**Resumen de Java Script**

**------------- o -------------**

**Cyntia Fernández Marenco**

Tabla de contenido

[Windows Open() 6](#_Toc412151960)

[String toLowerCase() 6](#_Toc412151961)

[String toUpperCase() 6](#_Toc412151962)

[Prompt() 6](#_Toc412151963)

[Reserved Keywords (Palabras Reservadas) 7](#_Toc412151964)

[Confirm() 7](#_Toc412151965)

[Undefined 7](#_Toc412151966)

[Null 7](#_Toc412151967)

[TypeOf 7](#_Toc412151968)

[Web Browse Engine / Motor de renderizado 7](#_Toc412151969)

[Lista de Motores EcmaScript 8](#_Toc412151970)

[Motor EcmaScript 8](#_Toc412151971)

[V8 8](#_Toc412151972)

[Scripting Lenguaje 8](#_Toc412151973)

[Dynamic Typing 8](#_Toc412151974)

[Onunload 8](#_Toc412151975)

[Onload 8](#_Toc412151976)

[Delete 9](#_Toc412151977)

[OOP – Programación Orientada a Objetos 9](#_Toc412151978)

[Constructor de Objetos 9](#_Toc412151979)

[Diferencia entre Document.ready y Onunload 9](#_Toc412151980)

[== Igual Igual 9](#_Toc412151981)

[=== Igual Igual Igual 9](#_Toc412151982)

[Diferencia entre == o === 9](#_Toc412151983)

[Capitulo 2-Gramática 9](#_Toc412151984)

[El espacio en blanco 9](#_Toc412151985)

[Nombres 10](#_Toc412151986)

[Números 10](#_Toc412151987)

[String 10](#_Toc412151988)

[Declaraciones 10](#_Toc412151989)

[Do 10](#_Toc412151990)

[Try 10](#_Toc412151991)

[Throw 11](#_Toc412151992)

[Return 11](#_Toc412151993)

[Expresiones 11](#_Toc412151994)

[Los literales 11](#_Toc412151995)

[Funciones 11](#_Toc412151996)

[Capitulo 3-Objetos 11](#_Toc412151997)

[Los literales de objetos 11](#_Toc412151998)

[Recuperación 11](#_Toc412151999)

[Actualización 12](#_Toc412152000)

[Referencia 12](#_Toc412152001)

[Prototype / Prototipo 12](#_Toc412152002)

[Reflexión 12](#_Toc412152003)

[Enumeración 12](#_Toc412152004)

[Delete / Borrar 12](#_Toc412152005)

[Apendice A 12](#_Toc412152006)

[Variables globales 12](#_Toc412152007)

[Scope 13](#_Toc412152008)

[Semicolon Insertion 13](#_Toc412152009)

[Unicode 13](#_Toc412152010)

[ParseInt 13](#_Toc412152011)

[Operador + 13](#_Toc412152012)

[Floating Point 14](#_Toc412152013)

[NaN 14](#_Toc412152014)

[Phony Arrays 14](#_Toc412152015)

[Falsy Values 14](#_Toc412152016)

[hasOwnProperty 14](#_Toc412152017)

[Objects 14](#_Toc412152018)

[JavaScript Tipo de conversión 15](#_Toc412152019)

[Tipos de datos JavaScript 15](#_Toc412152020)

[El operador typeof 15](#_Toc412152021)

[La propiedad constructor 16](#_Toc412152022)

[La conversión de números a cadenas 16](#_Toc412152023)

[La conversión de los Booleanos para Cuerdas 17](#_Toc412152024)

[Conversión de Fechas para Cuerdas 17](#_Toc412152025)

[La conversión de cadenas en números 17](#_Toc412152026)

[Conversión Booleans a Números 17](#_Toc412152027)

[La conversión de fechas a Números 18](#_Toc412152028)

[Tipo de conversión automática 18](#_Toc412152029)

[Conversión automática de Cuerdas 18](#_Toc412152030)

[JavaScript Izar 18](#_Toc412152031)

[IndexOf() 18](#_Toc412152032)

[JavaScript Errores Throw y tratar de atrapar 19](#_Toc412152033)

[JavaScript Lanza Errores 19](#_Toc412152034)

[La Declaración de tiro 19](#_Toc412152035)

[La sentencia finally 19](#_Toc412152036)

[JavaScript Timing Eventos 20](#_Toc412152037)

[El setInterval (Método) 20](#_Toc412152038)

[Cómo detener la ejecución? 20](#_Toc412152039)

[El setTimeout (Método) 21](#_Toc412152040)

[Cómo detener la ejecución? 21](#_Toc412152041)

[Capitulo 4-Funciones 21](#_Toc412152042)

[Function Objects 21](#_Toc412152043)

[The Method Invocation Pattern 22](#_Toc412152044)

[The Apply Invocation Pattern 22](#_Toc412152045)

[The Function Invocation Pattern 22](#_Toc412152046)

[The Constructor Invocation Pattern 22](#_Toc412152047)

[Recursion 23](#_Toc412152048)

[Scope 23](#_Toc412152049)

[Callbacks 23](#_Toc412152050)

[Module 23](#_Toc412152051)

[Cascade 24](#_Toc412152052)

[Memoization 24](#_Toc412152053)

[Capitulo 5-Inheritance 25](#_Toc412152054)

[Pseudoclassical 25](#_Toc412152055)

[Objeto amplificaciones 25](#_Toc412152056)

[Prototypal 25](#_Toc412152057)

[Funcional 25](#_Toc412152058)

[Regiones 25](#_Toc412152059)

[Apendice B-Bad Parts 25](#_Toc412152060)

[With Statement 25](#_Toc412152061)

[Eval 26](#_Toc412152062)

[Continue Statement 26](#_Toc412152063)

[Switch Fall Through 26](#_Toc412152064)

[Block-less Statements 26](#_Toc412152065)

[Bitwise Operators 26](#_Toc412152066)

[Typed Wrappers 26](#_Toc412152067)

[New 27](#_Toc412152068)

[Void 27](#_Toc412152069)

[Closures 27](#_Toc412152070)

[Emulando métodos privados con closures 27](#_Toc412152071)

[Consideraciones de rendimiento 27](#_Toc412152072)

# Windows Open()

Abre una nueva ventana.

Close() Cierra una ventana.

Ejemplo:

SYNTAXIS:

window.open(*URL,name,specs,replace*)

Eval()

Evalúa o realiza un argumento que se le pase

Ejemplo:

SYNTAXIS:

eval(*string*)

# String toLowerCase()

Convierte un String a letras minúsculas

Ejemplo:

SYNTAXIS:

*string*.toLowerCase()

# String toUpperCase()

Convierte un String a letras mayúsculas.

Ejemplo:

SYNTAXIS:

*string*.toUpperCase()

# Prompt()

Cuadros de diálogos que se le muestran al visitante a la hora de entrada

Ejemplo:

SYNTAXIS:

prompt(*text,defaultText*)

# Reserved Keywords (Palabras Reservadas)

Las siguientes palabras están reservadas en JavaScript:

abstract boolean break byte case catch char class const continue debugger default delete do double else enum export extends false final finally float for function goto if implements import in instanceof int interface long native new null package private protected public return short static super switch synchronized this throw throws transient true try typeof var volatile void while with

La mayoría de estas palabras no se usan en el lenguaje. No pueden ser utilizados para las variables o parámetros de nombres. Cuando las palabras reservadas se utilizan como claves en objetos literales, deben ser citados. Ellos no se pueden utilizar con la notación de puntos, por lo que es necesario a veces utilizar la notación de soporte en su lugar.

# Confirm()

Es una caja de dialogo con la que se especifica un mensaje a lo largo con un Ok o un Cancelar.

Una caja de Confirm es a menudo usada para darle al usuario una aceptación o verificación algunas veces.

Ejemplo:

SYNTAXIS:

confirm(*if, alert, else*)

# Undefined

Son las variables que no han sido definidas por lo tanto no existen o que han sido definidas sin asignarles un valor.

# Null

Este en Javascript es un valor, especial. El valor null es en si mismo un calor, pero un valor que indica la ausencia de contenido, el valor vacio.

# TypeOf

Devuelve una cadena que indica el tipo del operando a evaluar.

# Web Browse Engine / Motor de renderizado

Conocido también como el motor del navegador web, es un software de componente que tiene marcado contenido HtML, XtMl y otros.

Este permite dibujar el contenido en una ventana, para así mostrar el código ya formateado.

# Lista de Motores EcmaScript

# Motor EcmaScript

Es un programa que ejecuta código fuente escrito en una versión del EcmaScript.

Lenguaje estándar:

V8(Utilizado por google Chrome)

Carakan

Chakra

Javacript core, entre otros.

# V8

Es un motor de código abierto para javascript creado por google.

Esta escrito en C++(Lenguaje de programación que permite la manipulación de objetos) y es usado por Google Chrome. Además de que tienen integrado el Sistema Operativo Android 2.2. Puede funcionar solo o incorporado a cualquier aplicación C++.

# Scripting Lenguaje

Es un lenguaje de programación que admite secuencias de comandos, es un lenguaje de alto nivel que esta interpretado por otro programa en tiempo de ejecución que en lugar de compilarlo lo interpreta y automatiza.

# Dynamic Typing

Lenguaje de programación mecanografiado dinámico. En el cual necesariamente deben definirse las variables antes de ser utilizadas.

PHP y PHYTON son lenguajes de programación con lenguaje dinámico.

# Onunload

Este evento se produce cuando una página se ha descargado o la ventana del navegador se ha cerrado, o también cuando el usuario se desplaza fuera de la página o carga la página.

# Onload

Carga el DOM y todas las imágenes y hasta que esté completamente cargado se ejecuta.

# Delete

Borra un objeto o elemento de una variable.

# OOP – Programación Orientada a Objetos

Programa y refleja lo que queremos resolver en la vida cotidiana.

# Constructor de Objetos

Nos ayuda a devolver una instancia. Es una función que esta dentro de una clase.

# Diferencia entre Document.ready y Onunload

Uno carga únicamente el DOM.

# == Igual Igual

Compara solo el valor

# === Igual Igual Igual

Compara valor y tipo.

# Diferencia entre == o ===

Que uno compara si el valor y el tipo son iguales.

# Capitulo 2-Gramática

# El espacio en blanco

El espacio en blanco puede tomar la forma de caracteres de formato o comentarios. Es necesario el uso de los espacios en blanco para separar las secuencias de caracteres que de otro modo se combinan en un único token.

JavaScript ofrece dos formas de comentarios, comentarios en bloque formados con / \* \* / y los comentarios de fin de línea comenzando con //. Los comentarios deben ser usados libremente para mejorar la legibilidad de los programas. Tenga cuidado de que los comentarios siempre describen con precisión el código. Comentarios obsoletos son peores que no tiene comentarios.

# Nombres

Es una carta seguido opcionalmente por una o más letras, dígitos o guiones bajos. Un nombre no puede ser una de estas palabras reservadas:

abstract boolean byte descanso clase Char capturas caso const continuar depurador predeterminado eliminar hacer doble

La mayoría de las palabras reservadas en esta lista no se utilizan en el lenguaje. La lista no incluye algunas palabras que deberían haberse reservado pero no lo fueron, como no definido, NaN, y el Infinito. No está permitido nombrar una variable o parámetro con una palabra reservada.

# Números

JavaScript tiene un solo tipo de número. Internamente, se representa como punto flotante de 64 bits, la misma que la de doble Java. A diferencia de la mayoría de otros lenguajes de programación, no hay ningún tipo de entero por separado, por lo que 1 y 1.0 son el mismo valor.

# String

Cuerdas una cadena literal se pueden envolver en comillas simples o dobles. Puede contener cero o más caracteres. El \ (barra invertida) es el carácter de escape.

# Declaraciones

Es una unidad de compilación que contiene un conjunto de instrucciones ejecutables. En los navegadores web, cada etiqueta <script> entrega una unidad de compilación que se compila y ejecuta inmediatamente. Al carecer de un enlazador, JavaScript todas lanza juntos en un espacio de nombres global común.

Las declaraciones tienden a ser ejecutados en orden de arriba a abajo.

# Do

La instrucción do es como la instrucción while excepto que la expresión se ensaya después el bloque se ejecuta en lugar de antes. Eso significa que el bloque siempre se ejecutará al menos una vez.

# Try

La sentencia try ejecuta un bloque y las capturas de las excepciones que fueron arrojados por el bloque. La cláusula catch define una nueva variable que recibirá el objeto de excepción.

# Throw

La sentencia throw lanza una excepción. Si la sentencia throw está en un bloque try, el control continúa en la cláusula catch. De lo contrario, la invocación de la función es abandonado, y el control pasa a la cláusula catch de la prueba en la función que llama.

# Return

La sentencia return hace que el pronto retorno de una función. También puede especificar el valor a devolver. Si no se especifica una expresión de retorno, entonces el valor de retorno será indefinido.

# Expresiones

Las expresiones más simples son un valor literal (tal como una cadena o un número), una variable, un valor integrado entre otros.

# Los literales

Los literales de objetos son una notación conveniente para especificar los objetos nuevos. Los nombres de las propiedades se pueden especificar como nombres o como cadenas.

# Funciones

Una literal función define un valor de función. Puede tener un nombre opcional que puede usar para llamar a sí mismo de forma recursiva. Puede especificar una lista de parámetros que actuarán como variables incluye definiciones y declaraciones.

# Capitulo 3-Objetos

# 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.

# 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 la notación se puede utilizar en su lugar. Se prefiere la notación porque es más compacto y se lee mejor.

# Actualización

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.

# Referencia

Objetos se pasan alrededor por referencia. Nunca se copian.

# Prototype / Prototipo

Es cada objeto que 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.

# Reflexión

Es para inspeccionar de manera fácil 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.

# Enumeración

Es donde se incluirán 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 / Borrar

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.

# Apendice A

# Variables globales

La peor de todas las malas características de JavaScript es su dependencia de las variables globales.

Una variable global es una variable que es visible en todos los ámbitos. Las variables globales pueden ser una conveniencia en programas muy pequeños, pero convertido rápidamente en difícil de manejar como los programas se hacen más grandes. Debido a que una variable global se puede cambiar por cualquier parte del programa en cualquier momento, pueden complicar significativamente el comportamiento del programa.

# 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.

# 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.

# 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.

# ParseInt

ParseInt es una función que convierte una cadena en un número entero. Se detiene cuando ve un nondigit, por lo parseInt ("16") y parseInt ("16 toneladas") producen el mismo resultado.

# Operador +

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. Este es el error más frecuentemente reportado en JavaScript, y es una consecuencia intencional de haber adoptado el estándar IEEE para aritmética binaria FloatingPoint (IEEE 754). Esta norma es muy adecuado para muchas aplicaciones, pero viola la mayoría de las cosas que ha aprendido acerca de los números en la escuela media.

# NaN

El valor NaN es una cantidad especial definido por IEEE 754. Representa no es un número, a pesar de que:

typeof "número" NaN === // true

El valor puede ser producido por el intento de convertir una cadena en un número cuando la cadena no está en la forma de un número.

# 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.

# Falsy Values

JavaScript tiene un sorprendente conjunto de valores falsy. Estos valores son todos falsy, pero no son intercambiables.

# 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.

# Objects

Objetos de JavaScript nunca son verdaderamente vacío, ya que pueden recoger a los miembros de la cadena de prototipo.

# JavaScript Tipo de conversión

Número () convierte a un Number, String () convierte a String, Boolean ()

convierte a un booleano.

# Tipos de datos JavaScript

En JavaScript hay 5 tipos de datos diferentes que pueden contener valores:

● cadena

● número

● boolean

● objeto

● función

Hay 3 tipos de objetos:

● Objeto

● Fecha

● Colección

Y 2 tipos de datos que no pueden contener valores:

● nulo

● indefinido

# El operador typeof

Puede utilizar el typeof operador para encontrar el tipo de datos de una variable

de JavaScript.

**Ejemplo**

typeof "John" // Returns string

typeof 3.14 // Returns number

typeof NaN // Returns number

typeof false // Returns boolean

typeof [ 1 , 2 , 3 , 4 ] // Returns object

typeof {name: 'John' , age:34} // Returns object

typeof new Date() // Returns object

typeof function () {} // Returns function

typeof myCar // Returns undefined (if myCar is not

declared)

typeof null // Returns object

**I**

**nténtelo usted mismo »**

Tenga en cuenta:

● El tipo de datos de NaN es el número

● El tipo de datos de una matriz es objeto

● El tipo de datos de una fecha es objeto

● El tipo de datos de nula es objeto

● El tipo de datos de una variable no definida es indefinido

No se puede usar typeof para definir si un objeto es un array JavaScript o una

Fecha de JavaScript.

# La propiedad constructor

El constructor propiedad devuelve la función constructora para todas las variables

de JavaScript.

**Ejemplo**

"John" .constructor // Returns function String() { [native

code] }

( 3.14 ).constructor // Returns function Number() { [native

code] }

false.constructor // Returns function Boolean() { [native

code] }

[ 1 , 2 , 3 , 4 ].constructor // Returns function Array() { [native

code] }

{name: 'John' , age:34}.constructor // Returns function Object() { [native

code] }

new Date().constructor // Returns function Date() { [native

code] }

function () {}.constructor // Returns function Function(){ [native

code] }

**Inténtelo usted mismo »**

Usted puede comprobar la propiedad constructor para averiguar si un objeto es

un array (contiene la palabra "arrays"):

# La conversión de números a cadenas

El método global String () puede convertir números a cadenas.

Se puede utilizar en cualquier tipo de números, literales, variables o expresiones:

**Ejemplo**

String(x) // returns a string from a number variable x

String( 123 ) // returns a string from a number literal 123

String( 100 + 23 ) // returns a string from a number from an expression

# La conversión de los Booleanos para Cuerdas

El método global String () puede convertir booleanos para cuerdas.

String( false ) // returns "false"

String( true ) // returns "true"

El método booleano toString () hace lo mismo.

false.toString() // returns "false"

true.toString() // returns "true"

# Conversión de Fechas para Cuerdas

El método global String () puede convertir fechas para cuerdas.

String(Date()) // returns Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe

Daylight Time)

El método Fecha toString () hace lo mismo.

Ejemplo

Date().toString() // returns Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe

Daylight Time

# La conversión de cadenas en números

El método global Number () puede convertir cadenas a números.

Las cadenas que contengan números (como "3.14") se convierten en números

(como 3.14).

Las cadenas vacías se convierten en 0.

Todo lo demás se convierte en NaN (no es un número).

Number( "3.14" ) // returns 3.14

Number( " " ) // returns 0

Number( "" ) // returns 0

Number( "99 88" ) // returns NaN

# Conversión Booleans a Números

El método global Number () también puede convertir booleans a números.

Number( false ) // returns 0

Number( true ) // returns 1

# La conversión de fechas a Números

El método global Number () puede ser usado para convertir las fechas para

números.

d = new Date();

Number(d) // returns 1404568027739

El método de la fecha getTime () hace lo mismo.

d = new Date();

d.getTime() // returns 1404568027739

# Tipo de conversión automática

Cuando JavaScript intenta operar en un tipo de datos "mal", que tratará de

convertir el valor a un tipo de "derecho".

El resultado no siempre es lo que esperas:

5 + null // returns 5 because null is converted to 0

"5" + null // returns "5null" because null is converted to "null"

"5" + 1 // returns "51" because 1 is converted to "1"

"5" - 1 // returns 4 because "5" is converted to 5

# Conversión automática de Cuerdas

JavaScript llama automáticamente a la función toString de la variable () cuando

intenta "salida" de un objeto o una variable:

document.getElementById( "demo" ).innerHTML = myVar

# JavaScript Izar

Izar es JavaScript comportamiento predeterminado 's de las declaraciones de pasar a la parte superior.

# IndexOf()

El **indexOf()** método 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.

# JavaScript Errores Throw y tratar de atrapar

El intento declaración le permite probar un bloque de código para los

errores.

La captura declaración le permite manejar el error.

El tiro declaración le permite crear errores personalizados.

El último comunicado le permite ejecutar código, después de tratar de

atrapar, sin importar el resultado.

# JavaScript Lanza Errores

Cuando se produce un error, JavaScript será normalmente parar, y generar un

mensaje de error.

El término técnico para esto es: JavaScript será lanzar un error.

# La Declaración de tiro

El tiro declaración le permite crear un error personalizado.

El término técnico para esto es: una excepción .

La excepción puede ser una cadena JavaScript, un número, un booleano o un objeto.

# La sentencia finally

El último comunicado le permite ejecutar código, después de tratar de atrapar,

independientemente del resultado.

Ejemplo

function myFunction() {

var message, x;

message = document.getElementById( "message" );

message.innerHTML = "" ;

x = document.getElementById( "demo" ).value;

try {

x = Number(x);

if (x == "" ) throw "is empty" ;

if (isNaN(x)) throw "is not a number" ;

if (x > 10 ) throw "is too high" ;

if (x < 5 ) throw "is too low" ;

}

catch (err) {

message.innerHTML = "Error: " + err + "." ;

}

finally {

document.getElementById( "demo" ).value = "" ;

}

}

# JavaScript Timing Eventos

Con JavaScript, es posible ejecutar un código en intervalos de tiempo especificados.

Esto se llama eventos de tiempo.

Es muy fácil para los acontecimientos del tiempo en JavaScript. Los dos métodos

principales que se utilizan son:

● setInterval () - ejecuta una función, una y otra vez, a intervalos de tiempo

especificados

● setTimeout () - ejecuta una función, una vez, después de esperar un número

especificado de milisegundos

Nota: El setInterval () y setTimeout () son los dos métodos del objeto HTML DOM

ventana.

# El setInterval (Método)

El método setInterval () esperará un número especificado de milisegundos, y luego

ejecutar una función específica, y continuará para ejecutar la función, una vez en

cada intervalo de tiempo dado.

Sintaxis

window.setInterval(" *javascript function* ", *milliseconds* );

El window.setInterval () método puede ser escrita sin el prefijo ventana.

El primer parámetro de setInterval () debe ser una función.

El segundo parámetro indica la longitud de los intervalos de tiempo entre cada

ejecución.

Nota: Hay 1.000 milisegundos en un segundo.

# Cómo detener la ejecución?

El método clearInterval () se utiliza para detener nuevas ejecuciones de la función

especificada en el método setInterval ().

Sintaxis

window.clearInterval( *intervalVariable* )

El window.clearInterval () método puede ser escrita sin el prefijo ventana.

Para poder utilizar el método clearInterval (), debe utilizar una variable global al

crear el método de intervalo:

myVar=setInterval(" *javascript function* ", *milliseconds* );

Entonces usted será capaz de detener la ejecución llamando al método clearInterval

().

# El setTimeout (Método)

Sintaxis

window.setTimeout(" *javascript function* ", *milliseconds* );

El window.setTimeout () método puede ser escrita sin el prefijo ventana.

El método setTimeout () esperará el número especificado de milisegundos, y luego ejecutar la función especificada.

El primer parámetro de setTimeout () debe ser una función.

El segundo parámetro indica el número de milisegundos, a partir de ahora, que quiere ejecutar el primer parámetro.

# Cómo detener la ejecución?

El método clearTimeout () se utiliza para detener la ejecución de la función

especificada en el método setTimeout ().

Sintaxis

window.clearTimeout( *timeoutVariable* )

El window.clearTimeout () método puede ser escrita sin el prefijo ventana.

Para poder utilizar el método clearTimeout (), debe utilizar una variable global al

crear el método de tiempo de espera:

myVar=setTimeout(" *javascript function* ", *milliseconds* );

Entonces, si la función no ha sido ya ejecutado, usted será capaz de detener la

ejecución llamando al método clearTimeout ().

# Capitulo 4-Funciones

# Function Objects

Functions in JavaScript are objects. Objects are collections of name/value pairs having a hidden link to a prototype object. Objects produced from object literals are

linked to Object.prototype. Function objects are linked to Function.prototype (which is itself linked to Object.prototype). Every function is also created with two additional hidden properties: the function’s context and the code that implements the function’s behavior.

# The Method Invocation Pattern

When a function is stored as a property of an object, we call it a *method*. When a

method is invoked, this is bound to that object. If an invocation expression contains

a refinement (that is, a . dot expression or [*subscript*] expression), it is invoked as a method.

A method can use this to access the object so that it can retrieve values from the

object or modify the object. The binding of this to the object happens at invocation

time. This very late binding makes functions that use this highly reusable. Methods that get their object context from this are called *public methods*.

# The Apply Invocation Pattern

Because JavaScript is a functional object-oriented language, functions can have methods. The apply method lets us construct an array of arguments to use to invoke a function.

It also lets us choose the value of this. The apply method takes two parameters.

The first is the value that should be bound to this. The second is an array of

parameters.

# The Function Invocation Pattern

When a function is not the property of an object, then it is invoked as a function:

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

When a function is invoked with this pattern, this is bound to the global object.

This was a mistake in the design of the language. Had the language been designed correctly, when the inner function is invoked, this would still be bound to the this Invocation variable of the outer function.

# The Constructor Invocation Pattern

JavaScript is a *prototypal* inheritance language. That means that objects can inherit properties directly from other objects. The language is class-free.

This is a radical departure from the current fashion. Most languages today are *classical*.

Prototypal inheritance is powerfully expressive, but is not widely understood.

JavaScript itself is not confident in its prototypal nature, so it offers an object-making syntax that is reminiscent of the classical languages. Few classical programmers found prototypal inheritance to be acceptable, and classically inspired syntax obscures the language’s true prototypal nature. It is the worst of both worlds. If a function is invoked with the new prefix, then a new object will be created with a hidden link to the value of the function’s prototype member, and this will be bound to that new object.

# Recursion

A *recursive* function is a function that calls itself, either directly or indirectly. Recursion is a powerful programming technique in which a problem is divided into a set of similar subproblems, each solved with a trivial solution. Generally, a recursive function calls itself to solve its subproblems.

# Scope

*Scope* in a programming language controls the visibility and lifetimes of variables and parameters. This is an important service to the programmer because it reduces naming collisions and provides automatic memory managemen.

# Callbacks

Functions can make it easier to deal with discontinuous events. For example, suppose there is a sequence that begins with a user interaction, making a request of the server, and finally displaying the server’s response. The naive way to write that would be:

request = prepare\_the\_request( );

response = send\_request\_synchronously(request);

display(response);

The problem with this approach is that a synchronous request over the network will

leave the client in a frozen state. If either the network or the server is slow, the degradation in responsiveness will be unacceptable.

# Module

We can use functions and closure to make modules. A module is a function or object that presents an interface but that hides its state and implementation. By using functions to produce modules, we can almost completely eliminate our use of global variables, thereby mitigating one of JavaScript’s worst features.

For example, suppose we want to augment String with a deentityify method. Its

job is to look for HTML entities in a string and replace them with their equivalents.

It makes sense to keep the names of the entities and their equivalents in an object.

But where should we keep the object? We could put it in a global variable, but global variables are evil. We could define it in the function itself, but that has a runtime cost because the literal must be evaluated every time the function is invoked. The ideal approach is to put it in a closure, and perhaps provide an extra method that can add additional entities.

# Cascade

Some methods do not have a return value. For example, it is typical for methods that set or change the state of an object to return nothing. If we have those methods return this instead of undefined, we can enable *cascades*. In a cascade, we can call many methods on the same object in sequence in a single statement. An Ajax library that enables cascades would allow us to write in a style like this:

getElement('myBoxDiv').

move(350, 150).

width(100).

height(100).

color('red').

border('10px outset').

padding('4px').

appendText("Please stand by").

Curry | 43

on('mousedown', function (m) {

this.startDrag(m, this.getNinth(m));

}).

on('mousemove', 'drag').

on('mouseup', 'stopDrag').

later(2000, function ( ) {

this.

color('yellow').

setHTML("What hath God wraught?").

slide(400, 40, 200, 200);

}).

tip('This box is resizeable');

Cascading can produce interfaces that are very expressive. It can help control the tendency to make interfaces that try to do too much at once.

# Memoization

Functions can use objects to remember the results of previous operations, making it possible to avoid unnecessary work. This optimization is called *memoization*.

JavaScript’s objects and arrays are very convenient for this.

Let’s say we want a recursive function to compute Fibonacci numbers. A Fibonacci number is the sum of the two previous Fibonacci numbers. The first two are 0 and 1:

var fibonacci = function (n) {

return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);

};

for (var i = 0; i <= 10; i += 1) {

document.writeln('// ' + i + ': ' + fibonacci(i));

}

// 0: 0

// 1: 1

// 2: 1

// 3: 2

// 4: 3

// 5: 5

// 6: 8

// 7: 13

// 8: 21

// 9: 34

// 10: 55

# Capitulo 5-Inheritance

# Pseudoclassical

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.

# Objeto amplificaciones

 Utilizado cuando 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

Patrón puramente prototípico, prescindimos de clases. Herencias de prototipos es conceptualmente más simple que la herencia clásica: un nuevo objeto puede heredar las propiedades de un objeto antiguo. De esta manera hacemos objetos reconocidos.

# Funcional

Usada para iniciar 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.

# Regiones

 Es cuando 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.

# Apendice B-Bad Parts

# With Statement

Declaración de JavaScript que fue con la intención de proporcionar un atajo al acceder a las propiedades de un objeto. Por desgracia, sus resultados pueden ser a veces impredecible, por lo que deben evitarse.

# Eval

Pasa una cadena al compilador JavaScript y ejecuta el resultado. Es el hecho más mal utilizados de JavaScript.

Es más comúnmente utilizado por las personas que tienen una comprensión incompleta de la lengua.

# Continue Statement

Ésta sentencia continue salta a la parte superior del bucle. Nunca se ha visto un trozo de código que no fue mejorada por la refactorización para eliminar la sentencia continúe.

# 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.

# 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.

# Bitwise Operators

JavaScript tiene el mismo conjunto de operadores de bits como Java:

& and

| or

^ xor

~ not

>> signed right shift

>>> unsigned right shift

<< left shift

En Java, los operadores de bits trabajan con números enteros. JavaScript no tiene números enteros. Sólo tiene los números de coma flotante de doble precisión. Por lo tanto, los operadores de bits convierten sus operandos numéricos en enteros, hacen sus negocios, y luego convertirlos de nuevo

# Typed Wrappers

Conjunto de envolturas mecanografiadas. Por ejemplo:

new Boolean (false) produce un objeto que tiene un método valueOf que devuelve el valor ajustado.

Esta resulta ser completamente innecesario y ocasionalmente confuso. No utilice nuevo número booleano o una nueva o una nueva cadena. También evite nuevo Objeto y new Array. Use {} y [] en lugar.

# New

Nuevo operador de JavaScript el cual crea un nuevo objeto que hereda de miembro prototipo del operando, y luego llama el operando, la unión del nuevo objeto a esto. Esto da el operando (que mejor había ser una función constructor) la oportunidad de personalizar el nuevo objeto antes de que se devuelve al solicitante.

# Void

En muchos idiomas, nula es un tipo que no tiene valores. En JavaScript, nula es un operador que toma un operando y devuelve indefinido. Esto no es útil, y es muy confuso. Evite vacío.

# Closures

Los closures son funciones que manejan variables independientes. En otras palabras,

la función definida en el closure "recuerda" el entorno en el que se ha creado.

# Emulando métodos privados con closures

Lenguajes como Java ofrecen la posibilidad de declarar métodos privados, es decir, que sólo pueden ser llamados por otros métodos en la misma clase.

JavaScript no proporciona una forma nativa de hacer esto, pero es posible emular

métodos privados utilizando closures. Los métodos privados no son sólo útiles para restringir el acceso al código: también proporcionan una poderosa manera de

administrar tu espacio de nombres global, evitando que los métodos no esenciales

embrollen la interfaz pública de tu código.

# Consideraciones de rendimiento

No es aconsejable crear innecesariamente funciones dentro de otras funciones si no se necesitan los closures para una tarea particular ya que afectará negativamente el rendimiento del script tanto en consumo de memoria como en velocidad de procesamiento.