado, sin embargo, con JavaScript sólo podemos escribir programas para que se ejecuten en páginas web.

Estructuras Fuertes

Java es UN lenguaje de programación fuertemente tipado, esto quiere decir que al declarar una variable tendremos que indicar su tipo y no podrá cambiar de UN tipo a otro automáticamente. Por su parte JavaScript no tiene esta característica, y podemos meter en una variable la información que deseemos, independientemente del tipo de ésta. Además, podremos cambiar el tipo de información de una variable cuando queramos.

Otras características

Como vemos Java es mucho más complejo, aunque también más potente, robusto y seguro. Tiene más funcionalidades que JavaScript y las diferencias que los separan son lo suficientemente importantes Como para distinguirlos fácilmente.

JavaScript y la especificación ECMA

Netscape inventó JavaScript y JavaScript fue utilizado por primera vez en los navegadores Netscape. Sin embargo, Netscape está trabajando con Ecma International - la asociación Europea para la

estandarización de la información y de los sistemas de comunicación (European Computer Manufacturers Association) para la entrega de un lenguaje de programación internacional estandarizado basado en el núcleo de JavaScript. Esta versión estandarizada de JavaScript, llamada ECMAScript, se comporta de la misma manera en todas las aplicaciones que soportan el estándar. Las compañías pueden usar el estándar abierto del lenguaje para desarrollar sus implementaciones en JavaScript. El estándar ECMAScript está documentado en la especificación ECMA-262.
El estándar ECMA-262 también es provisto por la organización para la estandarización internacional ISO (International Organization for Standardization) La especificación ECMAScript no describe el modelo objeto documento (DOM) [Document Object Model], el cual está estandarizado por el consorcio W3C World Wide Web Consortium (W3C). El DOM define la manera en la cual los objetos documentos HTML son expuestos para el diseño de su script.

La documentación de JavaScript versus la especificación ECMA

La especificación ECMAScript es un conjunto de requerimientos para la implementación de ECMAScript; esto es útil si desea determinar si la característica de JavaScript es soportada en otras implementaciones ECMAScript. Si planea escribir código JavaScript que utilice sólo características soportadas por ECMAScript, entonces necesita revisar la especificación ECMAScript.

Introducción a OOP

La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990.
Los objetos son entidades que tienen un determinado estado, comportamiento (método) e identidad:
-El estado está compuesto de datos o informaciones; serán uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
-El comportamiento está definido por los métodos o mensajes a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.
-La identidad es una propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

La POO difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada solo se escriben funciones que procesan datos. Los programadores que emplean Programación Orientada a Objetos, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

Conceptos

Clase

Definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ella.

Herencia

(Por ejemplo, herencia de la clase C a la clase D) es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C, como si esos atributos y operaciones hubiesen sido definidos por la misma D. Por lo tanto, puede usar los mismos métodos y variables públicas declaradas en C. Los componentes registrados como "privados" (private) también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos. Esto es así para mantener hegemónico el ideal de POO.

Objeto

Instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa). Es una instancia a una clase.

Método

Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.

Evento

Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento la reacción que puede desencadenar un objeto; es decir, la acción que genera.

Atributos

Características que tiene la clase.

Mensaje

Una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

Propiedad o atributo

Contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

Estado interno

Es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos). No es visible al programador que maneja una instancia de la clase.

Componentes de un objeto

Atributos, identidad, relaciones y métodos.

Identificación de un objeto

Un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

En comparación con un lenguaje imperativo, una "variable" no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

Reserved keywords as of ECMAScript 6

Break	case	class	catch
const	continue	debugger	default
Delete	do	else	export
Extends	finally	for	function
If	import	ininstanceof	let
New	return	super	switch
This	throw	try	type of
Var	void	while	with
Yield			

Los siguientes están reservadas como futura palabras clave cuando se encuentran en el código de modo estricto:

Implements	package	protected	static
Interface	private	public	

Los siguientes son palabras reservadas como palabras clave futura por mayores especificaciones de ECMAScript:

Abstract	Boolean	by	techar
Double	final	float	goto
Int	long	native	short
Synchronized	transient	volatile	

Estas son también palabras reservadas, aunque no son utilizadas en este momento en JavaScript y se reservan para usos futuros:

class, enum, extends, super, const, export e import

En modo estricto ("use strict") también se consideran las siguientes palabras como reservadas:

implements, let, private, public, yield, interface, package, protected y static

V8 (motor JavaScript)

V8 es un motor de código abierto para JavaScript creado por Google, siendo su programador jefe Lars Bak.[3]
Está escrito en C++ y es usado en Google Chrome. También el "V8 JavaScript" está integrado en el navegador de internet del sistema operativo Android 2.2 “Froyo”. Implementa ECMAScript como especifica ECMA-262 5.ª edición y corre en Windows XP, Vista, Mac OS X 10.5 (Leopard) y Linux en procesadores IA-32 y ARM.
V8 puede funcionar de manera individual (standalone) o incorporada a cualquier aplicación C++.

Server-side and client-side programming

Web development is all about communication. In this case, communication between two parties, over the HTTP protocol:

- The **Server** - This party is responsible for **serving** pages.
- The **Client** - This party *requests* pages from the **Server**, and displays them to the user. In most cases, the client is a **web browser**.
    - The **User** - The user *uses* the **Client** in order to surf the web, fill in forms, watch videos online, etc.

Server-side Programming

Server-side programming, is the general name for the kinds of programs, which are run on the **Server**.
Uses

- Process user input.
- Display pages.
- Structure web applications.
- Interact with permanent storage (SQL, files).

Example Languages

- PHP
- ASP.Net in C#, C++, or Visual Basic.
- Nearly any language (C++, C#, Java). These were not designed specifically for the task, but are now often used for application-level web services.

Client-side programming
Much like the server-side, Client-side programming is the name for all of the programs, which are run on the **Client**.
Uses

- Make interactive webpages.
- Make stuff happen dynamically on the web page.
- Interact with temporary storage, and local storage (Cookies, local Storage).
- Send requests to the server, and retrieve data from it.
- Provide a remote service for client-side applications, such as software registration, content delivery, or remote multi-player gaming.

Example languages

- JavaScript (primarily)
- HTML*
- CSS*

Any language running on a client device that interacts with a remote service is a client-side language.

Scripting Language

A high-level programming language that is interpreted by another program at runtime rather than compiled by the computer's processor as other programming languages (such as C and C++) are. Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a Web page, such as different menu styles or graphic displays or to serve dynamic advertisements. These types of languages are client-side scripting languages, affecting the data that the end user sees in a browser window. Other scripting languages are server-side scripting languages that manipulate the data, usually in a database, on the server.
Scripting languages came about largely because of the development of the Internet as a communications tool. JavaScript, ASP, JSP, PHP, Perl, Tcl and Python are examples of scripting languages.

The origin of the term was similar to its meaning in "a movie script tells actors what to do": a scripting language controlled the operation of a normally-interactive program, giving it a sequence of work to do all in

one batch. For instance, one could put a series of editing commands in a file, and tell an editor to run that "script" as if those commands had been typed interactively.
This usage was generalized under Unix to describe scripts for the shell to execute, which meant that the running of a whole series of programs was controlled, not just one. Thus "shell script".

Investigaciones

Diferencia entre Window.Onload y Document.Ready

El Document.Ready ocurre después que el HTML ha sido cargado, mientras que el Window.Onload ocurre cuando el contenido (por ejemplo las imágenes) han sido cargadas.
Document.Ready es un evento específico de Jquery, Window.Onload es un evento estándar del DOM.

Diferencia entre Window.Onload y Window.Onunload

Onload es aquel que se produce cuando un navegador carga un documento HTML o una imagen.
Onunload tiene como misión ejecutar un script cuando la página web actual se descarga, ya sea porque se accede a otra página o porque se pulsan los botones de retroceder y avanzar.

¿Para qué sirve NoScript?

Muestra un mensaje al usuario cuando su navegador no puede ejecutar JavaScript.

Diferencia entre Null y Undefined

Undefined

Para Javascript, no existe. O bien no ha sido declarada o jamás se le asignó un valor.

Null

Para Javascript, **la variable existe**. En algún momento, explícitamente, la variable se estableció a null.

La presencia de operadores

La precedencia de operadores determina el orden en que se evalúan los operadores. Los operadores con mayor precedencia se evalúan primero.

Asociatividad

La asociatividad determina el orden en que se procesan los operadores con la misma precedencia. Por ejemplo, considere una expresión:
una OP OP b c
Asociatividad por la izquierda (de izquierda a derecha) significa que se procesa como (a OP b) c OP, mientras asociatividad por la derecha (de derecha a izquierda) significa que se interpreta como una OP (b OP c). Operadores de asignación son asociativo por la derecha, por lo que puede escribir:
a = b = 5;

con el resultado esperado que ayb obtener el valor 5. Esto se debe a que el operador de asignación devuelve el valor que se le asigna. En primer lugar, b se establece en 5. A continuación, la una se establece en el valor de b.

BOOK (JavaScript: The Good Parts)

Chapter 1: Good Parts

Most programming languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts. After all, how can you build something good out of bad parts?
Fortunately, JavaScript has some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy.

WHY JAVASCRIPT?

JavaScript is an important language because it is the language of the web browser. Its association with the browser makes it one of the most popular programming languages in the world. At the same time, it is one of the most despised programming languages in the world.
The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming.

Analyzing JavaScript

The very good ideas include functions, loose typing, dynamic objects, and an expressive object literal notation. The bad ideas include a programming model based on global variables.
JavaScript's functions are first class objects with (mostly) lexical scoping. JavaScript is the first lambda language to go mainstream. Deep down, JavaScript has more in common with Lisp and Scheme than with Java. It is Lisp in C's clothing. This makes JavaScript a remarkably powerful language.
JavaScript has a very powerful object literal notation. Objects can be created simply by listing their components. This notation was the inspiration for JSON, the popular data interchange format.
JavaScript is much maligned for its choice of key ideas. For the most part, though, those choices were good, if unusual. But there was one choice that was particularly bad: JavaScript depends on global variables for linkage. All of the top-level variables of all compilation units are tossed together in a common namespace called the global object . This is a bad thing because global variables are evil, and in JavaScript they are fundamental.

“Why should I use JavaScript?” There are two answers. The first is that you don’t have a choice. The Web has become an important platform for application development, and JavaScript is the only language that is found in all browsers.

Capítulo 2.

GRAMMAR

This chapter introduces the grammar of the good parts of JavaScript, presenting a quick overview of how the language is structured.

Whitespace

JavaScript offers two forms of comments, block comments formed with /* */ and line-ending comments starting with // . Take care that the comments always accurately describe the code. Obsolete comments are worse than no comments.

Names

Most of the reserved words in this list are not used in the language. The list does not include some words that should have been reserved but were not, such as undefined , NaN , and Infinity .
Names are used for statements, variables, parameters, property names, operators, and labels.
Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value.
Negative numbers can be formed by using the – prefix operator.
The value NaN is a number value that is the result of an operation that cannot produce a normal result. NaN is not equal to any value, including itself.

Strings

A string literal can be wrapped in single quotes or double quotes. It can contain zero
or more characters.
Strings have a length property. For example, "seven".length is 5. Strings are immutable. Once it is made, a string can never be changed. But it is easy to make a new string by concatenating other strings together with the + operator.
Two strings containing exactly the same characters in the same order are considered to be the same string. So:
'c' + 'a' + 't' === 'cat'
is true.

Statements

A compilation unit contains a set of executable statements. In web browsers, each <script> tag delivers a compilation unit that is compiled and immediately executed
Statements tend to be executed in order from top to bottom. The sequence of execution can be altered by the conditional statements (if and switch), by the looping statements (while , for , and do), by the disruptive statements (break , return , and throw), and by function invocation.

A block is a set of statements wrapped in curly braces. Unlike many other languages, blocks in JavaScript do not create a new scope, so variables should be defined at the top of the function, not in blocks.

Switch statement

The switch statement performs a multi way branch. It compares the expression for equality with all of the specified cases. The expression can produce a number or a string.

Case clause

A case clause contains one or more case expressions. The case expressions need not be constants. The statement following a clause should be a disruptive statement to prevent fall through into the next case. The break statement can be used to exit from a switch.

While statement

The while statement performs a simple loop. If the expression is false, then the loop will break. While the expression is truth, the block will be executed.
The conventional form is controlled by three optional clauses: the initialization, the condition, and the increment. First, the initialization is done, which typically initializes the loop variable. Then, the condition is evaluated. Typically, this tests the loop variable against a completion criterion. If the condition is omitted, then a condition of true is assumed. If the condition is false, the loop breaks. Otherwise, the block is executed, then the increment executes, and then the loop repeats with the condition

For statement

The other form (called for in) enumerates the property names (or keys) of an object. On each iteration, another property name string from the object is assigned to the variable.

Do statement

The do statement is like the while statement except that the expression is tested after the block is executed instead of before. That means that the block will always be executed at least once.

Try statement

The try statement executes a block and catches any exceptions that were thrown by the block. The catch clause defines a new variable that will receive the exception object..

Throw statement

The expression is usually an object literal containing a name property and a message property. The catcher of the exception can use that information to determine what to do.

Return statement

The return statement causes the early return from a function. It can also specify the value to be returned. If a return expression is not specified, then the return value will be undefined.

Break statement

The break statement causes the exit from a loop statement or a switch statement. It can optionally have a label that will cause an exit from the labeled statement.

Expression statement

An expression statement can either assign values to one or more variables or members, invoke a method, delete a property from an object.

Expressions

The simplest expressions are a literal value (such as a string or number), a variable, a built-in value (true , false , null , undefined , NaN , or Infinity), an invocation expression preceded by new , a refinement expression preceded by delete , an expression wrapped in parentheses, an expression preceded by a prefix operator.

The operators at the top of the operator precedence list in Table 2-1 have higher precedence. They bind the tightest. The operators at the bottom have the lowest precedence.

Operator precedence

. [] ( )	Refinement and invocation
delete new typeof + - !	Unary operators
* / %	Multiplication, division, modulo
+ -	Addition/concatenation, subtraction
>= <= > <	Inequality
=== !==	Equality
&&	Logical and
||	Logical or
?:	Ternary

Prefix operator

If the operand of! Is truth, it produces false. Otherwise, it produces true .

The + operator adds or concatenates. If you want it to add, make sure both operands are numbers.

The / operator can produce a no ninteger result even if both operands are integers.

The && operator produces the value of its first operand if the first operand is false. Otherwise, it produces the value of the second operand.

Infix operator

The || operator produces the value of its first operand if the first operand is truth. Otherwise, it produces the value of the second operand.

Invocation

Invocation causes the execution of a function value. The invocation operator is a pair of parentheses that follow the function value. The parentheses can contain arguments that will be delivered to the function.

Refinement

A refinement is used to specify a property or element of an object or array. This will be described in detail in the next chapter**.**

Literals

Object literals are a convenient notation for specifying new objects. The names of the properties can be specified as names or as strings.

Functions

A function literal defines a function value. It can have an optional name that it can use to call itself recursively. It can specify a list of parameters that will act as variables initialized by the invocation arguments. The body of the function includes variable definitions and statements.

String.prototype.indexOf()

El método **indexOf()** devuelve el índice, dentro del objeto String que realiza la llamada, de la primera ocurrencia del valor especificado, comenzando la búsqueda desde indiceBusqueda; o -1 si no se encuentra dicho valor.

Syntax

cadena.indexOf(*valorBusqueda*[, *indiceDesde*])

-ValorBusqueda

Una cadena que representa el valor de búsqueda.

-IndiceDesde

La localización dentro de la cadena llamada desde la que empezará la búsqueda. Puede ser un entero entre 0 y la longitud de la cadena. El valor predeterminado es 0.

El método indexOf es sensible a mayúsculas.

Ejemplo: Usando indexOf y lastIndexOf:

```
var cualquierCadena="Brave new world"

-document.write("<P>The index of the first w from the beginning is " +    cualquierCadena.indexOf("w"))
// Muestra 8
-document.write("<P>The index of the first w from the end is " +    cualquierCadena.lastIndexOf("w"))    //
Muestra 10
-document.write("<P>The index of 'new' from the beginning is " +    cualquierCadena.indexOf("new"))     //
Muestra 6
-document.write("<P>The index of 'new' from the end is " +    cualquierCadena.lastIndexOf("new"))  //
Muestra 6
```

Example: Con indexOf() para contar las ocurrencias de una letra en un string. El siguiente ejemplo se establece el número de apariciones de la letra e de la cadena str:

```
var str = 'To be, or not to be, that is the question.';
var count = 0;
var pos = str.indexOf('e');

while (pos !== -1) {
      count++;
      pos = str.indexOf('e', pos + 1);
}
console.log(count); // displays 4
```

JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top.
A variable can be used before it has been declared, but it’s not a good practice.
Declare Your Variables At the Top!
Hoisting is (to many developers) an unknown or overlooked behavior of JavaScript.
If a developer doesn't understand hoisting, programs may contain bugs (errors).
To avoid bugs, always declare all variables at the beginning of every scope.
Since this is how JavaScript interprets the code, it is always a good rule.

Strict JS

"Use Strict"; Defines that JavaScript code should be executed in "strict mode".
Declared at the beginning of a JavaScript file, it has global scope (all code will execute in strict mode).
Declared inside a function, it has local scope (only the code inside the function is in strict mode).

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.
Strict mode changes previously accepted "bad syntax" into real errors.
As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Not Allowed in Strict Mode

Using a variable (property or object) without declaring it, is not allowed:

```
"use strict";
x = 3.14;  // This will cause an error (if x has not been declared)
```

Deleting a variable, a function, or an argument, is not allowed.

```
"use strict";
x = 3.14;
delete x;  // This will cause an error
```

Defining a property more than once, is not allowed:

```
"use strict";
var x = {p1:10, p1:20};   // This will cause an error
```

Duplicating a parameter name is not allowed:

```
"use strict";
function x(p1, p1) {};    // This will cause an error
```

Octal numeric literals and escape characters are not allowed:

```
"use strict";
var x = 010;            // This will cause an error
var y = \010;           // This will cause an error
```

Writing to a read-only property is not allowed:

```
"use strict";
var obj = {};
obj.defineProperty(obj, "x", {value:0, writable:false});
obj.x = 3.14;                // This will cause an error
```

Writing to a get-only property is not allowed:

```
"use strict";
var obj = {get x() {return 0} };
obj.x = 3.14;           // This will cause an error
```

Deleting an undeletable property is not allowed:

```
"use strict";
delete Object.prototype; // This will cause an error
```

The string "eval" cannot be used as a variable:

```
"use strict";
var eval = 3.14;         // This will cause an error
```

The string "arguments" cannot be used as a variable:

```
"use strict";
var arguments = 3.14;    // This will cause an error
```

The with statement is not allowed:

```
"use strict";
with (Math){x = cos(2)}; // This will cause an error
```

For security reasons, eval() are not allowed to create variables in the scope from which it was called:

```
"use strict";
eval ("var x = 2");
alert (x)               // This will cause an error
```

In function calls like f(), the this value was the global object. In strict mode, it is now undefined.

JavaScript Common Mistakes

Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This **if** statement returns **false** (as expected) because x is not equal to 10:

```
var x = 0;
if (x == 10)
```

This **if** statement returns **true** (maybe not as expected), because 10 is true:

```
var x = 0;
if (x = 10)
```

This **if** statement returns **false** (maybe not as expected), because 0 is false:

```
var x = 0;
if (x = 0)
```

Expecting Loosely Comparison

In regular comparison, data type does not matter. This if statement returns true:

```
var x = 10;
var y = "10";
if (x == y)
```

In strict comparison, data type does matter. This if statement returns false:

```
var x = 10;
var y = "10";
if (x === y)
```

It is a common mistake to forget that switch statements use strict comparison:
This case switch will display an alert:

```
var x = 10;
switch(x) {
   case 10: alert("Hello");
}
```

This case switch will not display an alert:

```
var x = 10;
switch(x) {
   case "10": alert("Hello");
}
```

Confusing Addition & Concatenation

Addition is about adding **numbers**.
Concatenation is about adding **strings**.

In JavaScript both operations use the same + operator.

Because of this, when adding a number as a number, will produce a different result from adding a number as a string:

```
var x = 10 + 5;          // the result in x is 15
var x = 10 + "5";        // the result in x is "105"
```

When adding two variables, it can be difficult to anticipate the result:

```
var x = 10;
var y = 5;
var z = x + y;           // the result in z is 15

var x = 10;
var y = "5";
var z = x + y;           // the result in z is "105"
```

Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits Floating point numbers (Floats).
All programming languages, including JavaScript, have difficulties with precise floating point values:

```
var x = 0.1;
var y = 0.2;
var z = x + y            // the result in z will not be 0.3
if (z == 0.3)            // this if test will fail
```

Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

Example 1

```
var x =
"Hello World!";
```

But, breaking a statement in the middle of a string will not work:
Example 2

```
var x = "Hello
World!";
```

You must use a "backslash" if you must break a statement in a string:
Example 3

```
var x = "Hello \
World!";
```

Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

```
if (x == 19);
{
   // code block
}
```

Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line.
Because of this, these two examples will return the same result:

Example 1

```
function myFunction(a) {
   var power = 10
   return a * power
}
```

Example 2

```
function myFunction(a) {
   var power = 10;
   return a * power;
}
```

JavaScript will also allow you to break a statement into two lines.
Because of this, example 3 will also return the same result:

Example 3

```
function myFunction(a) {
   var
   power = 10;
   return a * power;
}
```

But, what will happen if you break the return statement in two lines like this:

Example 4

```
function myFunction(a) {
   var
   power = 10;
   return
   a * power;
}
```

The function will return undefined!
Why? Because JavaScript thinks you meant:

Example 5
```
function myFunction(a) {
    var
    power = 10;
    return;
    a * power;
}
```

Explanation
If a statement is incomplete like:

var
JavaScript will try to complete the statement by reading the next line:

power = 10;
But since this statement is complete:

return
JavaScript will automatically close it like this:

return;
This happens because closing (ending) statements with semicolon is optional in JavaScript.

JavaScript will close the return statement at the end of the line, because it is a complete statement.

Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.
Arrays with named indexes are called associative arrays (or hashes).
JavaScript does not support arrays with named indexes.

In JavaScript, arrays use numbered indexes:

Example:
```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;         // person.length will return 3
```

```
var y = person[0];            // person[0] will return "John"
```

In JavaScript, objects use named indexes.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object. After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

Example:

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;        // person.length will return 0
var y = person[0];            // person[0] will return undefined
```

```
<script>
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
document.getElementById("demo").innerHTML =
person[0] + " " + person.length;
</script>
```

RESULT:
undefined 0

Ending an Array Definition with a Comma

Incorrect:

```
points = [40, 100, 1, 5, 25, 10,];
```

Some JSON and JavaScript engines will fail, or behave unexpectedly.

Correct:

```
points = [40, 100, 1, 5, 25, 10];
```

Ending an Object Definition with a Comma

Incorrect:

```
person = {firstName:"John", lastName:"Doe", age:46,}
```

Some JSON and JavaScript engines will fail, or behave unexpectedly.

Correct:

```
person = {firstName:"John", lastName:"Doe", age:46}
```

Undefined is Not Null

With JavaScript, null is for objects, undefined is for variables, properties, and methods.
To be null, an object has to be defined, otherwise it will be undefined.
If you want to test if an object exists, this will throw an error if the object is undefined:

Incorrect:
if (myObj !== null && typeof myObj !== "undefined")
Because of this, you must test typeof() first:

Correct:
if (typeof myObj !== "undefined" && myObj !== null)

Expecting Block Level Scope

JavaScript does not create a new scope for each code block.
It is true in many programming languages, but not true in JavaScript.
It is a common mistake, among new JavaScript developers, to believe that this code returns undefined:

Example:
```
for (var i = 0; i < 10; i++) {
   // some code
}
return i;
```

JavaScript Style Guide and Coding Conventions

Coding conventions are **style guidelines for programming**. They typically cover:

- Naming and declaration rules for variables and functions.
- Rules for the use of white space, indentation, and comments.
- Programming practices and principles

Coding conventions **secure quality**:

- Improves code readability
- Make code maintenance easier

Coding conventions can be documented rules for teams to follow, or just be your individual coding practice.

Variable Names

Use **camelCase** for identifier names (variables and functions).
All names start with a **letter**.

```
firstName = "John";
lastName = "Doe";

price = 19.90;
tax = 0.20;
fullPrice = price + (price * tax);
```

Spaces Around Operators

Always put spaces around operators (= + / *), and after commas:
Examples:

```
var x = y + z;
var values = ["Volvo", "Saab", "Fiat"];
```

Code Indentation

Always use 4 spaces for indentation of code blocks:
Functions:

```
function toCelsius(fahrenheit) {
   return (5/9) * (fahrenheit-32)
```

Do not use tabs (tabulators) for indentation. Text editors interpret tabs differently.

Statement Rules

General rules for simple statements:

- Always end simple statement with a semicolon.

Examples:

```
var values = ["Volvo", "Saab", "Fiat"];

var person = {
        firstName: "John",
        lastName: "Doe",
        age: 50,
        eyeColor: "blue"
};
```

General rules for complex (compound) statements:

- Put the opening bracket at the end of the first line.
- Use one space before the opening bracket.
- Put the closing bracket on a new line, without leading spaces.
- Do not end complex statement with a semicolon.

Functions:

```
function toCelsius(fahrenheit) {
   return (5/9) * (fahrenheit-32);
}
```

Loops:

```
for (i = 0; i < 5; i++) {
        x += i;
}
```

Conditionals:

```
if (time < 20) {
        greeting = "Good day";
} else {
        greeting = "Good evening";
}
```

Object Rules

General rules for object definitions:

- Place the opening bracket on the same line as the object name.
- Use colon plus one space between each property and it's value.
- Use quotes around string values, not around numeric values.
- Do not add a comma after the last property-value pair.
- Place the closing bracket, on a new line, without leading spaces.
- Always end an object definition with a semicolon.

Example:

```
var person = {
        firstName: "John",
        lastName: "Doe",
        age: 50,
        eyeColor: "blue"
};
```

Short objects can be written compressed, on one line, like this:

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Line Length < 80

For readability, avoid lines longer than 80 characters.
If a JavaScript statement does not fit on one line, the best place to break it, is after an operator or a comma.
Example

```
document.getElementById("demo").innerHTML =
   "Hello Dolly.";
```

Naming Conventions
Always use the same naming convention for all your code. For example:

- Variable and function names written as **camelCase**
- Global variable written in **UPPERCASE**
- Constants (like PI) written in **UPPERCASE**

Should you use **hyp-hens**, **camelCase**, or **under_scores** in variable names?
This is a question programmers often discuss. The answer depends on who you ask:

Hyphens in HTML and CSS:

HTML5 attributes can start with data- (data-quantity, data-price).

CSS uses hyphens in property-names (font-size).

Underscores:
Many programmers prefer to use underscores (date_of_birth), especially in SQL databases.
Underscores are often used in PHP documentation.

PascalCase:
PascalCase is often preferred by C programmers.

camelCase:
camelCase is used by JavaScript itself, by jQuery, and other JavaScript libraries.

Don't start names with a $ sign. It will put you in conflict with many JavaScript library names.

Loading JavaScript in HTML
Use simple syntax for loading external scripts (the type attribute is not necessary):
<script src="myscript.js">

Accessing HTML Elements
A consequence of using "untidy" HTML styles, might result in JavaScript errors.
These two JavaScript statements will produce different results:
var obj = getElementById("Demo")

var obj = getElementById("demo")
If possible, use the same naming convention (as JavaScript) in HTML.

File Extensions
HTML files should have a **.html** extension (not **.htm**).
CSS files should have a **.css** extension.
JavaScript files should have a **.js** extension.

Use Lower Case File Names
Most web servers (Apache, Unix) are case sensitive about file names:
london.jpg cannot be accessed as London.jpg.
Other web servers (Microsoft, IIS) are not case sensitive:
london.jpg can be accessed as London.jpg or london.jpg.
If you use a mix of upper and lower case, you have to be extremely consistent.
If you move from a case insensitive, to a case sensitive server, even small errors will break your web.
To avoid these problems, always use lower case file names (if possible).

JavaScript JSON
JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

What is JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is lightweight data interchange format
- JSON is language independent *
- JSON is "self-describing" and easy to understand

JSON Example
This JSON syntax defines an employees object: an array of 3 employee records (objects):
JSON Example

```
{"employees":[
        {"firstName":"John", "lastName":"Doe"},
        {"firstName":"Anna", "lastName":"Smith"},
        {"firstName":"Peter", "lastName":"Jones"}
]}
```

The JSON Format Evaluates to JavaScript Objects

The JSON format is syntactically identical to the code for creating JavaScript objects.
Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, Just like JavaScript object properties.
A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:
"firstName":"John"

JSON Objects

JSON objects are written inside curly braces.
Just like in JavaScript, objects can contain multiple name/values pairs:
{"firstName":"John", "lastName":"Doe"}

JSON Arrays

JSON arrays are written inside square brackets.
Just like in JavaScript, an array can contain objects:

```
"employees":[
        {"firstName":"John", "lastName":"Doe"},
```

```
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

In the example above, the object "employees" is an array. It contains three objects.
Each object is a record of a person (with a first name and a last name).

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.
For simplicity, this can be demonstrated using a string as input.
First, create a JavaScript string containing JSON syntax:

```
var text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function JSON.parse() to convert the string into a JavaScript object:

```
var obj = JSON.parse(text);
```

Finally, use the new JavaScript object in your page:
Example

```
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>
```

JavaScript Performance

Reduce Activity in Loops

Loops are often used in programming.
Each statement in a loop, including the for statement, is executed for each iteration of the loop.
Search for statements or assignments that can be placed outside the loop.
Bad Code:

```
for i = 0; i < arr.length; i++) {
```

Better Code:

```
l = arr.length;
for i = 0; i < l; i++) {
```

The better code access the length property outside the loop, and makes the loop run faster.

Reduce DOM Access

Accessing the HTML DOM is very slow, compared to other JavaScript statements.
If you expect to access a DOM element several times, access it once, and use it as a local variable:
Example

```
obj = document.getElementById("demo");
obj.innerHTML = "Hello";
```

Avoid Unnecessary Variables

Don't create new variables if you don't plan to save values.
Often you can replace code like this:

```
var fullName = firstName + " " + lastName;
document.getElementById("demo").innerHTML = fullName;
```

With this:

```
document.getElementById("demo").innerHTML = firstName + " " + lastName
```

Apéndice A

Variables globales

La peor de todas las malas características de JavaScript es su dependencia de las variables globales. Hay tres maneras de definir las variables globales. La primera es la de colocar una declaración var fuera de cualquier función:

```
var foo = valor;
```

El segundo consiste en añadir una propiedad directamente al objeto global. El objetivo global es el contenedor de todas las variables globales. En los navegadores web, el objeto global va por la ventana del nombre:

```
window.foo = valor;
```

La tercera es usar una variable sin declararlo. Esto se llama implícita globales:

```
foo = valor;
```

Esto fue pensada como una conveniencia para los principiantes por lo que es necesario declarar las variables antes de usarlas. Desafortunadamente, olvidando que declarar una variable es un error muy común. La política de JavaScript de hacer las variables olvidadas global crea errores que pueden ser muy difíciles de encontrar.

Scope

La sintaxis de JavaScript viene de C. En todos los demás lenguajes de programación como C, un bloque (un conjunto de estados envueltos entre llaves) crea un ámbito. Las variables declaradas en un bloque que no son visibles fuera del bloque.
Es mejor que declarar todas las variables en la parte superior de cada función.

Semicolon Insertion

JavaScript tiene un mecanismo que intenta corregir los programas defectuosos por punto y coma insertar automáticamente. No dependen de esto. Esto puede enmascarar los errores más graves. A veces inserta un punto y coma en lugares donde no son bienvenidos. Tenga en cuenta las consecuencias de la inserción punto y coma en la sentencia return. Si una sentencia return devuelve un valor, que la expresión de valor debe comenzar en la misma línea que el retorno:

```
return { status: true };
```

Esto parece devolver un objeto que contiene un miembro de estado. Por desgracia, la inserción y coma lo convierte en una sentencia que devuelve indefinido. No hay ninguna advertencia que punto y coma inserción causó la mala interpretación del programa. El problema se puede evitar si el {se coloca al final de la línea anterior y no al principio de la línea siguiente:

```
return {status: true }
```

Unicode

JavaScript fue diseñado en un momento en que se espera Unicode tener como máximo 65.536 caracteres. Desde entonces ha crecido para tener una capacidad de más de 1 millón de caracteres. Los personajes de JavaScript son 16 bits. Eso es suficiente para cubrir el original 65536 (que ahora se conoce como el plano básico multilingüe). Cada uno de los millones de caracteres restantes se puede representar como un par de caracteres. Unicode considera el par a ser un único carácter. JavaScript piensa la pareja es de dos personajes distintos.

Typeof

El operador typeof devuelve una cadena que identifica el tipo de su operando. Por lo tanto:

```
typeof 98.6 produces 'number'.
```

Desafortunadamente:
typeof nulos devuelve 'objeto' en vez de 'nulo'. Ups. Una mejor prueba para nula es simplemente:

```
my_value === null
```

Un problema mayor es la prueba de un valor para objetualidad. typeof no pueden distinguir entre nulo y objetos, pero usted puede porque nula es falso y todos los objetos son verdaderos:

```
if (my_value && typeof my_value === 'object') {    // my_value is an object or an array! }
```

ParseInt

ParseInt es una función que convierte una cadena en un número entero.
El operador + puede agregar o concatenar. Cuál lo hace depende de los tipos de los parámetros. Si alguno de los operandos es una cadena vacía, produce el otro operando convierte en una cadena. Si ambos operandos son números, que produce la suma. De lo contrario, convierte ambos operandos de cadenas y las concatena. Este comportamiento complicado es una fuente común de errores. Si tiene la intención + añadir, asegúrese de que ambos operandos son números.

Floating Point

Números de punto flotante binarios son ineptos en el manejo de las fracciones decimales, por lo que 0.1 + 0.2 no es igual a 0,3.
Afortunadamente, la aritmética de enteros en coma flotante es exacta, por lo que los errores de representación decimales se puede evitar mediante la ampliación. Por ejemplo, los valores en dólares pueden ser convertidos a valores enteros centavos multiplicándolos por 100. La gente tiene una expectativa razonable cuando se cuentan el dinero que los resultados serán exactos.

NaN

El valor NaN representa no es un número, a pesar de que:

```
typeof NaN === 'number'  // true
```

El valor se puede producir al intentar convertir una cadena en un número cuando la cadena no está en la forma de un número. Por ejemplo:

```
+ '0'       // 0
+ 'oops'    // NaN
```

Si es un operando en una operación aritmética, entonces NaN será el resultado. Por lo tanto, si usted tiene una cadena de fórmulas que producen NaN como resultado, al menos una de las entradas era NaN, NaN o fue generada en alguna parte. Puede probar NaN. Como hemos visto, typeof no distingue entre los números y NaN, y resulta que NaN no es igual a sí mismo. Así, sorprendentemente:

```
NaN === NaN    // false
NaN !== NaN    // true
```

Phony Arrays

JavaScript no tiene matrices reales. Eso no es del todo malo. Matrices de JavaScript son muy fácil de usar. No hay necesidad de darles una dimensión, y nunca generar errores outof-grada. Sin embargo, su rendimiento puede ser considerablemente peor que las matrices reales.

hasOwnProperty

El método hasOwnProperty se ofrece como un filtro para evitar un problema con el en el comunicado. Desafortunadamente, hasOwnProperty es un método, no un operador, por lo que en cualquier objeto que pueda ser reemplazado con una función diferente o incluso un valor que no es una función de:

```
var name;
another_stooge.hasOwnProperty = null;       // trouble
for (name in another_stooge) {
        if (another_stooge.hasOwnProperty(name)) { // boom
                document.writeln(name + ': ' + another_stooge[name]);
        }
}
```

Objeto

Objetos de JavaScript nunca son verdaderamente vacío, ya que pueden recoger a los miembros de la cadena de prototipo. A veces lo que importa. Por ejemplo, supongamos que usted está escribiendo un programa que cuenta el número de ocurrencias de cada palabra en un texto. Podemos utilizar el método toLowerCase para normalizar el texto a minúsculas, y luego usar el método split con una expresión regular para producir una serie de palabras.

Capítulo 3

Objects

Los literales de objetos

Los literales de objetos proporcionan una notación muy conveniente para la creación de nuevos valores de objeto. Un objeto literal es un par de llaves que rodean cero o más pares nombre / valor. Un literal objeto puede aparecer en cualquier lugar puede parecer una expresión:

```
var empty_object = {};
var stooge = {
        "first-name": "Jerome",
        "last-name": "Howard"
 };
```

El nombre de una propiedad puede ser cualquier cadena, incluyendo la cadena vacía. Las comillas en el nombre de una propiedad de un objeto literal son opcionales si el nombre sería un nombre legal JavaScript y no una palabra reservada. Así que se requieren comillas alrededor de "nombre", pero son opcionales alrededor first_name. Las comas se utilizan para separar los pares.

Recuperación

Los valores se pueden recuperar de un objeto envolviendo una expresión de cadena en un [] sufijo. Si la expresión de cadena es una constante, y si es un nombre legal JavaScript y no una palabra reservada, entonces el. notación se puede utilizar en su lugar. El. se prefiere la notación porque es más compacto y se lee mejor:

Update

Un valor de un objeto puede ser actualizado por asignación. Si el nombre de la propiedad ya existe en el objeto, el valor de la propiedad se sustituye.

Reference

Objetos se pasan alrededor por referencia. Nunca se copian.

Prototype

Cada objeto está vinculado a un objeto prototipo de la que puede heredar propiedades. Todos los objetos creados a partir de objetos literales están vinculados a Object.prototype, un objeto que viene de serie con JavaScript. Cuando usted hace un nuevo objeto, puede seleccionar el objeto que debe ser su prototipo. El mecanismo que proporciona JavaScript de hacer esto es desordenado y complejo, pero se puede simplificar de manera significativa. Vamos a añadir un método create a la función del objeto. El método create crea un nuevo objeto que utiliza un objeto antiguo como su prototipo.

Reflection

Es fácil de inspeccionar un objeto para determinar qué propiedades tiene al tratar de recuperar las propiedades y el examen de los valores obtenidos. El operador typeof puede ser muy útil para determinar el tipo de una propiedad.

Enumeration

El en el comunicado puede bucle sobre todos los nombres de las propiedades de un objeto. La enumeración incluirá todas las propiedades, incluyendo funciones y propiedades de prototipo que podría no estar interesados en lo que es necesario para filtrar los valores que no desea. Los filtros más comunes son el método hasOwnProperty y el uso de typeof para excluir funciones.

Delete

El operador de eliminación se puede utilizar para eliminar una propiedad de un objeto. Se eliminará una propiedad del objeto, si lo tiene. No va a tocar cualquiera de los objetos en el vínculo prototipo.

Global Abatement

JavaScript hace que sea fácil de definir variables globales que pueden contener todos los activos de su aplicación. Desafortunadamente, las variables globales debilitan la resistencia de programas y deben evitarse. Una forma de minimizar el uso de variables globales es crear una única variable global para su aplicación:

```
var myapp= {};
```

Capítulo 4

Functions

Se utilizan para la reutilización de código, ocultación de información, y la composición.Las funciones se utilizan para especificar el comportamiento de los objetos.

Function Objects

Funciones en JavaScript son objetos.
Los objetos son colecciones de pares name/value que tengan un vínculo oculto a un objeto prototipo.
Objetos producidos a partir de objetos literales están vinculados a Object.prototype.
Objetos de función están vinculados a Function.prototype (que está a su vez vinculado a Object.prototype).
Cada función también se crea con dos propiedades adicionales ocultos:

- Contexto de la función
- Y el código que implementa el comportamiento de la función.

Cada objeto función también se crea con una propiedad de prototipo. Su valor es un objeto con una propiedad constructor cuyo valor es la función. Esto es distinto de el enlace oculto para Function.prototype. Dado que las funciones son objetos, se pueden utilizar como cualquier otro valor. Las funciones pueden ser almacenados en las variables, objetos y arrays. Las funciones se pueden pasar como argumentos a funciones, y funciones pueden ser devueltos por funciones. Además, dado que las funciones son objetos, las funciones se tienen métodos. Lo que es especial acerca de las funciones es que pueden ser invocados.

Función Literal

Objetos de función se crean con literales de función:

```
// Crear una variable llamada complemento y almacenar una función
// En lo que suma dos números.
        var add = function (a, b) {
                        return a + b;
        };
```

Un literal de función tiene cuatro partes. La primera parte es la función de palabra reservada. La segunda parte opcional es el nombre de la función. La función se puede utilizar su nombre para llamarse a sí mismo de forma recursiva. El nombre también puede ser utilizado por los depuradores y herramientas de desarrollo para identificar la función. Si una función no se le da un nombre, como se muestra en el ejemplo anterior, se dice ser anónimo. La tercera parte es el conjunto de parámetros de la función, envueltos en paréntesis.

Invocation

Invocar una función suspende la ejecución de la función actual, pasando de control y parámetros para la nueva función. Además de los parámetros declarados, cada función recibe dos parámetros adicionales: this y arguments. El este parámetro es muy importante en la programación orientada a objetos, y su valor se determina por el patrón de invocación. Existen cuatro patrones de invocación en JavaScript: el patrón

de invocación de métodos, el patrón de invocación de la función, el patrón de invocación del constructor, y el patrón de invocación se aplican. Los patrones difieren en cómo el parámetro this se inicializa.

The Method Invocation Pattern

Cuando una función se almacena como una propiedad de un objeto, lo llamamos un método. Cuando se invoca un método, esto está ligado a ese objeto. Si una expresión de invocación contiene un refinamiento (. Es decir, una expresión de puntos o [índice] expresión), 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
```

The Function Invocation Pattern

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

```
 suma var = sumar (3, 4);
// Suma es 7
```

Cuando se invoca una función con este patrón, esto se enlaza con el objeto global

```
 // 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.getValue());    // 6
```

The Constructor Invocation Pattern

JavaScript es un lenguaje de herencias de prototipos. Eso significa que los objetos pueden heredar propiedades directamente de otros objetos. El lenguaje es de clase gratis. Se trata de un cambio radical de la moda actual. La mayoría de los lenguajes de hoy son clásicos. Herencias de prototipos es poderosamente expresivo, pero no es ampliamente entendido. Sí JavaScript no confía en su naturaleza prototípica, por lo que ofrece una sintaxis objeto de decisiones que es una reminiscencia de las lenguas clásicas. Pocos programadores clásicos encontraron herencias de prototipos para ser aceptable, y la sintaxis de inspiración clásica oscurece verdadera naturaleza prototípica de la lengua. Es lo peor de ambos mundos. Si una función se invoca con el nuevo prefijo, a continuación, un nuevo objeto se crea con un vínculo oculto por el importe de miembro prototipo de la función, y esto estará vinculado a ese nuevo objeto. El nuevo prefijo también cambia el comportamiento de la instrucción de retorno. Vamos a ver más de eso después.

La recursividad

Una función recursiva es una función que llama a sí misma, ya sea directa o indirectamente. La recursión es una técnica de programación de gran alcance en el que un problema se divide en un conjunto de subproblemas similares, cada uno resuelto con una solución trivial. En general, una función recursiva se llama a resolver sus subproblemas.

Scope(Alcance)

Alcance en un lenguaje de programación controla la visibilidad y tiempos de vida de las variables y parámetros. Este es un servicio importante para el programador, ya que reduce las colisiones de nombres y proporciona gestión automática de memoria:

Cierre

La buena noticia sobre el alcance es que las funciones internas tienen acceso a los parámetros y variables de las funciones que están definidas dentro (con la excepción de esto y argumentos). Esto es una cosa muy buena. Nuestra función getElementsByAttribute trabajado, por haber declarado una variable resultados, y la función interna que pasó a walk_the_DOM también tenía acceso a la variable de resultados. Un caso más interesante es cuando la función de interior tiene un tiempo de vida más largo que su función externa. Más temprano, hicimos una miObjeto que tenía un valor y un método de incremento. Supongamos que queremos proteger el valor de cambios no autorizados. En vez de inicializar miObjeto con un objeto literal, vamos a inicializar miObjeto llamando a una función que devuelve un objeto literal. Esa función define una variable de valor.

Las devoluciones de llamada

Las funciones pueden hacer que sea más fácil lidiar con eventos discontinuos. Por ejemplo, supongamos que hay una secuencia que comienza con la interacción del usuario, haciendo una petición del servidor, y finalmente mostrar la respuesta del servidor.

Módulo

Podemos utilizar las funciones y cierre para hacer módulos. Un módulo es una función u objeto que presenta una interfaz pero que esconde su estado y la aplicación. Mediante el uso de funciones para producir módulos, podemos eliminar casi por completo el uso de variables globales, mitigando de esta

manera una de las peores características de JavaScript. Por ejemplo, supongamos que queremos aumentar la secuencia con un método deentityify. Su trabajo consiste en buscar las entidades HTML en una cadena y reemplazarlos con sus equivalentes. Tiene sentido mantener los nombres de las entidades y sus equivalentes en un objeto. Pero ¿dónde debemos mantener el objeto? Podríamos ponerlo en una variable global, pero las variables globales son malas. Podríamos definirlo en la función en sí, sino que tiene un costo de tiempo de ejecución porque el literal debe ser evaluado cada vez que se invoca la función

Cascada

Algunos métodos no tienen un valor de retorno. Por ejemplo, es habitual que los métodos que establecen o modifican el estado de un objeto a devolver nada. Si tenemos estos métodos devuelven esto en vez de definir, podemos permitir a las cascadas. En una cascada, podemos llamar a muchos métodos en el mismo objeto en secuencia en una sola sentencia.

Memoization

Las funciones se pueden usar objetos para recordar los resultados de las operaciones anteriores, por lo que es posible evitar trabajo innecesario. Esta optimización se llama memoization. Objetos y arrays de JavaScript son muy conveniente para esto.

JavaScript Errors - Throw and Try to Catch

La sentencia try le permite probar un bloque de código para los errores.
La sentencia catch le permite manejar el error.
La sentencia throw permite crear errores personalizados.
La sentencia finally permite ejecutar código, después de tratar de atrapar, sin importar el resultado.

Los errores sucederán!
Cuando se ejecuta el código JavaScript, pueden ocurrir errores diferentes.

Los errores pueden ser los errores cometidos por el programador, los errores debidos a la entrada equivocada, y otras cosas imprevisibles codificación

Capítulo 5

Herencia.

La herencia es un tema importante en la mayoría de los lenguajes de programación. En las lenguas clásicas (como Java), la herencia (o se extiende) proporciona dos servicios útiles. En primer lugar, es una forma de reutilización de código. Si una nueva clase es principalmente similar a una clase existente, sólo tiene que especificar las diferencias. Los patrones de reutilización de código son extremadamente importantes porque tienen el potencial de reducir significativamente el coste de desarrollo de software. La otra ventaja de la herencia clásica es que incluye la especificación de un sistema de tipos. Esto libera sobre todo al programador de tener que escribir las operaciones de fundición explícitas, que es una cosa muy buena porque cuando se lanza, se pierden los beneficios de seguridad de un sistema de tipos.

JavaScript es un lenguaje de programación relajado escrito, nunca arroja. El linaje de un objeto es irrelevante. Lo importante de un objeto es lo que puede hacer, no lo que es descendiente de. JavaScript proporciona un conjunto mucho más rico de patrones de reutilización de código. Puede imitar el patrón clásico, pero también es compatible con otros modelos que son más expresivos. El conjunto de posibles patrones de herencia en JavaScript es enorme. En este capítulo, vamos a ver algunos de los modelos más sencillos. Mucho más complicado construcciones son posibles, pero por lo general es mejor que sea sencillo. En las lenguas clásicas, los objetos son instancias de clases, y una clase puede heredar de otra clase. JavaScript es un lenguaje de prototipos, lo que significa que los objetos heredan directamente de otros objetos

Pseudoclassical

JavaScript está en conflicto acerca de su carácter prototípico. Su mecanismo prototipo está oscurecida por una complicada empresa sintáctica que se parece vagamente clásico. En lugar de tener objetos heredan directamente de otros objetos, un nivel innecesario de indirección se inserta de manera que los objetos son producidos por las funciones constructoras. Cuando se crea un objeto función, el constructor Función que produce el objeto función se ejecuta algún código como este:
this.prototype = {constructor: this}; El nuevo objeto de función se da una propiedad prototipo cuyo valor es un objeto que contiene una propiedad constructor cuyo valor es el nuevo objeto de función. El objeto prototipo es el lugar donde los rasgos hereditarios son a depositar. Cada función obtiene un objeto prototipo porque el lenguaje no proporciona una manera de determinar qué funciones están destinados a ser utilizados como constructores. La propiedad constructor no es útil. Es el objeto prototipo que es importante. Cuando se invoca una función con el patrón de invocación del constructor utilizando el nuevo prefijo, esto modifica la forma en que se ejecuta la función.

Object Specifiers

A veces sucede que un constructor se le da un número muy grande de parámetros. Esto puede ser problemático porque puede ser muy difícil de recordar el orden de los argumentos. En tales casos, puede ser mucho más amigable si escribimos el constructor para aceptar un único objeto especificador lugar. Ese objeto contiene la especificación del objeto a ser construido.

Prototypal

En un patrón puramente prototípico, prescindimos de clases. Nos concentramos en cambio en los objetos. Herencias de prototipos es conceptualmente más simple que la herencia clásica: un nuevo objeto puede heredar las propiedades de un objeto antiguo. Esta es tal vez desconocida, pero es muy fácil de entender. Se empieza por hacer un objeto útil.

Funcional

Una de las debilidades de los patrones de herencia que hemos visto hasta ahora es que no recibimos ninguna privacidad.Todas las propiedades de un objeto son visibles. Llegamos sin variables privadas y no hay métodos privados. A veces, eso no importa, pero a veces importa mucho. En la frustración, algunos programadores desinformadas han adoptado un patrón de privacidad de simulación. Si tienen una propiedad que desean hacer privado, que le dan un nombre oddlooking, con la esperanza de que los

demás usuarios del código pretenderán que no pueden ver a los miembros que buscan impares.Afortunadamente, tenemos una alternativa mucho mejor en una aplicación del modelo de módulo. Empezamos haciendo una función que va a producir objetos. Vamos a darle un nombre que comienza con una letra minúscula, ya que no será necesario el uso del nuevo prefijo. La función consta de cuatro pasos:

1. Se crea un nuevo objeto. Hay un montón de maneras de hacer que un objeto. Puede hacer que un objeto literal, o puede llamar a una función constructora con el nuevo prefijo, o se puede utilizar el método Object.create hacer una nueva instancia de un objeto existente, o puede llamar a cualquier función que devuelve un objeto.
2. Define opcionalmente variables de instancia y métodos privados. Estos son sólo vars ordinarias de la función.
3. Se aumenta ese nuevo objeto con métodos. Esos métodos tendrán un acceso privilegiado a los parámetros y las vars definidas en el segundo paso.
4. Se devuelve ese nuevo objeto. Aquí está una plantilla pseudocódigo para un constructor funcional.

Regiones

Podemos componer objetos de conjuntos de piezas. Por ejemplo, podemos hacer una función que puede agregar características simples de procesamiento de eventos a cualquier objeto. Añade una sobre el método, un método de fuego, y un registro de evento privado.

Apéndice B

Bad parts

With Statement

JavaScript has a with statement that was intended to provide a shorthand when accessing the properties of an object. Unfortunately, its results can sometimes be unpredictable, so it should be avoided.

The statement:

```
with (obj) {
    a = b;
}
```

does the same thing as:

```
if (obj.a === undefined) {
    a = obj.b === undefined ? b : obj.b;
} else {
    obj.a = obj.b === undefined ? b : obj.b;
}
```

So, it is the same as one of these statements:

```
a = b;
a = obj.b;
obj.a = b;
obj.a = obj.b;
```

It is not possible to tell from reading the program which of those statements you will get. It can vary from one running of the program to the next. It can even vary while the program is running. If you can't read a program and understand what it is going to do, it is impossible to have confidence that it will correctly do what you want.

eval

The eval function passes a string to the JavaScript compiler and executes the result. It is the single most misused feature of JavaScript. It is most commonly used by people who have an incomplete understanding of the language. For example, if you know about the dot notation, but are ignorant of the subscript notation, you might write:

```
eval("myValue = myObject." + myKey + ";");
```

instead of:

```
myvalue = myObject[myKey];
```

The eval form is much harder to read. This form will be significantly slower because it needs to run the compiler just to execute a trivial assignment statement. The eval function also compromises the security of your application because it grants too much authority to the eval'd text. And it compromises the performance of the language as a whole in the same way that the with statement does.

Continue Statement

The continue statement jumps to the top of the loop. I have never seen a piece of code that was not improved by refactoring it to remove the continue statement.

Switch Fall Through

La sentencia switch fue modelado después del FORTRAN IV computarizada ir a la declaración. Cada caso entra a través en el siguiente caso a menos que interrumpes explícitamente el flujo. Alguien me escribió una vez lo que sugiere que JSLint debe dar un aviso cuando pasa al siguiente caso en otro caso. Señaló que se trata de una fuente muy común de los errores, y es un error difícil de ver en el código. Le respondí que eso era todo cierto, pero que el beneficio de compacidad obtenida por la caída a través de más que compensado por la posibilidad de error. Al día siguiente, se informó que hubo un error en JSLint. Se misidentifying un error. Investigué, y resultó que tenía un caso que estaba cayendo a través. En ese momento, he logrado la iluminación. Ya no uso through intencionales otoño. Esa disciplina hace que sea mucho más fácil encontrar los pasantes no intencionales de la caída. Los peores características de un

lenguaje no son las características que son obviamente peligroso o inútil. Esos son fácilmente evitados. Los peores características son las molestias atractivos, las características que son a la vez útil y peligroso.

Block-less Statements

Un caso o mientras, hacer o para la declaración puede tener un bloque o una sola sentencia. El formulario de declaración solo es otra molestia atractiva. Ofrece la ventaja de ahorrar dos personajes, una ventaja dudosa. Se oscurece la estructura del programa para que los manipuladores posteriores del código pueden insertar fácilmente los insectos.

++ --

The increment and decrement operators make it possible to write in an extremely terse style. In languages such as C, they made it possible to write one-liners that could do string copies:

```
for (p = src, q = dest; !*p; p++, q++) *q = *p;
```

They also encourage a programming style that, as it turns out, is reckless. Most of the buffer overrun bugs that created terrible security vulnerabilities were due to code like this.
In my own practice, I observed that when I used ++ and --, my code tended to be too tight, too tricky, too cryptic. So, as a matter of discipline, I don't use them any more. I think that as a result, my coding style has become cleaner.

Bitwise Operators

JavaScript has the same set of bitwise operators as Java:

```
&   and
|   or
^   xor
~   not
>>  signed right shift
>>> unsigned right shift
<<  left shift
```

Capítulo 6

Arrays

Los arreglos literales proporcionan una notación muy conveniente para la creación de nuevos valores de la matriz. Un literal de matriz es un par de corchetes cero o más valores separados por comas. Un literal de matriz puede aparecer en cualquier lugar puede parecer una expresión.

Length

Cada array tiene una propiedad length. A diferencia de otros idiomas, longitud de la matriz de JavaScript no es un límite superior. Si almacena un elemento con un subíndice que es mayor que o igual a la longitud

actual, la longitud aumentará para contener el nuevo elemento. No hay error límites de la matriz. La propiedad length es el mayor nombre de la propiedad de entero en la matriz más uno.

Delete

delete numbers[2]; //
numbers is ['zero', 'one', undefined, 'shi', 'go']

==========

numbers.splice(2, 1);
// numbers is ['zero', 'one', 'shi', 'go']

Enumeration

Desde matrices de JavaScript son realmente objetos, el de en la declaración se puede utilizar para iterar sobre todas las propiedades de una matriz. Por desgracia, en el que hace ninguna garantía sobre el orden de las propiedades, y la mayoría de las aplicaciones de arreglo esperan que los elementos que se producen en orden numérico. Además, todavía existe el problema con propiedades inesperadas que se dragados arriba de la cadena de prototipo.

Confusion

Un error común en los programas JavaScript es utilizar un objeto cuando se requiere una matriz o una matriz cuando se requiere un objeto. La regla es simple: cuando los nombres de propiedad son pequeños números enteros consecutivos, se debe utilizar una matriz. De lo contrario, utilice un objeto. Sí JavaScript está confundido acerca de la diferencia entre arrays y objetos. El operador typeof informa que el tipo de una matriz es "objeto", que no es muy útil.

Methods

JavaScript proporciona un conjunto de métodos para actuar sobre arrays. Los métodos son funciones almacenadas en Array.prototype. En el capítulo 3 vimos que Object.prototype puede ser aumentado. Array.prototype se puede aumentar también.

Dimensions

Arrays de JavaScript por lo general no se inicializan. Si usted pide un nuevo array con [], que estará vacío. Si accede a un elemento que falta, obtendrá el valor undefined. Si usted es consciente de ello, o si se quiere, naturalmente, establecer todos los elementos antes de intentar recuperarlo, entonces todo está bien. Pero si va a implementar algoritmos que asumen que todos los elementos se inicia con un valor conocido (como 0), entonces usted debe preparar el array usted mismo.

Capítulo 8

Métodos

Array

array.concat(item…)
El método concat produce un nuevo array conteniendo una copia superficial de esta array con los elementos añadidos al final. Si un elemento es una array, entonces cada uno de sus elementos se adjunta por separado.

array.join(separator)
El método join hace una cadena de una matriz. Para ello, hacer una cadena de cada uno de los elementos de la matriz y, a continuación, concatenar todos juntos con un separador entre ellos. El separador predeterminado es ','. Para unirse sin separación, utilice una cadena vacía como el separador. Si está montando una cadena de un gran número de piezas, por lo general es más rápido que poner las piezas en una matriz y unirse a ellos de lo que es para concatenar las piezas con el operador +

array.sort(comparefn)
El método sort ordena el contenido de una matriz en su lugar. Ordena matrices de números incorrectamente:

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// n is [15, 16, 23, 4, 42, 8]
```

Función de comparación por defecto de JavaScript asume que los elementos a ser ordenados son cadenas. No es lo suficientemente inteligente como para probar el tipo de los elementos antes de compararlos, lo que convierte los números a cadenas como las compara, asegurando un resultado sorprendentemente incorrecta. Afortunadamente, es posible reemplazar la función de comparación con el suyo propio. Su función de comparación debe tener dos parámetros y devuelve 0 si los dos parámetros son iguales, un número negativo si el primer parámetro debe ser lo primero, y un número positivo si el segundo parámetro debe ser lo primero. (Los veteranos podrían ser recordados de la aritmética FORTRAN II IF).

array.splice(start,deleteCount,item…)
El método de empalme elimina elementos de una matriz, reemplazándolos con nuevos elementos. El parámetro de inicio es el número de una posición dentro de la matriz. El parámetro deleteCount es el número de elementos a eliminar a partir de esa posición. Si hay parámetros adicionales, esos elementos se insertarán en la posición. Devuelve una matriz que contiene los elementos eliminados. El uso más popular de empalme es borrar elementos de una matriz. No se debe confundir con la rebanada de empalme:

```
var a = ['a', 'b', 'c'];
var r = a.splice(1, 1, 'ache', 'bug');
```

```
// a is ['a', 'ache', 'bug', 'c']
// r is ['b']
```

array.unshift(item…)

El método unshift es como el método push excepto que empuja a los elementos al frente de esta matriz en lugar de al final. Devuelve la nueva longitud de la matriz:

Funciones

function.apply(thisArg,argArray)

El método apply invoca una función, pasando el objeto que se une a esto y un conjunto opcional de argumentos. El método de aplicación se utiliza en el patrón de aplicar invocación

Number

number.toExponential(fractionDigits)

El método apply invoca una función, pasando el objeto que se une a esto y un conjunto opcional de argumentos. El método de aplicación se utiliza en el patrón de aplicar invocación.

number.toFixed(fractionDigits)

El método toFixed convierte este número en una cadena en forma decimal. El parámetro fractionDigits opcional controla el número de decimales. Debe estar entre 0 y 20. El valor predeterminado es 0

number.toPrecision(precision)

El método toPrecision convierte este número en una cadena en forma decimal. El parámetro de precisión opcional controla el número de dígitos de precisión.

number.toString(radix)

El método toString convierte este número en una cadena. El parámetro opcional radix controla radix, o base. Debe estar entre 2 y 36. El valor predeterminado es radix base 10. El parámetro radix se utiliza más comúnmente con números enteros, pero se puede utilizar en cualquier número.

Object

object.hasOwnProperty(name)

El método hasOwnProperty devuelve true si el objeto contiene una propiedad que tiene el nombre. No se examina la cadena de prototipo. Este método es inútil si el nombre es hasOwnProperty

String

string.charAt(pos)

El método charAt devuelve el carácter en la posición pos en esta cadena. Ifpos es menor que cero o mayor que o igual a String.length, devuelve la cadena vacía. JavaScript no tiene un tipo de carácter. El resultado de este método es una cadena

string.charCodeAt(pos)

El método charCodeAt es el mismo que charAt excepto que en lugar de devolver una cadena, se devuelve una representación entera del valor de punto de código del carácter en la posición pos de esa cadena. Si pos es menor que cero o mayor que o igual a string.length, devuelve NaN

string.concat(string…)

El método concat hace una nueva cadena concatenando otras cadenas juntas. Se usa rara vez debido a que el operador + es más conveniente

string.indexOf(searchString,position)

El método indexOf busca una searchstring dentro de una cadena. Si se encuentra, devuelve la posición del primer carácter emparejado; de lo contrario, devuelve -1. El parámetro opcional posición hace que la búsqueda para comenzar en alguna posición especificada en la cadena

string.lastIndexOf(searchString,position)

El método lastIndexOf es como el método indexOf, excepto que busca desde el extremo de la cadena en lugar de la parte delantera

string.localeCompare(that)

El método localCompare compara dos cadenas. No se especifican las reglas de cómo se comparan las cuerdas. Si esta cadena es menor que la cadena, el resultado es negativo. Si son iguales, el resultado es cero. Esto es similar a la convención para la función de comparación array.sort

string.match(regexp)

El método partido coincide con una cadena y una expresión regular. ¿Cómo hace esto depende del indicador g. Si no hay ningún indicador g, entonces el resultado de la llamada String.match (regexp) es el mismo que llamar regexp.exec (string). Sin embargo, si la expresión regular tiene el indicador g, entonces se produce una matriz de todos los partidos, pero excluye a los grupos de captura

string.replace(searchValue,replaceValue)

El método replace hace una búsqueda y reemplazar operación en esta cadena, la producción de una nueva cadena. El argumento searchValue puede ser una cadena o un objeto de expresión regular. Si se trata de una cadena, sólo se reemplaza la primera aparición de la searchValue

string.search(regexp)

El método de búsqueda es como el método indexOf, excepto que toma un objeto de expresión regular en lugar de una cadena. Devuelve la posición del primer carácter del primer partido, si lo hay, o -1 si la búsqueda falla. El indicador g se ignora. No hay parámetro de posición

string.slice(start,end)

El método slice hace una nueva cadena copiando una parte de otra cadena. Si el parámetro de inicio es negativo, se añade string.length a ella. El parámetro final es opcional, y su valor por defecto es string.length. Si el parámetro final es negativo, entonces string.length se añade a la misma. El parámetro final es uno mayor que la posición del último carácter. Para obtener n caracteres empezando en la posición p, usestring.slice (p, p + n).

string.split(separator,limit)
El método split crea una matriz de cadenas mediante el fraccionamiento de esta cadena en trozos. El parámetro de límite opcional puede limitar el número de piezas que se dividen. El parámetro separador puede ser una cadena o una expresión regular.

Capítulo 9.

Style

Computer programs are the most complex things that humans make. Programs are made up of a huge number of parts, expressed as functions, statements, and expressions that are arranged in sequences that must be virtually free of error. The runtime behavior has little resemblance to the program that implements it. Software is usually expected to be modified over the course of its productive life. The process of converting one correct program into a different correct program is extremely challenging.

Good programs have a structure that anticipates—but is not overly burdened by—the possible modifications that will be required in the future. Good programs also have a clear presentation. If a program is expressed well, then we have the best chance of being able to understand it so that it can be successfully modified or repaired.