Funciones

En términos generales, una función es un "subprograma" que puede ser llamado por código externo (o interno en caso de recursión) a la función. Al igual que el programa en sí mismo, una función se compone de una secuencia de declaraciones, que conforman el llamado cuerpo de la función. Se pueden pasar valores a una función, y la función puede devolver un valor.

En JavaScript, las funciones son objetos de primera clase, es decir, son objetos y se pueden manipular y transmitir al igual que cualquier otro objeto. Concretamente son objetos Function.

General

Toda función en JavaScript es un objeto Function.

Las funciones no son lo mismo que los procedimientos. Una función siempre devuelve un valor, pero un procedimiento, puede o no puede devolver un valor.

Para devolver un valor específico distinto del predeterminado, una función debe tener una sentencia return, que especifique el valor a devolver. Una función sin una instrucción return devolverá el valor predeterminado. En el caso de un constructor llamado con la palabra clave new, el valor predeterminado es el valor de su parámetro. Para el resto de funciones, el valor predeterminado es undefined.

Los parámetros en la llamada a una función son los argumentos de la función. Los argumentos se pasan a las funciones por valor. Si la función cambia el valor de un argumento, este cambio no se refleja globalmente ni en la llamada de la función. Sin embargo, las referencias a objetos también son valores, y son especiales: si la función cambia las propiedades del objeto referenciado, ese cambio es visible fuera de la función, tal y como se muestra en el siguiente ejemplo:

```
/* Declarando la función 'myFunc' */
 function myFunc(elobjeto)
 {
   elobjeto.marca= "Toyota";
 }

 /*
  * Declarando la variable 'mycar';
  * Se crea e inicializa el nuevo objeto;
  * para hacer referencia a él mediante 'mycar'
  */
 var mycar = {
   marca: "Honda",
   modelo: "Accord",
   año: 1998
 };

 /* Mostrando 'Honda' */
 window.alert(mycar.marca);

 /* Paso por referencia del objeto 'mycar' a la función 'myFunc'*/
```

```
 myFunc(mycar);

 /*
  * Muestra 'Toyota' como valor de la propiedad 'marca'
  * del objeto, que ha sido cambiado por la función.
  */
 window.alert(mycar.marca);
```

La palabra clave this no hace referencia a la función que está ejecutándose actualmente, por lo que hay que referirse a los objetos Function por nombre, incluso dentro del cuerpo de la función.

Definiendo funciones

Hay varias formas de definir funciones:

Declaración de una función (La instrucción function)

Su sintaxis es:

```
function nombre([param[,param[, ...param]]]) {
   instrucciones
}
```

en donde:

nombre	El nombre de la función.
Param	El nombre de un argumento que se le pasará a la función. Una función puede tener hasta 255 argumentos.
Instrucciones	Las instrucciones que forman el cuerpo de la función.

La expresión de función flecha (=>)

Nota: Las expresiones de función Flecha son una tecnología experimental, parte de la proposición Harmony (EcmaScript 6) y no son ampliamente implementadas por los navegadores.

Una expresión de función flecha tiene una sintaxis más corta y su léxico se une a este valor:

```
([param] [, param]) => { instrucciones }
```

```
param => expresión
```

param	El nombre de un argumento. Si no hay argumentos se tiene que indicar con (). Para un único argumento no son necesarios los paréntesis. (como foo => 1)

instrucciones o expresión	Múltiples instrucciones deben ser encerradas entre llaves. Una única expresión no necesita llaves. La expresión es, así mismo, el valor de retorno implícito de esa función.

El constructor Function

Como todos los demás objetos, los objetos Function se pueden crear mediante el operador new:

new Function (arg1, arg2, ... argN, functionBody)

arg1, arg2, ... argN
Ningún o varios argumentos son pasados para ser utilizados por la función como nombres de argumentos formales. Cada uno debe ser una cadena que se ajuste a las reglas de identificadores válidos en JavaScript, o a una lista de este tipo de cadenas separadas por comas; por ejemplo "x", "theValue", o "a,b".

Cuerpo de la función
Se trata de una cadena conteniendo las instrucciones JavaScript que comprenden la definición de la función.

Llamar al constructor Function como una función, sin el operador new, tiene el mismo efecto que llamarlo como un constructor.

Nota: Utilizar el constructor Function no se recomienda, ya que necesita el cuerpo de la función como una cadena, lo cual puede ocasionar que no se optimice correctamente por el motor JS, y puede también causar otros problemas.

El objeto arguments
Puedes referirte a los argumentos de una función dentro de la misma, utilizando el objeto arguments.

Ámbito de ejecución y pila de funciones

Una función puede referirse y llamarse a sí misma. Hay tres maneras en las que una función puede referirse a sí misma.

El nombre de la función
arguments.callee
una función dentro del ámbito de ejecución que refiere a la función

Por ejemplo, considere la siguiente definición de función:

```
var foo = function bar() {
   // el cuerpo va aquí
};
```

Dentro del cuerpo de la función, todo lo siguientes son lo mismo:

```
bar()
arguments.callee()
foo()
```

Una función que se llama a sí misma es llamada una función recursiva. En algunas ocasiones, la recursión es análoga a un bucle. Ambos ejecutan el mismo código múltiples veces, y ambas requieren una condición (para evitar un bucle infinito, o en su lugar, recursión infinita en este caso). Por ejemplo, el siguiente bucle:

```
var x = 0;
while (x < 10) { // "x < 10" es la condición
  // haz algo
  x++;
}
```

puede ser convertida en una función recursiva y una llamada a esa función:

```
function loop(x) {
  if (x >= 10) // "x >= 10" es la condición de salida (equivalente a "!(x < 10)")
    return;
  // haz algo
  loop(x + 1); // la llamada recursiva
}
loop(0);
```

Sin embargo, algunos algoritmos no pueden ser bucles iterativos simples. Por ejemplo, obtener todos los nodos de una estructura de árbol (e.g. el DOM) es realizado de manera más fácil usando recursión:

```
function recorrerArbol (nodo) {
  if (nodo == null) //
    return;
  // haz algo con el nodo
  for (var i = 0; i < nodo.nodosHijos.length; i++) {
      recorrerArbol(nodo.nodosHijos[i]);
  }
}
```

En comparación con el bucle de la función loop, cada llamada recursiva hace muchas llamadas recursivas en este caso.

Es posible convertir cualquier algoritmo recursivo en uno no recursivo, pero a menudo la lógica es mucho más compleja y hacerlo requiere el uso de una pila. De hecho, la recursión utiliza una pila: la pila de funciones.

El comportamiento similar a la pila se puede ver en el ejemplo siguiente:

```
function foo(i) {
  if (i < 0)
    return;
  document.writeln('inicio:' + i);
  foo(i - 1);
  document.writeln('fin:' + i);
}
foo(3);
```

que produce:

```
inicio:3
inicio:2
inicio:1
inicio:0
fin:0
fin:1
fin:2
fin:3
```

Funciones anidadas y cierres

Puede anidar una función dentro de una función. La función anidada (inner) es privada a la función que la contiene (outer). También con la forma: closure.

Un cierre es una expresión (normalmente una función) que puede tener variables libres junto con un entorno que enlaza esas variables (que "cierra" la expresión).
Dado que una función anidada es un cierre, esto significa que una función anidada puede "heredar" los argumentos y las variables de su función contenedora. En otras palabras, la función interna contiene el ámbito de la función externa.
Desde que la función anidada es un cierre (closure), esto significa que una función anidada puede "heredar" los argumentos y variables de su función contenedora. En otras palabras, la función interna contiene un scope (alcance) de la función externa.

Para resumir:

La función interna se puede acceder sólo a partir de sentencias en la función externa.
La función interna forma un cierre: la función interna puede utilizar los argumentos y las variables de la función externa, mientras que la función externa no puede utilizar los argumentos y las variables de la función interna.

El ejemplo siguiente muestra funciones anidadas:

```
function addCuadrado(a,b) {
  function cuadrado(x) {
    return x * x;
  }
  return cuadrado(a) + cuadrado(b);
}
a = addCuadrado(2,3); // retorna 13
b = addCuadrado(3,4); // retorna 25
c = addCuadrado(4,5); // retorna 41
```

Dado que la función interna forma un cierre, puede llamar a la función externa y especificar argumentos para la función externa e interna

```
function fuerade(x) {
  function dentro(y) {
    return x + y;
  }
  return dentro;
}
resultado = fuerade(3)(5); // retorna 8
```

Consideraciones sobre la eficiencia

Observa cómo se conserva x cuando se devuelve dentro. Un cierre conserva los argumentos y las variables en todos los ámbitos que contiene. Puesto que cada llamada proporciona argumentos potencialmente diferentes, debe crearse un cierre para cada llamada a la función externa. En otras palabras, cada llamada a out crea un cierre. Por esta razón, los cierres pueden usar una gran cantidad de memoria. La memoria se puede liberar sólo cuando el dentro devuelto ya no es accesible. En este caso, el cierre del dentro se almacena en resultado. Como el resultado está en el ámbito global, el cierre permanecerá hasta que se descargue el script (en un navegador, esto sucedería cuando la página que contiene el script esté cerrada).

Debido a esta ineficiencia, evita cierres siempre que sea posible.

Constructor vs declaración vs expresión

Las diferencias entre la Function constructora, la de declaración y la de expresión.

Comparando:

Una función definida con el constructor Function asignado a la variable multiply

```
var multiply = new Function("x", "y", "return x * y;");
```

Una declaración de una función denominada multiply

```
function multiply(x, y) {
  return x * y;
}
```

Una expresión de función anónima asignada a la variable multiply

```
var multiply = function(x, y) {
  return x * y;
}
```

Una declaración de una función denominada func_name asignada a la variable multiply

```
var multiply = function func_name(x, y) {
  return x * y;
}
```

Todas hacen aproximadamente la misma cosa, con algunas diferencias sutiles:

Existe una distinción entre el nombre de la función y la variable a la que se asigna la función:
El nombre de la función no se puede cambiar, mientras que la variable a la que se asigna la función puede ser reasignada.

El nombre de la función sólo se puede utilizar en el cuerpo de la función. Intentar utilizarlo fuera del cuerpo de la función da como resultado un error (o undefined si el nombre de la función se declaró previamente mediante una instrucción var).

Por ejemplo:
var y = function x() {};

```
alert(x); // arroja un error
```

El nombre de la función también aparece cuando la función se serializa vía el método de la Function 'toString'.

Por otro lado, la variable a la que se asigna la función está limitada sólo por su ámbito, que está garantizado para incluir el ámbito en el que se declara la función.

Como muestra el ejemplo anterior, el nombre de la función puede ser diferente de la variable a la que se asigna la función. No tienen relación entre sí.

Una declaración de función también crea una variable con el mismo nombre que el nombre de la función. Por lo tanto, a diferencia de las definidas por las expresiones de función, las funciones definidas por las declaraciones de función se puede acceder por su nombre en el ámbito que se definieron en:

```
function x() {}
alert(x); // salida x serializado en un string
```

El siguiente ejemplo muestra cómo los nombres de las funciones no están relacionados con las variables a las que están asignadas las funciones. Si una "variable de función" se asigna a otro valor, seguirá teniendo el mismo nombre de función:

```
function foo() {}
alert(foo); // el string  contiene el nombre
        // de la función "foo"
var bar = foo;
alert(bar); // el string todavía contiene el nombre
        // de la función "foo"
```

Dado que la función en realidad no tiene un nombre, anonymous no es una variable que se puede acceder dentro de la función. Por ejemplo, lo siguiente resultaría en un error:

```
var foo = new Function("alert(anonymous);");
foo();
```

A diferencia de las funciones definidas por expresiones de función o constructores Function se puede utilizar una función definida por una declaración de función antes de la propia declaración de la función. Por ejemplo:

```
foo(); // alerts FOO!
function foo() {
  alert('FOO!');
}
```

Una función definida por una expresión de función hereda el ámbito actual. Es decir, la función forma un cierre.

Por otro lado, una función definida por un constructor de Function no hereda ningún ámbito que no sea el ámbito global (que todas las funciones heredan).

Las funciones definidas por expresiones de función y declaraciones de función son analizadas una sola vez, mientras que las definidas por el constructor de Function no lo son. Es decir, la cadena de cuerpo de función pasada al constructor de Function debe ser analizada cada vez que se evalúa.

Aunque una expresión de función crea un cierre cada vez, el cuerpo de la función no es “parseado”, por lo que las expresiones de función son más rápidas que "new Function(...)". Por lo tanto, el constructor de la Function debe evitarse siempre que sea posible.

Una declaración de función es muy fácilmente convertida en una expresión de función. Una declaración de función deja de ser una cuando:

Se convierte en parte de una expresión
Ya no es un "elemento fuente" de una función o el propio script. Un "elemento de origen" es una sentencia no anidada en el script o un cuerpo de función:

```
var x = 0;               // elemento fuente
if (x == 0) {            // elemento fuente
  x = 10;                // no es un elemento fuente
  function boo() {}      // no es un elemento fuente
}
function foo() {         // elemento fuente
  var y = 20;            // elemento fuente
  function bar() {}      // elemento fuente
  while (y == 10) {      // elemento fuente
    function blah() {} // no es un elemento fuente
    y++;                 // no es un elemento fuente
  }
}
```

Ejemplos:

```
// function declaración
function foo() {}

// expresión de una función
(function bar() {})

// expresión de una función
x = function hello() {}
if (x) {
  // expresión de la función
  function world() {}
}
// instrucción de la función
function a() {
  // instrucción de la función
  function b() {}
  if (0) {
    // expresión de la función
    function c() {}
```

```
  }
}
```

Definición condicional de una función
Las funciones se pueden definir de forma condicional utilizando expresiones de función o el constructor Function.

En la siguiente secuencia de comandos, la función zero nunca se define y no se puede invocar, porque 'if (0)' se evalúa como false:

```
if (0)
  function zero() {
    document.writeln("Esto es zero.");
  }
```
Si se cambia el script para que la condición se convierta en 'if (1)', se define la función zero.

Nota: Aunque esto parece una declaración de función, ésta es en realidad una expresión de función ya que está anidada dentro de otra instrucción.

Funciones como manejadores de eventos

En JavaScript, los controladores de eventos DOM son funciones. Las funciones se pasan un objeto de evento como el primer y único parámetro. Como cualquier otro parámetro, si el objeto del evento no necesita ser utilizado, puede omitirse en la lista de parámetros formales.

Los posibles objetivos de eventos en un documento HTML incluyen: window (Window objects("objeto de ventana"), including frames("marcos")), document (HTMLDocument objects("objetos HTMLDocument")), y elementos (Element objects("objetos Elemento")). En el HTML DOM, los destinos de evento tienen propiedades de controlador de eventos. Estas propiedades son nombres de eventos en minúsculas con prefijo "on", por ej: onfocus. Los eventos DOM Level 2 Events proporcionan una forma alternativa y más sólida de agregar oyentes de eventos.

Los eventos son parte del DOM, no de JavaScript. (JavaScript simplemente proporciona un enlace al DOM.)

El ejemplo siguiente asigna una función a un manejador de eventos de "foco"("focus") de ventana.

```
window.onfocus = function() {
  document.body.style.backgroundColor = 'white';
}
```

Si se asigna una función a una variable, puede asignar la variable a un controlador de eventos. El siguiente código asigna una función a la variable setBGColor.

```
var setBGColor = new Function("document.body.style.backgroundColor = 'white';");
```

Al igual que cualquier otra propiedad que se refiere a una función, el controlador de eventos puede actuar como un método, y this se refiere al elemento que contiene el

controlador de eventos. En el ejemplo siguiente, se llama a la función referida por onfocus con this igual a window.

```
window.onfocus();
```

Un error común en JavaScript es el añadir paréntesis y / o parámetros al final de la variable, es decir, llamar al manejador de eventos cuando lo asigna. La adición de estos paréntesis asignará el valor devuelto al llamar al manejador de eventos, que a menudo es undefined (si la función no devuelve nada), en lugar del controlador de eventos en sí:

```
document.form1.button1.onclick = setBGColor();
```

Para pasar parámetros a un manejador de eventos, el manejador debe ser envuelto en otra función de la siguiente manera:

```
document.form1.button1.onclick = function() {
  setBGColor('Algun valor');
};
```

Variables locales dentro de las funciones

arguments: Objeto similar a una matriz que contiene los argumentos pasados a la función en ejecución.

arguments.callee: Especifica la función en ejecución.

arguments.caller: Especifica la función que invocó la función en ejecución.

arguments.length: Especifica el número de argumentos pasados a la función.

Ejemplos
1) Devolver un número con formato
La siguiente función devuelve una cadena que contiene la representación formateada de un número rellenado con ceros a la izquierda.

```
// Esta función devuelve una cadena rellenada con ceros a la izquierda

function padZeros(num, totalLen) {
  var numStr = num.toString();             // Inicializa un valor de retorno como cadena
  var numZeros = totalLen - numStr.length; // Calcula el no. de ceros
  for (var i = 1; i <= numZeros; i++) {
    numStr = "0" + numStr;
  }
  return numStr;
}
```

Las siguientes sentencias llaman a la función padZeros.

```
var resultado;
resultado = padZeros(42,4); // retorna "0042"
resultado = padZeros(42,2); // retorna "42"
resultado = padZeros(5,4);  // retorna "0005"
```

2) Determinar si existe una función

Puede determinar si existe una función utilizando el operador typeof. En el ejemplo siguiente, se realiza una prueba para determinar si el objeto window tiene una propiedad llamada noFunc que es una función. Si es así, se utiliza; de lo contrario, se tomarán otras medidas.

```
 if ('function' == typeof window.noFunc) {
   // utiliza noFunc()
 } else {
   // hacer algo más
 }
```

Nota: Tenga en cuenta que en la prueba if, se utiliza una referencia a noFunc aquí no hay paréntesis "()" después del nombre de la función para que la función real no se llame.

Callback

Una función de callback es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción.

Ejemplo:

```
function saludar(nombre) { // ←-- definición de la función
  alert('Hola ' + nombre);
}

function procesarEntradaUsuario(callback) {
  var nombre = prompt('Por favor ingresa tu nombre.');
  callback(nombre);
}

procesarEntradaUsuario(saludar); // ←-- función pasada como parámetro
```

El ejemplo anterior es una callback sincrónica, ya que se ejecuta inmediatamente.

Sin embargo, tenga en cuenta que las callbacks a menudo se utilizan para continuar con la ejecución del código después de que se haya completado una operación asincrónica — estas se denominan devoluciones de llamada asincrónicas.

Closures

Una clausura o closure es una función que guarda referencias del estado adyacente, o sea, permite acceder al ámbito de una función exterior desde una función interior. En JavaScript, las clausuras se crean cada vez que una función es creada.

Ámbito léxico
Consideremos el siguiente ejemplo:

```
function iniciar() {
  var nombre = "internet";  // La variable nombre es una variable local creada por iniciar.
  function mostrarNombre() {  // La función mostrarNombre es una función interna, una
clausura.
    alert(nombre);  // Usa una variable declarada en la función externa.
  }
  mostrarNombre();
}
iniciar();
```

La función iniciar() crea una variable local llamada nombre y una función interna llamada mostrarNombre(). Por ser una función interna, esta última solo está disponible dentro del cuerpo de iniciar(). Notemos a su vez que mostrarNombre() no tiene ninguna variable propia; pero, dado que las funciones internas tienen acceso a las variables de las funciones externas, mostrarNombre() puede acceder a la variable nombre declarada en la función iniciar().

```
function creaFunc() {
  var nombre = "internet";
  function muestraNombre() {
    alert(nombre);
  }
  return muestraNombre;
}

var miFunc = creaFunc();
miFunc();
```

Si se ejecuta este código tendrá exactamente el mismo efecto que el ejemplo anterior: se mostrará el texto "internet" en un cuadro de alerta de Javascript. Lo que lo hace diferente es que la función externa nos ha devuelto la función interna muestraNombre() antes de ejecutarla.

Puede parecer poco intuitivo que este código funcione. Normalmente, las variables locales dentro de una función sólo existen mientras dura la ejecución de dicha función. Una vez que creaFunc() haya terminado de ejecutarse, es razonable suponer que no se pueda ya acceder a la variable nombre. Dado que el código funciona como se esperaba, esto obviamente no es el caso.

La solución a este rompecabezas es que miFunc se ha convertido en un closure. Un closure es un tipo especial de objeto que combina dos cosas: una función, y el entorno en

que se creó esa función. El entorno está formado por las variables locales que estaban dentro del alcance en el momento que se creó el closure. En este caso, miFunc es un closure que incorpora tanto la función muestraNombre como el string "internet" que existían cuando se creó el closure.

Este es un ejemplo un poco más interesante: una función creaSumador:

```
function creaSumador(x) {
  return function(y) {
    return x + y;
  };
}

var suma5 = creaSumador(5);
var suma10 = creaSumador(10);

console.log(suma5(2));  // muestra 7
console.log(suma10(2)); // muestra 12
```

En este ejemplo, hemos definido una función creaSumador(x) que toma un argumento único x y devuelve una nueva función. Esa nueva función toma un único argumento y, devolviendo la suma de x + y.

En esencia, creaSumador es una fábrica de función: crea funciones que pueden sumar un valor específico a su argumento. En el ejemplo anterior utilizamos nuestra fábrica de función para crear dos nuevas funciones: una que agrega 5 a su argumento y otra que agrega 10.

suma5 y suma10 son ambos closures. Comparten la misma definición de cuerpo de función, pero almacenan diferentes entornos. En el entorno suma5, x es 5. En lo que respecta a suma10, x es 10.

Closures prácticos

Hasta aquí hemos visto teoría, pero ¿son los closures realmente útiles? Vamos a considerar sus implicaciones prácticas. Un closure permite asociar algunos datos (el entorno) con una función que opera sobre esos datos. Esto tiene evidentes paralelismos con la programación orientada a objetos, en la que los objetos nos permiten asociar algunos datos (las propiedades del objeto) con uno o más métodos.

En consecuencia, puede utilizar un closure en cualquier lugar en el que normalmente pondría un objeto con un solo método.

En la web hay situaciones habituales en las que aplicarlos. Gran parte del código JavaScript para web está basado en eventos: definimos un comportamiento y lo conectamos a un evento que es activado por el usuario (como un click o pulsación de una tecla). Nuestro código generalmente se adjunta como una devolución de llamada (callback): que es una función que se ejecuta en respuesta al evento.

Aquí está un ejemplo práctico: Supongamos que queremos añadir algunos botones a una página para ajustar el tamaño del texto. Una manera de hacer esto es especificar el tamaño de fuente del elemento body en píxeles y, a continuación, ajustar el tamaño de los demás elementos de la página (como los encabezados) utilizando la unidad relativa em:

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}
h2 {
  font-size: 1.2em;
}
```
Nuestros botones interactivos de tamaño de texto pueden cambiar la propiedad font-size del elemento body, y los ajustes serán aplicados por los otros elementos de la página gracias a las unidades relativas.

Aquí está el código JavaScript:

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);
```
size12, size14 y size16 ahora son funciones que cambian el tamaño del texto de body a 12, 14 y 16 pixels, respectivamente. Podemos conectarlos a botones (en este caso enlaces) de la siguiente forma:

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

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

JavaScript no proporciona una forma nativa de hacer esto, pero es posible emular métodos privados utilizando closures. Los métodos privados no son sólo útiles para restringir el acceso al código, también proporcionan una poderosa manera de administrar tu espacio de nombres global, evitando que los métodos no esenciales compliquen la interfaz pública del código.

Aquí vemos cómo definir algunas funciones públicas que pueden acceder a variables y funciones privadas utilizando closures. A esto se le conoce también como el patrón módulo:

```
var Counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
})();

alert(Counter.value()); /* Muestra 0 */
Counter.increment();
Counter.increment();
alert(Counter.value()); /* Muestra 2 */
Counter.decrement();
alert(Counter.value()); /* Muestra 1 */
```

En los ejemplos anteriores cada closure ha tenido su propio entorno; aquí creamos un único entorno compartido por tres funciones: Counter.increment, Counter.decrement y Counter.value.

El entorno compartido se crea en el cuerpo de una función anónima, que se ejecuta en el momento que se define. El entorno contiene dos elementos privados: una variable llamada privateCounter y una función llamada changeBy. No se puede acceder a ninguno de estos elementos privados directamente desde fuera de la función anónima. Se accede a ellos por las tres funciones públicas que se devuelven desde el contenedor anónimo.

Esas tres funciones públicas son closures que comparten el mismo entorno. Gracias al ámbito léxico de Javascript, cada uno de ellas tienen acceso a la variable privateCounter y a la función changeBy.

En este caso hemos definido una función anónima que crea un contador, y luego la llamamos inmediatamente y asignamos el resultado a la variable Counter. Pero podríamos almacenar esta función en una variable independiente y utilizarlo para crear varios contadores:

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
```

```
    changeBy(1);
  },
  decrement: function() {
    changeBy(-1);
  },
  value: function() {
    return privateCounter;
  }
 }
};

var Counter1 = makeCounter();
var Counter2 = makeCounter();
alert(Counter1.value()); /* Muestra 0 */
Counter1.increment();
Counter1.increment();
alert(Counter1.value()); /* Muestra 2 */
Counter1.decrement();
alert(Counter1.value()); /* Muestra 1 */
alert(Counter2.value()); /* Muestra 0 */
```

Tené en cuenta que cada uno de los dos contadores mantiene su independencia del otro. Su entorno durante la llamada de la función makeCounter() es diferente cada vez. La variable del closure llamada privateCounter contiene una instancia diferente cada vez.

Utilizar closures de este modo proporciona una serie de beneficios que se asocian normalmente con la programación orientada a objectos, en particular la encapsulación y la ocultación de datos.

Creando closures en loops:
Antes de la introducción de la palabra clave let en JavaScript 1.7, un problema común con closures ocurría cuando se creaban dentro de un bucle 'loop'. Veamos el siguiente ejemplo:

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>
<p>Age: <input type="text" id="age" name="age"></p>
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Dirección de correo electrónico'},
      {'id': 'name', 'help': 'Nombre completo'},
      {'id': 'age', 'help': 'Edad (debes tener más de 16 años)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
```

```
    showHelp(item.help);
  }
 }
}

setupHelp();
```

El array helpText define tres avisos de ayuda, cada uno asociado con el ID de un campo de entrada en el documento. El bucle recorre estas definiciones, enlazando un evento onfocus a cada uno que muestra el método de ayuda asociada.

Si probás este código, resulta que no funciona como esperabas. Independientemente del campo en el que se haga foco, siempre se mostrará el mensaje de ayuda relativo a la edad.

La razón de esto es que las funciones asignadas a onfocus son closures; que constan de la definición de la función y del entorno abarcado desde el ámbito de la función setupHelp. Se han creado tres closures, pero todos comparten el mismo entorno. En el momento en que se ejecutan las funciones callback de onfocus, el bucle ya ha finalizado y la variable item (compartida por los tres closures) ha quedado apuntando a la última entrada en la lista de helpText.

En este caso, una solución es utilizar más closures: concretamente añadiendo una fábrica de función como se ha descrito anteriormente:

```
function showHelp(help) {
 document.getElementById('help').innerHTML = help;
}

function makeHelpCallback(help) {
 return function() {
  showHelp(help);
 };
}

function setupHelp() {
 var helpText = [
    {'id': 'email', 'help': 'Dirección de correo electrónico'},
    {'id': 'name', 'help': 'Nombre completo'},
    {'id': 'age', 'help': 'Edad (debes tener más de 16 años)'}
  ];

 for (var i = 0; i < helpText.length; i++) {
  var item = helpText[i];
  document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
 }
}

setupHelp();
```

Esto si funciona como se esperaba. En lugar de los tres callbacks compartiendo el mismo entorno, la función makeHelpCallback crea un nuevo entorno para cada uno en el que help se refiere a la cadena correspondiente del array helpText.

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

Por ejemplo, cuando se crea un nuevo objeto/clase, los métodos normalmente deberían asociarse al prototipo del objeto en vez de definirse en el constructor del objeto. La razón es que con este último sistema, cada vez que se llama al constructor (cada vez que se crea un objeto) se tienen que reasignar los métodos.

Scope

Se refiere al contexto actual de ejecución. El contexto en el que los valores y las expresiones son "visibles" o pueden ser referenciados. Si una variable u otra expresión no está "en el Scope o alcance actual", entonces no está disponible para su uso.

Los Scope también se pueden superponer en una jerarquía, de modo que los Scope secundarios tengan acceso a los ámbitos primarios, pero no al revés.

Una función sirve como un cierre en JavaScript y, por lo tanto, crea un ámbito, de modo que (por ejemplo) no se puede acceder a una variable definida exclusivamente dentro de la función desde fuera de la función o dentro de otras funciones.

Por ejemplo, lo siguiente no es válido:

```
function exampleFunction() {
    var x = "declarada dentro de la función"; // x solo se puede utilizar en exampleFunction
    console.log("funcion interna");
    console.log(x);
}

console.log(x);  // error
```

Sin embargo, el siguiente código es válido debido a que la variable se declara fuera de la función, lo que la hace global:

```
var x = "función externa declarada";

exampleFunction();

function exampleFunction() {
    console.log("funcion interna");
    console.log(x);
}

console.log("funcion externa");
console.log(x);
```

Arrows Functions

Una expresión de función flecha es una alternativa compacta a una expresión de función tradicional, pero es limitada y no se puede utilizar en todas las situaciones.

Diferencias y limitaciones:

No tiene sus propios enlaces a this o super y no se debe usar como métodos.
No tiene argumentos o palabras clave new.target.
No apta para los métodos call, apply y bind, que generalmente se basan en establecer un ámbito o alcance
No se puede utilizar como constructor.
No se puede utilizar yield dentro de su cuerpo.

```
const materials = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

Comparación de funciones tradicionales con funciones flecha

Desglose de una "función tradicional" hasta la "función flecha" más simple:
Nota: Cada paso a lo largo del camino es una "función flecha" válida

```
// Función tradicional
function (a){
  return a + 100;
}

// Desglose de la función flecha

// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y el corchete de apertura.
(a) => {
  return a + 100;
}

// 2. Quita los corchetes del cuerpo y la palabra "return" — el return está implícito.
(a) => a + 100;

// 3. Suprime los paréntesis de los argumentos
a => a + 100;
```

Como se muestra arriba, los { corchetes }, (paréntesis) y "return" son opcionales, pero pueden ser obligatorios.

Por ejemplo, con varios argumentos o ningún argumento, tenés que volver a introducir paréntesis alrededor de los argumentos:

```
// Función tradicional
function (a, b){
  return a + b + 100;
}

// Función flecha
(a, b) => a + b + 100;

// Función tradicional (sin argumentos)
let a = 4;
let b = 2;
function (){
  return a + b + 100;
}

// Función flecha (sin argumentos)
let a = 4;
let b = 2;
() => a + b + 100;
```

Del mismo modo, si el cuerpo requiere líneas de procesamiento adicionales, tenés que volver a introducir los corchetes más el "return" (las funciones flecha no adivinan qué o cuándo querés "volver"):

```
// Función tradicional
function (a, b){
  let edad = 42;
  return a + b + edad;
}

// Función flecha
(a, b) => {
  let edad = 42;
  return a + b + edad;
}
```

Y finalmente, en las funciones con nombre tratamos las expresiones de flecha como variables

```
// Función tradicional
function suma (a){
  return a + 100;
}

// Función flecha
let suma = a => a + 100;
```

Sintaxis básica
Un parámetro. Con una expresión simple no se necesita return:

param => expression
Varios parámetros requieren paréntesis. Con una expresión simple no se necesita return:

(param1, paramN) => expression
Las declaraciones de varias líneas requieren corchetes y return:

```
param => {
  let a = 1;
  return a + b;
}
```

Varios parámetros requieren paréntesis. Las declaraciones de varias líneas requieren corchetes y return:

```
(param1, paramN) => {
   let a = 1;
   return a + b;
}
```

Sintaxis avanzada
Para devolver una expresión de objeto literal, se requieren paréntesis alrededor de la expresión:

```
params => ({foo: "a"}) // devuelve el objeto {foo: "a"}
```

Los parámetros rest son compatibles:

```
(a, b, ...r) => expression
```

Se admiten los parámetros predeterminados:

```
(a=400, b=20, c) => expression
```

Desestructuración dentro de los parámetros admitidos:

```
([a, b] = [10, 20]) => a + b;  // el resultado es 30
({ a, b } = { a: 10, b: 20 }) => a + b; // resultado es 30
```

Descripción
Consulta también "ES6 en profundidad: funciones flecha" en hacks.mozilla.org.

"this" y funciones flecha
Una de las razones por las que se introdujeron las funciones flecha fue para eliminar complejidades del ámbito (this) y hacer que la ejecución de funciones sea mucho más intuitiva.

This se refiere a la instancia. Las instancias se crean cuando se invoca la palabra clave new. De lo contrario, this se establecerá —de forma predeterminada— en el ámbito o alcance de window.

En las funciones tradicionales de manera predeterminada this está en el ámbito de window:

```
window.age = 10; // <-- definición de age por primera vez
function Person() {
  this.age = 42; // <--  definición de age por segunda vez
```

```
  setTimeout(function () { // <-- La función tradicional se está ejecutando en el ámbito de window
    console.log("this.age", this.age); // genera "10" porque se ejecuta en el ámbito window
  }, 100);
}

var p = new Person();
```

Las funciones flecha no predeterminan this al ámbito o alcance de window, más bien se ejecutan en el ámbito o alcance en que se crean:

```
window.age = 10; // <-- acá
function Person() {
  this.age = 42; // <-- acá
  setTimeout(() => { // <-- Función flecha ejecutándose en el ámbito de "p" (una instancia de Person)
    console.log("this.age", this.age); // genera "42" porque la función se ejecuta en el ámbito de Person
  }, 100);
}

var p = new Person();
```

En el ejemplo anterior, la función flecha no tiene su propio this. Se utiliza el valor this del ámbito léxico adjunto; las funciones flecha siguen las reglas normales de búsqueda de variables. Entonces, mientras busca this que no está presente en el ámbito actual, una función flecha termina encontrando el this de su ámbito adjunto.

Relación con el modo estricto

Dado que this proviene del contexto léxico circundante, en el modo estricto se ignoran las reglas con respecto a this.

```
var f = () => {
    'use strict';
    return this;
};

f() === window; // o el objeto global
```

Todas las demás reglas del modo estricto se aplican normalmente.

Funciones flecha utilizadas como métodos

Como se indicó anteriormente, las expresiones de función flecha son más adecuadas para funciones que no son métodos. Observa qué sucede cuando intentas usarlas como métodos:

```
var obj = { // no crea un nuevo ámbito
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
```

```
    console.log(this.i, this);
  }
}

obj.b(); // imprime indefinido, Window {...} (o el objeto global)
obj.c(); // imprime 10, Object {…}
```

Las funciones flecha no tienen su propio this. Otro ejemplo que involucra Object.defineProperty():

```
var obj = {
  a: 10
};

Object.defineProperty(obj, 'b', {
  get: () => {
    console.log(this.a, typeof this.a, this); // indefinida 'undefined' Window {...} (o el objeto global)
    return this.a + 10; // representa el objeto global 'Window', por lo tanto 'this.a' devuelve 'undefined'
  }
});
```

call, apply y bind

Los métodos call, apply y bind NO son adecuados para las funciones flecha, ya que fueron diseñados para permitir que los métodos se ejecuten dentro de diferentes ámbitos, porque las funciones flecha establecen "this" según el ámbito dentro del cual se define la función flecha.

Por ejemplo, call, apply y bind funcionan como se esperaba con las funciones tradicionales, porque establecen el ámbito para cada uno de los métodos:

```
// ----------------------
// Ejemplo tradicional
// ----------------------
// Un objeto simplista con su propio "this".
var obj = {
    num: 100
}

// Establece "num" en window para mostrar cómo NO se usa.
window.num = 2020; // ¡Ay!

// Una función tradicional simple para operar en "this"
var add = function (a, b, c) {
  return this.num + a + b + c;
}

// call
var result = add.call(obj, 1, 2, 3) // establece el ámbito como "obj"
console.log(result) // resultado 106
```

```
// apply
const arr = [1, 2, 3]
var result = add.apply(obj, arr) // establece el ámbito como "obj"
console.log(result) // resultado 106

// bind
var result = add.bind(obj) // estable el ámbito como "obj"
console.log(result(1, 2, 3)) // resultado 106
```
Con las funciones flecha, dado que la función add esencialmente se crea en el ámbito del window (global), asumirá que this es window.

```
// ----------------------
// Ejemplo de flecha
// ----------------------

// Un objeto simplista con su propio "this".
var obj = {
    num: 100
}

// Establecer "num" en window para mostrar cómo se recoge.
window.num = 2020; // ¡Ay!

// Función flecha
var add = (a, b, c) => this.num + a + b + c;

// call
console.log(add.call(obj, 1, 2, 3)) // resultado 2026

// apply
const arr = [1, 2, 3]
console.log(add.apply(obj, arr)) // resultado 2026

// bind
const bound = add.bind(obj)
console.log(bound(1, 2, 3)) // resultado 2026
```

Quizás el mayor beneficio de usar las funciones flecha es con los métodos a nivel del DOM (setTimeout, setInterval, addEventListener) que generalmente requieren algún tipo de cierre, llamada, aplicación o vinculación para garantizar que la función se ejecute en el ámbito adecuado.

Ejemplo tradicional:

```
var obj = {
    count : 10,
    doSomethingLater : function (){
        setTimeout(function(){ // la función se ejecuta en el ámbito de window
            this.count++;
            console.log(this.count);
        }, 300);
```

```
    }
}

obj.doSomethingLater(); // la consola imprime "NaN", porque la propiedad "count" no está en el ámbito de window.
```

Ejemplo de flecha:

```
var obj = {
    count : 10,
    doSomethingLater : function(){ // las funciones flecha no son adecuadas para métodos
        setTimeout( () => { // dado que la función flecha se creó dentro del "obj", asume el "this" del objeto
            this.count++;
            console.log(this.count);
        }, 300);
    }
}

obj.doSomethingLater();
```

Sin enlace de arguments

Las funciones flecha no tienen su propio objeto arguments. Por tanto, en este ejemplo, arguments simplemente es una referencia a los argumentos del ámbito adjunto:

```
var arguments = [1, 2, 3];
var arr = () => arguments[0];

arr(); // 1

function foo(n) {
  var f = () => arguments[0] + n; // Los argumentos implícitos de foo son vinculantes. arguments[0] es n
  return f();
}

foo(3); // 6
```

En la mayoría de los casos, usar parámetros rest es una buena alternativa a usar un objeto arguments.

```
function foo(n) {
  var f = (...args) => args[0] + n;
  return f(10);
}

foo(1); // 11
```

Uso del operador new

Las funciones flecha no se pueden usar como constructores y arrojarán un error cuando se usen con new.

```
var Foo = () => {};
var foo = new Foo(); // TypeError: Foo no es un constructor
```

Uso de la propiedad prototype
Las funciones flecha no tienen una propiedad prototype.

```
var Foo = () => {};
console.log(Foo.prototype); // undefined
```

Uso de la palabra clave yield
La palabra clave yield no se puede utilizar en el cuerpo de una función flecha (excepto cuando está permitido dentro de las funciones anidadas dentro de ella). Como consecuencia, las funciones flecha no se pueden utilizar como generadores.

Cuerpo de función
Las funciones flecha pueden tener un "cuerpo conciso" o el "cuerpo de bloque" habitual.

En un cuerpo conciso, solo se especifica una expresión, que se convierte en el valor de retorno implícito. En el cuerpo de un bloque, debes utilizar una instrucción return explícita.

```
var func = x => x * x;
// sintaxis de cuerpo conciso, "return" implícito

var func = (x, y) => { return x + y; };
// con cuerpo de bloque, se necesita un "return" explícito
```

Devolver objetos literales
Ten en cuenta que devolver objetos literales utilizando la sintaxis de cuerpo conciso params => {object: literal} no funcionará como se esperaba.

```
var func = () => { foo: 1 };
// ¡Llamar a func() devuelve undefined!

var func = () => { foo: function() {} };
// SyntaxError: la declaración function requiere un nombre
```

Esto se debe a que el código entre llaves ({}) se procesa como una secuencia de declaraciones (es decir, foo se trata como una etiqueta, no como una clave en un objeto literal).

Debes envolver el objeto literal entre paréntesis:

```
var func = () => ({ foo: 1 });
```

Saltos de línea
Una función flecha no puede contener un salto de línea entre sus parámetros y su flecha.

```
var func = (a, b, c)
  => 1;

// SyntaxError: expresión esperada, arroja: '=>'
```

Sin embargo, esto se puede modificar colocando el salto de línea después de la flecha o usando paréntesis/llaves como se ve a continuación para garantizar que el código se

mantenga más claro en su lectura. También podés poner saltos de línea entre argumentos.

```
var func = (a, b, c) =>
  1;

var func = (a, b, c) => (
  1
);

var func = (a, b, c) => {
  return 1
};

var func = (
  a,
  b,
  c
) => 1;

// no se lanza SyntaxError
```

Orden de procesamiento

Aunque la flecha en una función flecha no es un operador, las funciones flecha tienen reglas de procesamiento especiales que interactúan de manera diferente con prioridad de operadores en comparación con las funciones regulares.

```
let callback;

callback = callback || function() {}; // ok

callback = callback || () => {};
// SyntaxError: argumentos de función flecha no válidos

callback = callback || (() => {});    // bien
```

Ejemplos
Uso básico

```
// Una función flecha vacía devuelve undefinided
let empty = () => {};

(() => 'foobar')();
// Devuelve "foobar"
// (esta es una expresión de función invocada inmediatamente)

var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10

let max = (a, b) => a > b ? a : b;

// Fácil filtrado de arreglos, mapeo,
```

```
var arr = [5, 6, 13, 0, 1, 18, 23];

var sum = arr.reduce((a, b) => a + b);
// 66

var even = arr.filter(v => v % 2 == 0);
// [6, 0, 18]

var double = arr.map(v => v * 2);
// [10, 12, 26, 0, 2, 36, 46]

// Cadenas de promesas más concisas
promise.then(a => {
  // ...
}).then(b => {
  // ...
});

// Funciones flecha sin parámetros que son visualmente más fáciles de procesar
setTimeout( () => {
  console.log('sucederá antes');
  setTimeout( () => {
    // código más profundo
    console.log ('Sucederá más tarde');
  }, 1);
}, 1);
```

Modo estricto

Todos los navegadores modernos admiten la ejecución de JavaScript en "Modo estricto".

En el "Modo estricto", las variables no declaradas no son automáticamente globales.

Variables globales en HTML

Con JavaScript, el alcance global es el entorno completo de JavaScript.

En HTML, el ámbito global es el objeto de ventana. Todas las variables globales pertenecen al objeto de ventana.

Ejemplo

```
var carName = "Volvo";

// code here can use window.carName
```

Advertencias

NO cree variables globales a menos que lo desee.

Sus variables (o funciones) globales pueden sobrescribir las variables (o funciones) de la ventana.

Cualquier función, incluido el objeto de ventana, puede sobrescribir sus variables y funciones globales.

La vida útil de las variables de JavaScript

La vida útil de una variable de JavaScript comienza cuando se declara.

Las variables locales se eliminan cuando se completa la función.

En un navegador web, las variables globales se eliminan cuando cierra la ventana (o pestaña) del navegador.

Argumentos de función

Los argumentos de la función (parámetros) funcionan como variables locales dentro de las funciones.

La instrucción **let** declara una variable de alcance local con ámbito de bloque (block scope https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/block), la cual, opcionalmente, puede ser inicializada con algún valor.

Sintaxis

let var1 [= valor1] [, var2 [= valor2]] [, ..., varN [= valorN]];

Parámetros

var1, var2, …, varN

Los nombres de la variable o las variables a declarar. Cada una de ellas debe ser un identificador legal de JavaScript

value1, value2, …, valueN

Por cada una de las variables declaradas puedes, opcionalmente, especificar su valor inicial como una expresión legal JavaScript.

Descripción

let te permite declarar variables limitando su alcance (scope) al bloque, declaración, o expresión donde se está usando. Lo anterior diferencia let de la palabra reservada var https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var, la cual define una variable global o local en una función sin importar el ámbito del bloque.

Alcance (scope) a nivel de bloque con let

Usar la palabra reservada **let** para definir variables dentro de un bloque.

if (x > y) { let gamma = 12.7 + y; i = gamma * x; }

Es posible usar definiciones let para asociar código en extensiones con un pseudo-espacio-de-nombre (pseudo-namespace).

let Cc = Components.classes, Ci = Components.interfaces;

let puede ser útil para escribir código más limpio cuando usamos funciones internas.

var list = document.getElementById("list"); for (var i = 1; i <= 5; i++) { var item = document.createElement("LI"); item.appendChild(document.createTextNode("Item " + i)); let j = i; item.onclick = function (ev) { console.log("Item " + j + " is clicked."); }; list.appendChild(item); }

El ejemplo anterior trabaja como se espera porque las cinco instancias de la función (anónima) interna hacen referencia a cinco diferentes instancias de la variable j. Nótese que esto no funcionaría como se espera si reemplazamos let con var o si removemos la variable j y simplemente usamos la variable i dentro de la función interna.

Reglas de alcance

Variables declaradas por **let** tienen por alcance el bloque en el que se han definido, así mismo, como en cualquier bloque interno. De esta manera, let trabaja muy parecido a var.

La más notable diferencia es que el alcance de una variable var es la función contenedora:

function varTest() {

var x = 31;

if (true) {

var x = 71; // ¡misma variable!

console.log(x); // 71

}

console.log(x); // 71

}

```
function letTest() {

let x = 31;

if (true) {

let x = 71; // variable diferente

console.log(x); // 71

}

console.log(x); // 31

}

// llamamos a las funciones varTest(); letTest();
```

En el nivel superior de un programa y funciones, let , a diferencia de var, no crea una propiedad en el objeto global, por ejemplo:

```
var x = 'global';

let y = 'global';

console.log(this.x); // "global"

console.log(this.y); // undefined
```

La salida de este código desplegaría "global" una vez.

Zona muerta temporal y errores con let

La redeclaración de la misma variable bajo un mismo ámbito léxico terminaría en un error de tipo SyntaxError https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/SyntaxError. Esto también es extensible si usamos var dentro del ámbito léxico. Esto nos salvaguarda de re-declarar una variable accidentalmente y que no era posible solo con var.

```
if (x) {

let foo;

let foo; // Terminamos con un SyntaxError.

}

if (x) {

let foo;

var foo; // Terminamos con un SyntaxError.

}
```

En ECMAScript 2015, let no eleva la variable a la parte superior del bloque. Si se hace una referencia a la variable declarada con let (let foo) antes de su declaración, terminaríamos con un error de tipo ReferenceError (al contrario de la variable declarada con var, que tendrá el valor undefined), esto porque la variables vive en una "zona muerta temporal" desde el inicio del bloque hasta que la declaración ha sido procesada.

```
function do_something() {

console.log(bar); // undefined

console.log(foo); // ReferenceError: foo no está definido

var bar = 1;

let foo = 2;

}
```

Es posible encontrar errores en bloques de control **switch** debido a que solamente existe un block subyacente.

```
switch (x) {

case 0:

let foo;
```

```
break;

case 1:

let foo; // Terminamos con un error de tipo SyntaxError.

// esto debido a la redeclaracion

break;

}
```

Otro ejemplo de zona muerta temporal combinada con ámbito léxico

Debido al alcance léxico, el identificador num dentro de la expresión (num + 55) se evalúa como num del bloque if, y no como la variable num con el valor 33 que esta por encima

En esa misma línea, el num del bloque if ya se ha creado en el ámbito léxico, pero aún no ha alcanzado (y terminado) su inicialización (que es parte de la propia declaración): todavía está en la zona muerta temporal.

```
function prueba(){

var num = 33;

if (true) {

let num = (num + 55);

//ReferenceError: no se puede acceder a la declaración léxica ‘num’ antes de la inicialización

}

}

prueba();
```

Ejemplos

let vs **var**

Cuando usamos let dentro de un bloque, podemos limitar el alcance de la variable a dicho bloque. Notemos la diferencia con var, cuyo alcance reside dentro de la función donde ha sido declarada la variable.

```
var a = 5;

var b = 10;

if (a === 5) {

let a = 4; // El alcance es dentro del bloque

if var b = 1; // El alcance es global

console.log(a); // 4

console.log(b); // 1

}

console.log(a); // 5

console.log(b); // 1

}
```

let en bucles

Es posible usar la palabra reservada let para enlazar variables con alcance local dentro del alcance de un bucle en lugar de usar una variable global (definida usando var) para dicho propósito.

```
for (let i = 0; i<10; i++) {

console.log(i); // 0, 1, 2, 3, 4 ... 9

}

console.log(i); // ReferenceError: i is not defined
```

Extensiones let

Extensiones let no-estándar

Bloques let

La sintaxis del bloque y expresión let es no-estándar y será desechado en un futuro. ¡No deben ser usados! ver error 1023609 y error 1167029 para mas detalles.

Un bloque let provee una manera de asociar valores con variables dentro del alcance de un bloque sin afectar el valor de variables con nombre similar fuera del bloque.

Sintaxis

let (var1 [= value1] [, var2 [= value2]] [, ..., varN [= valueN]]) {declaración};

Descripción

El bloque let provee alcance local para las variables. Funciona enlazando cero o más variables en el alcance léxico de un solo bloque de código; de otra manera, es exactamente lo mismo que una declaración de bloque. Hay que notar particularmente que el alcance de una variable declarada dentro de un bloque let usando var es equivalente a declarar esa variable fuera del bloque let; dicha variable aún tiene alcance dentro de la función. Al usar la sintaxis de bloque let, los paréntesis siguientes a let son requeridos. Una falla al incluir dichos paréntesis resultará en un error de sintaxis.

Ejemplo

```
var x = 5;

var y = 0;

let (x = x+10, y = 12) {

console.log(x+y); // 27

}

console.log(x + y); // 5
```

Las reglas para el bloque de código son las mismas que para cualquier otro bloque de código en JavaScript. Es posible tener sus propias variables locales usando declaraciones let en dicho bloque.

Reglas de alcance

El alcance de las variables definidas usando let es el mismo bloque let, así como cualquier bloque interno contenido dentro de el bloque, a menos que esos bloques internos definan variables con el mismo nombre.

expresiones let

Soporte de expresiones let ha sido removido en Gecko 41 (error 1023609 https://bugzilla.mozilla.org/show_bug.cgi?id=1023609).

Una expresión **let** permite establecer variables con alcance dentro de una expresión.

Sintaxis

let (var1 [= value1] [, var2 [= value2]] [, ..., varN [= valueN]]) expression;

Ejemplo

Podemos usar let para establecer variables que tienen como alcance solo una expresión:

var a = 5;

let(a = 6) console.log(a); // 6

console.log(a); // 5

Reglas de alcance

Dada la expresión let siguiente:

let (decls) expr

Existe un bloque implícito creado alrededor de expr.

Arrays

El objeto Array de JavaScript es un objeto global que es usado en la construcción de arrays, que son objetos tipo lista de alto nivel.

Descripción

Los arrays son objetos similares a una lista cuyo prototipo proporciona métodos para efectuar operaciones de recorrido y de mutación. Tanto la longitud como el tipo de los elementos de un array son variables. Dado que la longitud de un array puede cambiar en cualquier momento, y los datos se pueden almacenar en ubicaciones no contiguas, no hay garantía de que los arrays de JavaScript sean correlativos y de extensión fija. Esto depende de cómo el programador elija usarlos. En general estas características son cómodas, aunque si en algún caso particular, no resultan deseables, se puede considerar el uso de arrays con tipo.

Operaciones habituales

Crear un Array

```
let frutas = ["Manzana", "Banana"]

console.log(frutas.length)
// 2
```

Acceder a un elemento de Array mediante su índice

```
let primero = frutas[0]
// Manzana

let ultimo = frutas[frutas.length - 1]
// Banana
```

Recorrer un Array

```
frutas.forEach(function(elemento, indice, array) {
   console.log(elemento, indice);
})
// Manzana 0
// Banana 1
```

Añadir un elemento al final de un Array

```
let nuevaLongitud = frutas.push('Naranja') // Añade "Naranja" al final
// ["Manzana", "Banana", "Naranja"]
```

Eliminar el último elemento de un Array

```
let ultimo = frutas.pop() // Elimina "Naranja" del final
// ["Manzana", "Banana"]
```

Añadir un elemento al principio de un Array

```
let nuevaLongitud = frutas.unshift('Fresa') // Añade "Fresa" al inicio
// ["Fresa" ,"Manzana", "Banana"]
```

Eliminar el primer elemento de un Array

```
let primero = frutas.shift() // Elimina "Fresa" del inicio
// ["Manzana", "Banana"]
```

Encontrar el índice de un elemento del Array

```
frutas.push('Pera')
// ["Manzana", "Banana", "Pera"]

let pos = frutas.indexOf('Banana') // (pos) es la posición para abreviar
// 1
```

Eliminar un único elemento mediante su posición

Ejemplo:
Eliminamos "Banana" del array pasándole dos parámetros: la posición del primer elemento que se elimina y el número de elementos que queremos eliminar. De esta forma, .splice(pos, 1) empieza en la posición que nos indica el valor de la variable pos y elimina 1 elemento. En este caso, como pos vale 1, elimina un elemento comenzando en la posición 1 del array, es decir "Banana".

```
let elementoEliminado = frutas.splice(pos, 1)
// ["Manzana", "Pera"]
```

Eliminar varios elementos a partir de una posición

Nota:
Con .splice() no solo se puede eliminar elementos del array, si no que también podemos extraerlos guardándolo en un nuevo array. Al hacer esto estaríamos modificando el array de origen.

```
let vegetales = ['Repollo', 'Coliflor', 'Zapallo', 'Zanahoria']
console.log(vegetales)
// ["Repollo", "Coliflor", "Zapallo", "Zanahoria"]

let pos = 1, numElementos = 2

let elementosEliminados = vegetales.splice(pos, numElementos)
// ["Coliflor", "Zapallo"] ==> Lo que se ha guardado en "elementosEliminados"

console.log(vegetales)
// ["Zapallo", "Zanahoria"] ==> Lo que actualmente tiene "vegetales"
```

Copiar un Array

```
let copiaArray = vegetales.slice();
// ["Repollo", "Zanahoria"]; ==> Copiado en "copiaArray"
```

Acceso a elementos de un array
Los índices de los arrays de JavaScript comienzan en cero, es decir, el índice del primer elemento de un array es 0, y el del último elemento es igual al valor de la propiedad length del array restándole 1.

Si se utiliza un número de índice no válido, se obtendrá undefined.

```
let arr = ['este es el primer elemento', 'este es el segundo elemento', 'este es el último elemento']
console.log(arr[0])              // escribe en consola 'este es el primer elemento'
console.log(arr[1])              // escribe en consola 'este es el segundo elemento'
console.log(arr[arr.length - 1]) // escribe en consola 'este es el último elemento'
```

Los elementos de un array pueden considerarse propiedades del objeto tanto como toString (sin embargo, para ser precisos, toString() es un método). Sin embargo, se obtendrá un error de sintaxis si se intenta acceder a un elemento de un array de la forma siguiente, ya que el nombre de la propiedad no sería válido:

```
console.log(arr.0) // error de sintaxis
```

No hay nada especial ni en los arrays de JavaScript ni en sus propiedades que ocasione esto. En JavaScript, las propiedades cuyo nombre comienza con un dígito no pueden referenciarse con la notación punto y debe accederse a ellas mediante la notación corchete.

Por ejemplo, dado un objeto con una propiedad de nombre '3d', sólo podría accederse a dicha propiedad con la notación corchete.

```
let decadas = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
console.log(decadas.0)  // error de sintaxis
console.log(decadas[0]) // funciona correctamente
renderizador.3d.usarTextura(modelo, 'personaje.png')
renderizador['3d'].usarTextura(modelo, 'personaje.png')
```

En el último ejemplo, ha sido necesario poner '3d' entre comillas. Es posible usar también comillas con los índices del los arrays de JavaScript (p. ej., decadas['2'] en vez de decadas[2]), aunque no es necesario.

El motor de JavaScript transforma en un string el 2 de decadas[2] a través de una conversión implícita mediante toString. Por tanto, '2' y '02' harían referencia a dos posiciones diferentes en el objeto decadas, y el siguiente ejemplo podría dar true como resultado:

```
console.log(decadas['2'] != decadas['02'])
```

Relación entre length y las propiedades numéricas

La propiedad length de un array de JavaScript está conectada con algunas otras de sus propiedades numéricas.

Varios de los métodos propios de un array (p. ej., join(), slice(), indexOf(), etc.) tienen en cuenta el valor de la propiedad length de un array cuando se les llama.

Otros métodos (p. ej., push(), splice(), etc.) modifican la propiedad length de un array.

```
const frutas = []
frutas.push('banana', 'manzana', 'pera')

console.log(frutas.length) // 3
```

Cuando se le da a una propiedad de un array JavaScript un valor que corresponda a un índice válido para el array pero que se encuentre fuera de sus límites, el motor actualizará el valor de la propiedad length como corresponda:

```
frutas[5] = 'pera'
console.log(frutas[5])          // 'pera'
console.log(Object.keys(frutas)) // ['0', '1', '2', '5']
console.log(frutas.length)       // 6
```

Si se aumenta el valor de length:

```
frutas.length = 10
console.log(frutas)             // ['banana', 'manzana', 'pera', <2 empty items>, 'pera', <4 empty items>]
console.log(Object.keys(frutas)) // ['0', '1', '2', '5']
console.log(frutas.length)       // 10
console.log(frutas[8])          // undefined
```

Si se disminuye el valor de la propiedad length pueden eliminarse elementos:

```
frutas.length = 2
console.log(Object.keys(frutas)) // ['0', '1']
console.log(frutas.length)       // 2
```

Creación de un array a partir de una expresión regular

El resultado de una búsqueda con una RegExp en un string puede crear un array de JavaScript. Este array tendrá propiedades y elementos que proporcionan información sobre la correspondencia encontrada. Para obtener un array de esta forma puede utilizarse RegExp.exec(), String.match() o String.replace().

El siguiente ejemplo, y la tabla que le sigue, pueden ayudar a comprender mejor las propiedades y elementos a los que nos referimos:

```
// Buscar una d seguida de una o más b y, al final, de otra d
// Recordar las b y la d final
// No distinguir mayúsculas y minúsculas

const miRe = /d(b+)(d)/i
```

Every

El método every ejecuta la función callback dada una vez por cada elemento presente en el arreglo hasta encontrar uno que haga retornar un valor falso a callback (un valor que resulte falso cuando se convierta a booleano). Si no se encuentra tal elemento, el método every de inmediato retorna false. O si callback retorna verdadero para todos los elementos, every retornará true. callback es llamada sólo para índices del arreglo que tengan valores asignados; no se llama para índices que hayan sido eliminados o a los que no se les haya asignado un valor.

callback es llamada con tres argumentos: el valor del elemento, el índice del elemento y el objeto Array que está siendo recorrido.

Si se proporciona un parámetro thisArg a every, será pasado a la función callback cuando sea llamada, usándolo como valor this. En otro caso, se pasará el valor undefined para que sea usado

como valor this. El valor this observable por parte de callback se determina de acuerdo a las normas usuales para determinar el this visto por una función.

every no modifica el arreglo sobre el cual es llamado.

El intervalo de elementos procesados por every se establece antes de la primera llamada a callback. Los elementos que se agreguen al arreglo después de que la función every comience no serán vistos por la función callback. Si se modifican elementos existentes en el arreglo, su valor cuando sea pasado a callback será el valor que tengan cuando sean visitados; los elementos que se eliminen no serán visitados.

every opera como el cuantificador "para todo" en matemáticas. En particular con el arreglo vacío retorna true. (es un true que todos los elementos del conjunto vacío satisfacen una condición dada.)

Ejemplos
Probando el tamaño de todos los elementos de un arreglo

El siguiente ejemplo prueba si todos los elementos de un arreglo son mayores que 10.

```
function esGrande(elemento, indice, arrreglo) {
  return elemento >= 10;
}
[12, 5, 8, 130, 44].every(esGrande);   // false
[12, 54, 18, 130, 44].every(esGrande); // true
```

Usar funciones flecha
Las funciones flecha proveen una sintaxis más corta para la misma prueba.

```
[12, 5, 8, 130, 44].every(elem => elem >= 10); // false
[12, 54, 18, 130, 44].every(elem => elem >= 10); // true
```

Filters

filter() llama a la función callback sobre cada elemento del array, y construye un nuevo array con todos los valores para los cuales callback devuelve un valor verdadero. callback es invocada sólo para índices del array que tengan un valor asignado. No se invoca sobre índices que hayan sido borrados o a los que no se les haya asignado algún valor. Los elementos del array que no cumplan la condición callback simplemente los salta, y no son incluidos en el nuevo array.

callback se invoca con tres argumentos:

El valor de cada elemento
El índice del elemento
El objeto Array que se está recorriendo

Si se proporciona un parámetro thisArg a filter(), este será pasado a callback cuando sea invocado, para usarlo como valor this. De lo contrario, se pasará el valor undefined como valor this. El valor this dentro del callback se determina conforme a las las normas habituales para determinar el this visto por una función.

filter() no modifica el array sobre el cual es llamado.

El rango de elementos procesados por filter() se establece antes de la primera invocación de callback. Los elementos que se añadan al array después de que comience la llamada a filter() no serán visitados por callback. Si se modifica o elimina un elemento existente del array, cuando pase su valor a callback será el que tenga cuando filter() lo recorra; los elementos que son eliminados no son recorridos.

Ejemplos
Filtrando todos los valores pequeños
El siguiente ejemplo usa filter() para crear un array filtrado que excluye todos los elementos con valores inferiores a 10.

```
function esGrande(elemento) {
  return elemento >= 10;
}
var filtrados = [12, 5, 8, 130, 44].filter(esGrande);
// filtrados es [12, 130, 44]
```

Some()

some() ejecuta la función callback una vez por cada elemento presente en el array hasta que encuentre uno donde callback retorna un valor verdadero (true). Si se encuentra dicho elemento, some() retorna true inmediatamente. Si no, some() retorna false. callback es invocada sólo para los índices del array que tienen valores asignados; no es invocada para índices que han sido borrados o a los que nunca se les han asignado valores.

callback es invocada con tres argumentos: el valor del elemento, el índice del elemento, y el objeto array sobre el que se itera.

Si se indica un parámetro thisArg a some(), se pasará a callback cuando es invocada, para usar como valor this. Si no, el valor undefined será pasado para usar como valor this. El valor this value observable por callback se determina de acuerdo a las reglas habituales para determinar el this visible por una función.

some() no modifica el array con el cual fue llamada.

El rango de elementos procesados por some() es configurado antes de la primera invocación de callback. Los elementos anexados al array luego de que comience la llamada a some() no serán visitados por callback. Si un elemento existente y no visitado del array es alterado por callback, su valor pasado al callback será el valor al momento que some() visita el índice del elemento; los elementos borrados no son visitados.

Ejemplos
Verificando el valor de los elementos de un array

El siguiente ejemplo verifica si algún elemento del array es mayor a 10.

```
function masquediez(element, index, array) {
  return element > 10;
}
[2, 5, 8, 1, 4].some(masquediez);  // false
[12, 5, 8, 1, 4].some(masquediez);
```

Búsqueda, ordenación y comparación

Búsqueda y comprobación

Existen varios métodos para realizar ciertas comprobaciones con arrays:

Método	Descripción
Array.isArray(obj)	Comprueba si obj es un array. Devuelve true o false.
.includes(obj, from)	Comprueba si obj es uno de los elementos incluidos en el array.
.indexOf(obj, from)	Devuelve la posición de la primera aparición de obj desde from.
.lastIndexOf(obj, from)	Devuelve la posición de la última aparición de obj desde from.

El primero de ellos, Array.isArray(obj) se utiliza para comprobar si obj es un array o no, devolviendo un booleano. Los otros tres métodos funcionan exactamente igual que sus equivalentes en los . El método includes() comprueba si el elemento obj pasado por parámetro es uno de los elementos que incluye el array, partiendo desde la posición from. Si se omite from, se parte desde 0.

```
const array = [5, 10, 15, 20, 25];


Array.isArray(array); // true

array.includes(10); // true

array.includes(10, 2); // false

array.indexOf(25); // 4

array.lastIndexOf(10, 0); // -1
```

Por otro lado, tenemos indexOf() y lastIndexOf() dos funciones que se utilizan para devolver la posición del elemento obj pasado por parámetro, empezando a buscar en la posición from (o 0 si se omite). El primer método, devuelve la primera aparición, mientras que el segundo método devuelve la última aparición.

Modificación de arrays

Es posible que tengamos un array específico al que queremos hacer ciertas modificaciones donde slice() y splice() se quedan cortos (o resulta más cómodo utilizar los siguientes métodos). Existen algunos métodos introducidos en ECMAScript 6 que nos permiten crear una versión modificada de un array, mediante métodos como copyWithin() o fill():

Método	Descripción
.copyWithin(pos, ini, end)	Devuelve array, copiando en pos los ítems desde ini a end.
.fill(obj, ini, end)	Devuelve un array relleno de obj desde ini hasta end.

El primero de ellos, copyWithin(pos, ini, end) nos permite crear una copia del array que alteraremos de la siguiente forma: en la posición pos copiaremos los elementos del propio array que aparecen desde la posición ini hasta la posición end. Es decir, desde la posición 0 hasta pos será exactamente igual, y de ahí en adelante, será una copia de los valores de la posición ini a la posición end. Veamos algunos ejemplos:

```
const array = ["a", "b", "c", "d", "e", "f"];


// Estos métodos modifican el array original

array.copyWithin(5, 0, 1); // Devuelve ['a', 'b', 'c', 'd', 'e', 'a']

array.copyWithin(3, 0, 3); // Devuelve ['a', 'b', 'c', 'a', 'b', 'c']

array.fill("Z", 0, 5); // Devuelve ['Z', 'Z', 'Z', 'Z', 'Z', 'c']
```

Por otro lado, el método fill(obj, ini, end) es mucho más sencillo. Se encarga de devolver una versión del array, rellenando con el elemento obj desde la posición ini hasta la posición end.

Ordenaciones

En Javascript, es muy habitual que tengamos arrays y queramos ordenar su contenido por diferentes criterios. En este apartado, vamos a ver los métodos reverse() y sort(), utiles para ordenar un array:

Método	Descripción
.reverse()	Invierte el orden de elementos del array.
.sort()	Ordena los elementos del array bajo un criterio de ordenación alfabética.
.sort(func)	Ordena los elementos del array bajo un criterio de ordenación func.

En primer lugar, el método reverse() cambia los elementos del array en orden inverso, es decir, si tenemos [5, 4, 3] lo modifica de modo que ahora tenemos [3, 4, 5]. Por otro lado, el método sort() realiza una ordenación (por orden alfabético) de los elementos del array:

```
const array = ["Alberto", "Ana", "Mauricio", "Bernardo", "Zoe"];


// Ojo, cada línea está modificando el array original

array.sort(); // ['Alberto', 'Ana', 'Bernardo', 'Mauricio', 'Zoe']

array.reverse(); // ['Zoe', 'Mauricio', 'Bernardo', 'Ana', 'Alberto']
```

Un detalle muy importante es que estos dos métodos modifican el array original, además de devolver el array modificado. Si no quieres que el array original cambie, asegurate de crear primero una copia del array, para así realizar la ordenación sobre esa copia y no sobre el original.

Sin embargo, la ordenación anterior se realizó sobre String y todo fue bien. Veamos que ocurre si intentamos ordenar un array de números:

```
const array = [1, 8, 2, 32, 9, 7, 4];


array.sort(); // Devuelve [1, 2, 32, 4, 7, 8, 9], que NO es el resultado deseado
```

Esto ocurre porque, al igual que en el ejemplo anterior, el tipo de ordenación que realiza sort() por defecto es una ordenación alfabética, mientras que en esta ocasión buscamos una ordenación natural, que es la que se suele utilizar con números. Esto se puede hacer en Javascript, pero requiere pasarle por parámetro al sort() lo que se llama una función de comparación.

Función de comparación

Como hemos visto, la ordenación que realiza sort() por defecto es siempre una ordenación alfabética. Sin embargo, podemos pasarle por parámetro lo que se conoce con los nombres de función de ordenación o función de comparación. Dicha función, lo que hace es establecer otro criterio de ordenación, en lugar del que tiene por defecto:

```
const array = [1, 8, 2, 32, 9, 7, 4];


// Función de comparación para ordenación natural

const fc = function (a, b) {

return a > b;

};


array.sort(fc); // Devuelve [1, 2, 4, 7, 8, 9, 32], que SÍ es el resultado deseado
```

Como se puede ver en el ejemplo anterior, creando la función de ordenación fc y pasándola por parámetro a sort(), le indicamos como debe hacer la ordenación y ahora si la realiza correctamente.

Si profundizamos en la tarea que realiza el sort(), lo que hace concretamente es analizar pares de elementos del array en cuestión. El primer elemento es a y el segundo elemento es b. Por lo tanto, al pasarle la función de comparación fc, dicha función se encargará de, si devuelve true cambia el orden de a y b, si devuelve false los mantiene igual. Esto es lo que se conoce como el método de la burbuja, uno de los sistemas de ordenación más sencillos.

Obviamente, el usuario puede crear sus propias funciones de comparación con criterios específicos y personalizados, no sólo el que se muestra como ejemplo.

Arrays functions

¿Qué son las Array functions?

Básicamente, son métodos que tiene cualquier variable que sea de tipo array , y que permite realizar una operación con todos los elementos de dicho array para conseguir un objetivo concreto, dependiendo del método. En general, a dichos métodos se les pasa por parámetro una función callback y unos parámetros opcionales.

Estas son las Array functions que podemos encontrarnos en Javascript:

Método	Descripción
.forEach(cb, arg)	Realiza la operación definida en cb por cada elemento del array.
.every(cb, arg)	Comprueba si todos los elementos del array cumplen la condición de cb.
.some(cb, arg)	Comprueba si al menos un elem. del array cumple la condición de cb.
.map(cb, arg)	Construye un array con lo que devuelve cb por cada elemento del array.
.filter(cb, arg)	Construye un array con los elementos que cumplen el filtro de cb.
.findIndex(cb, arg)	Devuelve la posición del elemento que cumple la condición de cb.
.find(cb, arg)	Devuelve el elemento que cumple la condición de cb.
.reduce(cb, arg)	Ejecuta cb con cada elemento (de izq a der), acumulando el resultado.
.reduceRight(cb, arg)	Idem al anterior, pero en orden de derecha a izquierda.

A grandes rasgos, a cada uno de estos métodos se les pasa una función callback que se ejecutará por cada uno de los elementos que contiene el array. Empecemos por forEach(), que es quizás el más sencillo de todos.

forEach (Cada uno)

Como se puede ver, el método forEach() no devuelve nada y espera que se le pase por parámetro una función que se ejecutará por cada elemento del array. Esa función, puede ser pasada en cualquiera de los formatos que hemos visto: como función tradicional o como función flecha:

```
const arr = ["a", "b", "c", "d"];


// Con funciones por expresión

const f = function () {

console.log("Un elemento.");

};

arr.forEach(f);


// Con funciones anónimas

arr.forEach(function () {

console.log("Un elemento.");

});


// Con funciones flecha

arr.forEach(() => console.log("Un elemento."));
```

Sin embargo, este ejemplo no tiene demasiada utilidad. A la funcion callback se le pueden pasar varios parámetros opcionales:

Si se le pasa un primer parámetro, este será el elemento del array.

Si se le pasa un segundo parámetro, este será la posición en el array.

Si se le pasa un tercer parámetro, este será el array en cuestión.

Veamos un ejemplo:

```
const arr = ["a", "b", "c", "d"];


arr.forEach((e) => console.log(e)); // Devuelve 'a' / 'b' / 'c' / 'd'

arr.forEach((e, i) => console.log(e, i)); // Devuelve 'a' 0 / 'b' 1 / 'c' 2 / 'd' 3

arr.forEach((e, i, a) => console.log(a[0])); // Devuelve 'a' / 'a' / 'a' / 'a'
```

En este ejemplo, he nombrado e al parámetro que hará referencia al elemento, i al parámetro que hará referencia al índice (posición del array) y a al parámetro que hará referencia al array en cuestión. Aún así, el usuario puede ponerle a estos parámetros el nombre que prefiera. Como se puede ver, realmente forEach() es otra forma de hacer un bucle (sobre un array), sin tener que recurrir a bucles tradicionales como for o while.

Como vemos en la tabla anterior, al método forEach() se le puede pasar un segundo parámetro arg, que representa el valor que sobreescribiría a la palabra clave this en el código dentro de la función callback. De necesitar esta funcionalidad, recuerda que no puedes utilizar las funciones flecha, ya que el this no tiene efecto en ellas.

every (Todos)

El método every() permite comprobar si todos y cada uno de los elementos de un array cumplen la condición que se especifique en la funcion callback:

```
const arr = ["a", "b", "c", "d"];

arr.every((e) => e.length == 1); // true
```

En este caso, la magia está en el callback. La condición es que la longitud de cada elemento del array sea 1. Si dicha función devuelve true, significa que cumple la condición, si devuelve false, no la cumple. Por lo tanto, si todos los elementos del array devuelven true, entonces every() devolverá true.

Si expandimos el ejemplo anterior a un código más detallado, tendríamos el siguiente ejemplo equivalente, que quizás sea más comprensible para entenderlo:

```
const arr = ["a", "b", "c", "d"];
```

```
// Esta función se ejecuta por cada elemento del array

const todos = function (e) {

// Si el tamaño del string es igual a 1

if (e.length == 1) return true;

else return false;

};


arr.every(todos); // Le pasamos la función callback todos() a every
```

some (Al menos uno)

De la misma forma que el método anterior sirve para comprobar si todos los elementos del array cumplen una determinada condición, con some() podemos comprobar si al menos uno de los elementos del array, cumplen dicha condición definida por el callback.

```
const arr = ["a", "bb", "c", "d"];

arr.some((e) => e.length == 2); // true
```

Observa que en este ejemplo, el método some() devuelve true porque existe al menos un elemento del array con una longitud de 2 caracteres.

map (Transformaciones)

El método map() es un método muy potente y útil para trabajar con arrays, puesto que su objetivo es devolver un nuevo array donde cada uno de sus elementos será lo que devuelva la función callback por cada uno de los elementos del array original:

```
const arr = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];

const nuevoArr = arr.map((e) => e.length);

nuevoArr; // Devuelve [3, 5, 5, 9, 9]
```

Observa que el array devuelto por map() es nuevoArr, y cada uno de los elementos que lo componente, es el número devuelto por el callback (e.length), que no es otra cosa sino el tamaño de cada .

Este método nos permite hacer multitud de operaciones, ya que donde devolvemos e.length podriamos devolver el propio array modificado o cualquier otra cosa.

filter (Filtrado)

El método filter() nos permite filtrar los elementos de un array y devolver un nuevo array con sólo los elementos que queramos. Para ello, utilizaremos la función callback para establecer una condición que devuelve true sólo en los elementos que nos interesen:

```
const arr = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];

const nuevoArr = arr.filter((e) => e[0] == "P");


nuevoArr; // Devuelve ['Pablo', 'Pedro', 'Pancracio']
```

En este ejemplo, filtramos sólo los elementos en los que su primera letra sea P. Por lo tanto, la variable nuevoArr será un array con sólo esos elementos.

Ten en cuenta que si ningún elemento cumple la condición, filter() devuelve un vacío.

find (Búsqueda)

En ECMAScript 6 se introducen dos nuevos métodos dentro de las Array functions: find() y findIndex(). Ambos se utilizan para buscar elementos de un array mediante una condición, la diferencia es que el primero devuelve el elemento mientras que el segundo devuelve su posición en el array original. Veamos cómo funcionan:

```
const arr = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];

arr.find((e) => e.length == 5); // 'Pablo'

arr.findIndex((e) => e.length == 5); // 1
```

La condición que hemos utilizado en este ejemplo es buscar el elemento que tiene 5 caracteres de longitud. Al buscarlo en el array original, el primero que encontramos

es Pablo, puesto que find() devolverá 'Pablo' y findIndex() devolverá 1, que es la segunda posición del array donde se encuentra.

En el caso de no encontrar ningún elemento que cumpla la condición, find() devolverá , mientras que findIndex(), que debe devolver un , devolverá -1.

reduce (Acumuladores)

Por último, nos encontramos con una pareja de métodos denominados reduce() y reduceRight(). Ambos métodos se encargan de recorrer todos los elementos del array, e ir acumulando sus valores (o alguna operación diferente) y sumarlo todo, para devolver su resultado final.

En este par de métodos, encontraremos una primera diferencia en su función callback, puesto que en lugar de tener los clásicos parámetros opcionales (e, i, a) que hemos utilizado hasta ahora, tiene (p, e, i, a), donde vemos que aparece un primer parámetro extra inicial: p.

En la primera iteración, p contiene el valor del primer elemento del array y e del segundo. En siguientes iteraciones, p es el acumulador que contiene lo que devolvió el callback en la iteración anterior, mientras que e es el siguiente elemento del array, y así sucesivamente. Veamos un ejemplo para entenderlo:

```
const arr = [95, 5, 25, 10, 25];

arr.reduce((p, e) => {

console.log(`P=${p} e=${e}`);

return p + e;

});



// P=95 e=5 (1ª iteración: elemento 1: 95 + elemento 2: 5) = 100

// P=100 e=25 (2ª iteración: 100 + elemento 3: 25) = 125

// P=125 e=10 (3ª iteración: 125 + elemento 4: 10) = 135

// P=135 e=25 (4ª iteración: 135 + elemento 5: 25) = 160

// Finalmente, devuelve 160
```

Gracias a esto, podemos utilizar el método reduce() como acumulador de elementos de izquierda a derecha y reduceRight() como acumulador de elementos de derecha a izquierda. Veamos un ejemplo de cada uno, realizando una resta en lugar de una suma:

```
const arr = [95, 5, 25, 10, 25];

arr.reduce((p, e) => p - e); // 95 - 5 - 25 - 10 - 25. Devuelve 30

arr.reduceRight((p, e) => p - e); // 25 - 10 - 25 - 5 - 95. Devuelve -110
```

Recuerda que en cualquiera de estas array functions puedes realizar operaciones o condiciones tanto con el parámetro e (elemento), como con el parámetro i (índice o posición) o con el parámetro a (array).

Map

map llama a la función callback provista una vez por elemento de un array, en orden, y construye un nuevo array con los resultados. callback se invoca sólo para los índices del array que tienen valores asignados; no se invoca en los índices que han sido borrados o a los que no se ha asignado valor.

callback es llamada con tres argumentos: el valor del elemento, el índice del elemento, y el objeto array que se está recorriendo.

Si se indica un parámetro thisArg a un map, se usará como valor de this en la función callback. En otro caso, se pasará undefined como su valor this. El valor de this observable por el callback se determina de acuerdo a las reglas habituales para determinar el valor this visto por una función.

map no modifica el array original en el que es llamado (aunque callback, si es llamada, puede modificarlo).

El rango de elementos procesado por map es establecido antes de la primera invocación del callback. Los elementos que sean agregados al array después de que la llamada a map comience no serán visitados por el callback. Si los elementos existentes del array son modificados o eliminados, su valor pasado al callback será el valor en el momento que el map lo visita; los elementos que son eliminados no son visitados.

Ejemplos

Procesar un array de números aplicándoles la raíz cuadrada

El siguiente código itera sobre un array de números, aplicándoles la raíz cuadrada a cada uno de sus elementos, produciendo un nuevo array a partir del inicial.

```
var numeros= [1, 4, 9];
var raices = numeros.map(Math.sqrt);
// raices tiene [1, 2, 3]
// numeros aún mantiene [1, 4, 9]
```

Usando map para dar un nuevo formato a los objetos de un array

El siguiente código toma un array de objetos y crea un nuevo array que contiene los nuevos objetos formateados.

```
var kvArray = [{clave:1, valor:10},
               {clave:2, valor:20},
               {clave:3, valor: 30}];

var reformattedArray = kvArray.map(function(obj){
   var rObj = {};
   rObj[obj.clave] = obj.valor;
   return rObj;
});

// reformattedArray es ahora [{1:10}, {2:20}, {3:30}],

// kvArray sigue siendo:
// [{clave:1, valor:10},
//  {clave:2, valor:20},
```

// {clave:3, valor: 30}]

Mapear un array de números usando una función con un argumento

El siguiente código muestra cómo trabaja map cuando se utiliza una función que requiere de un argumento. El argumento será asignado automáticamente a cada elemento del arreglo conforme map itera el arreglo original.

```
var numeros = [1, 4, 9];
var dobles  = numeros.map(function(num) {
  return num * 2;
});

// dobles es ahora [2, 8, 18]
// números sigue siendo [1, 4, 9]
```

Usando map de forma genérica

Este ejemplo muestra como usar map en String para obtener un arreglo de bytes en codificación ASCII representando el valor de los caracteres:

```
var map = Array.prototype.map;
var valores = map.call('Hello World', function(char) { return char.charCodeAt(0); });
// valores ahora tiene [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
```

Reduce

El método reduce() ejecuta callback una vez por cada elemento presente en el array, excluyendo los huecos del mismo, recibe cuatro argumentos:

valorAnterior
valorActual
indiceActual
array

La primera vez que se llama la función, valorAnterior y valorActual pueden tener uno de dos valores. Si se indicó un valorInicial al llamar a reduce, entonces valorAnterior será igual al valorInicial y valorActual será igual al primer elemento del array. Si no se indicó un valorInicial, entonces valorAnterior será igual al primer valor en el array y valorActual será el segundo.

Si el array está vacío y no se indicó un valorInicial lanzará un TypeError. Si el array tiene un sólo elemento (sin importar la posición) y no se indicó un valorInicial, o si se indicó un valorInicial pero el arreglo está vacío, se retornará ese único valor sin llamar a la función.

Supongamos que ocurre el siguiente uso de reduce:

```
[0,1,2,3,4].reduce(function(valorAnterior, valorActual, indice, vector){
  return valorAnterior + valorActual;
});

// Primera llamada
valorAnterior = 0, valorActual = 1, indice = 1

// Segunda llamada
valorAnterior = 1, valorActual = 2, indice = 2

// Tercera llamada
valorAnterior = 3, valorActual = 3, indice = 3

// Cuarta llamada
valorAnterior = 6, valorActual = 4, indice = 4

// el array sobre el que se llama a reduce siempre es el objeto [0,1,2,3,4]

// Valor Devuelto: 10
```

Y si proporcionás un valorInicial, el resultado sería como este:

```
[0,1,2,3,4].reduce(function(valorAnterior, valorActual, indice, vector){
  return valorAnterior + valorActual;
}, 10);

// Primera llamada
valorAnterior = 10, valorActual = 0, indice = 0

// Segunda llamada
valorAnterior = 10, valorActual = 1, indice = 1
```

// Tercera llamada
valorAnterior = 11, valorActual = 2, indice = 2

// Cuarta llamada
valorAnterior = 13, valorActual = 3, indice = 3

// Quinta llamada
valorAnterior = 16, valorActual = 4, indice = 4

// el array sobre el que se llama a reduce siempre es el objeto [0,1,2,3,4]

// Valor Devuelto: 20

Iteradores

En ECMAScript 6 se introducen unos métodos muy útiles para utilizar como iteradores (objetos preparados para recorrer los elementos de un array y devolver información). Hablamos de los métodos keys(), values() y entries(). El primero de ellos permite avanzar en un array, mientras va devolviendo las posiciones, el segundo los valores (el elemento en sí) y el tercero devuelve un array con la posición en el primer elemento y el valor en el segundo elemento.

Método	Descripción
i .keys()	Permite iterar un array e ir devolviendo sus índices o posiciones (keys).
i .values()	Permite iterar un array e ir devolviendo sus valores (elementos).
i .entries()	Permite iterar un array e ir devolviendo un array [índice, valor].

Estos métodos, combinados con un for...of por ejemplo, permiten recorrer los arrays y obtener diferente información del array rápidamente. En el siguiente ejemplo utilizamos una característica avanzada que veremos más adelante llamada desestructuración:

```
const arr = ["Sonic", "Mario", "Luigi"];

// Obtiene un array con las keys (posiciones)

const keys = [...arr.keys()]; // [0, 1, 2]

// Obtiene un array con los valores (elementos)

const values = [...arr.values()]; // ['Sonic', 'Mario', 'Luigi']

// Obtiene un array con las entradas (par key, valor)

const entries = [...arr.entries()]; // [[0, 'Sonic'], [1, 'Mario'], [2, 'Luigi']]
```

Objeto String

El objeto String se utiliza para representar y manipular una secuencia de caracteres.

Descripción

Las cadenas son útiles para almacenar datos que se pueden representar en forma de texto. Algunas de las operaciones más utilizadas en cadenas son verificar su length, para construirlas y concatenarlas usando operadores de cadena + y +=, verificando la existencia o ubicación de subcadenas con indexOf() o extraer subcadenas con el método substring().

Crear cadenas

Las cadenas se pueden crear como primitivas, a partir de cadena literales o como objetos, usando el constructor String():

```
const string1 = "Una cadena primitiva";
const string2 = 'También una cadena primitiva';
const string3 = `Otra cadena primitiva más`;
const string4 = new String("Un objeto String");
```

Las strings primitivas y los objetos string se pueden usar indistintamente en la mayoría de las situaciones.

Las cadenas de literales se pueden especificar usando comillas simples o dobles, que se tratan de manera idéntica, o usando el carácter de comilla invertida `. Esta última forma especifica una Plantilla literal: con esta forma puedes interpolar expresiones.

Acceder a un caracter

Hay dos formas de acceder a un caracter individual en una cadena. La primera es con el método charAt():

```
return 'cat'.charAt(1) // devuelve "a"
```

La otra forma (introducida en ECMAScript 5) es tratar a la cadena como un objeto similar a un arreglo, donde los caracteres individuales corresponden a un índice numérico:

```
return 'cat'[1] // devuelve "a"
```

Cuando se usa la notación entre corchetes para acceder a los caracteres, no se puede intentar eliminar o asignar un valor a estas propiedades. Las propiedades involucradas no se pueden escribir ni configurar.

Comparar cadenas

En C, se usa la función strcmp() para comparar cadenas. En JavaScript, solo usás los operadores menor que y mayor que:

```
let a = 'a'
let b = 'b'
if (a < b) { // true
  console.log(a + ' es menor que ' + b)
} else if (a > b) {
  console.log(a + ' es mayor que ' + b)
```

```
} else {
  console.log(a + ' y ' + b + ' son iguales.')
}
```

Podés lograr un resultado similar usando el método localeCompare() heredado por las instancias de String.

Ten en cuenta que a == b compara las cadenas en a y b por ser igual en la forma habitual que distingue entre mayúsculas y minúsculas. Si deseas comparar sin tener en cuenta los caracteres en mayúsculas o minúsculas, usa una función similar a esta:

```
function isEqual(str1, str2)
{
    return str1.toUpperCase() === str2.toUpperCase()
} // isEqual
```

En esta función se utilizan mayúsculas en lugar de minúsculas, debido a problemas con ciertas conversiones de caracteres UTF-8.

Primitivas String y objetos String
Tené en cuenta que JavaScript distingue entre objetos String y valores de primitivas string. (Lo mismo ocurre con Booleanos y Números).

Las cadenas literales (denotadas por comillas simples o dobles) y cadenas devueltas de llamadas a String en un contexto que no es de constructor (es decir, llamado sin usar la palabra clave new) son cadenas primitivas. JavaScript automáticamente convierte las primitivas en objetos String, por lo que es posible utilizar métodos del objeto String en cadenas primitivas. En contextos donde se va a invocar a un método en una cadena primitiva o se produce una búsqueda de propiedad, JavaScript ajustará automáticamente la cadena primitiva y llamará al método o realizará la búsqueda de la propiedad.

```
let s_prim = 'foo'
let s_obj = new String(s_prim)

console.log(typeof s_prim) // Registra "string"
console.log(typeof s_obj)  // Registra "object"
```

Las primitivas de String y los objetos String también dan diferente resultado cuando se usa eval(). Las primitivas pasadas a eval se tratan como código fuente; Los objetos String se tratan como todos los demás objetos, devuelven el objeto. Por ejemplo:

```
let s1 = '2 + 2'             // crea una string primitiva
let s2 = new String('2 + 2')  // crea un objeto String
console.log(eval(s1))        // devuelve el número 4
console.log(eval(s2))        // devuelve la cadena "2 + 2"
```

Por estas razones, el código se puede romper cuando encuentra objetos String y espera una string primitiva en su lugar, aunque generalmente los programadores no necesitan preocuparse por la distinción.

Un objeto String siempre se puede convertir a su contraparte primitiva con el método valueOf().

```
console.log(eval(s2.valueOf()))  // devuelve el número 4
```

Notación de escape

Los caracteres especiales se pueden codificar mediante notación de escape:

Código	Salida
\XXX	(donde XXX es de 1 a 3 dígitos octales; rango de 0-377) Punto de código Unicode/carácter ISO-8859-1 entre U+0000 y U+00FF
\'	Comilla sencilla
\"	Comilla doble
\\	Barra inversa
\n	Nueva línea
\r	Retorno de carro
\v	Tabulación vertical
\t	Tabulación
\b	Retroceso
\f	Avance de página
\uXXXX	(donde XXXX son 4 dígitos hexadecimales; rango de 0x0000-0xFFFF) Unidad de código UTF-16/punto de código Unicode entre U+0000 y U+FFFF
\u{X} ... \u{XXXXXX}	(donde X…XXXXXX es de 1 a 6 dígitos hexadecimales; rango de 0x0-0x10FFFF) Unidad de código UTF-32/punto de código Unicode entre U+0000 y U+10FFFF
\xXX	(donde XX son 2 dígitos hexadecimales; rango de 0x00-0xFF) Punto de código Unicode/carácter ISO-8859-1 entre U+0000 y U+00FFCadenas literales largas

A veces, tu código incluirá cadenas que son muy largas. En lugar de tener líneas que se prolongan interminablemente o que se ajustan según el editor, es posible que desees dividir específicamente la cadena en varias líneas en el código fuente sin afectar el contenido real de la cadena. hay dos maneras de conseguirlo.

Método 1

Puedes usar el operador + para agregar varias cadenas juntas, así:

```
let longString = "Esta es una cadena muy larga que necesita " +
        "que dividimos en varias líneas porque " +
        "de lo contrario, mi código es ilegible."
```

Método 2

Puedes usar el caracter de barra invertida (\) al final de cada línea para indicar que la cadena continúa en la siguiente línea. Asegurate de que no haya ningún espacio ni ningún otro carácter después de la barra invertida (a excepción de un salto de línea) o como sangría; de lo contrario, no funcionará:

```
let longString = "Esta es una cadena muy larga que necesita \
que dividimos en varias líneas porque \
de lo contrario, mi código es ilegible."
```

Ambos métodos anteriores dan como resultado cadenas idénticas.

Constructor String()
Crea un nuevo objeto String. Realiza la conversión de tipos cuando se llama como función, en lugar de como constructor, lo cual suele ser más útil.
Métodos estáticos
String.fromCharCode(num1 [, ...[, numN]])
Devuelve una cadena creada utilizando la secuencia de valores Unicode especificada.
String.fromCodePoint(num1 [, ...[, numN]])
Devuelve una cadena creada utilizando la secuencia de puntos de código especificada.
String.raw()
Devuelve una cadena creada a partir de una plantilla literal sin formato.
Propiedades de la instancia
String.prototype.length
Refleja la length de la cadena. Solo lectura.
Métodos de instancia
String.prototype.charAt(index)
Devuelve el caracter (exactamente una unidad de código UTF-16) en el index especificado.
String.prototype.charCodeAt(index)
Devuelve un número que es el valor de la unidad de código UTF-16 en el index dado.
String.prototype.codePointAt(pos)
Devuelve un número entero no negativo que es el valor del punto de código del punto de código codificado en UTF-16 que comienza en la pos especificada.
String.prototype.concat(str[, ...strN])
Combina el texto de dos (o más) cadenas y devuelve una nueva cadena.
String.prototype.includes(searchString [, position])
Determina si la cadena de la llamada contiene searchString.
String.prototype.endsWith(searchString[, length])
Determina si una cadena termina con los caracteres de la cadena searchString.
String.prototype.indexOf(searchValue[, fromIndex])
Devuelve el índice dentro del objeto String llamador de la primera aparición de searchValue, o -1 si no lo encontró.
String.prototype.lastIndexOf(searchValue[, fromIndex])
Devuelve el índice dentro del objeto String llamador de la última aparición de searchValue, o -1 si no lo encontró.
String.prototype.localeCompare(compareString[, locales[, options]])
Devuelve un número que indica si la cadena de referencia compareString viene antes, después o es equivalente a la cadena dada en el orden de clasificación.
String.prototype.match(regexp)
Se utiliza para hacer coincidir la expresión regular regexp con una cadena.
String.prototype.matchAll(regexp)
Devuelve un iterador de todas las coincidencias de regexp.
String.prototype.normalize([form])
Devuelve la forma de normalización Unicode del valor de la cadena llamada.
String.prototype.padEnd(targetLength[, padString])
Rellena la cadena actual desde el final con una cadena dada y devuelve una nueva cadena de longitud targetLength.
String.prototype.padStart(targetLength[, padString])
Rellena la cadena actual desde el principio con una determinada cadena y devuelve una nueva cadena de longitud targetLength.
String.prototype.repeat(count)
Devuelve una cadena que consta de los elementos del objeto repetidos count veces.
String.prototype.replace(searchFor, replaceWith)

Se usa para reemplazar ocurrencias de searchFor usando replaceWith. searchFor puede ser una cadena o expresión regular, y replaceWith puede ser una cadena o función.
String.prototype.replaceAll(searchFor, replaceWith)
Se utiliza para reemplazar todas las apariciones de searchFor usando replaceWith. searchFor puede ser una cadena o expresión regular, y replaceWith puede ser una cadena o función.
String.prototype.search(regexp)
Busca una coincidencia entre una expresión regular regexp y la cadena llamadora.
String.prototype.slice(beginIndex[, endIndex])
Extrae una sección de una cadena y devuelve una nueva cadena.
String.prototype.split([sep[, limit]])
Devuelve un arreglo de cadenas pobladas al dividir la cadena llamadora en las ocurrencias de la subcadena sep.
String.prototype.startsWith(searchString[, length])
Determina si la cadena llamadora comienza con los caracteres de la cadena searchString.
String.prototype.substr()
Devuelve los caracteres en una cadena que comienza en la ubicación especificada hasta el número especificado de caracteres.
String.prototype.substring(indexStart[, indexEnd])
Devuelve una nueva cadena que contiene caracteres de la cadena llamadora de (o entre) el índice (o indeces) especificados.
String.prototype.toLocaleLowerCase([locale, ...locales])
Los caracteres dentro de una cadena se convierten a minúsculas respetando la configuración regional actual.

Para la mayoría de los idiomas, devolverá lo mismo que toLowerCase().

String.prototype.toLocaleUpperCase([locale, ...locales])
Los caracteres dentro de una cadena se convierten a mayúsculas respetando la configuración regional actual.

Para la mayoría de los idiomas, devolverá lo mismo que toUpperCase().

String.prototype.toLowerCase()
Devuelve el valor de la cadena llamadora convertido a minúsculas.
String.prototype.toString()
Devuelve una cadena que representa el objeto especificado. Redefine el método Object.prototype.toString().
String.prototype.toUpperCase()
Devuelve el valor de la cadena llamadora convertido a mayúsculas.
String.prototype.trim()
Recorta los espacios en blanco desde el principio y el final de la cadena. Parte del estándar ECMAScript 5.
String.prototype.trimStart()
Recorta los espacios en blanco desde el principio de la cadena.
String.prototype.trimEnd()
Recorta los espacios en blanco del final de la cadena.
String.prototype.valueOf()
Devuelve el valor primitivo del objeto especificado. Redefine el método Object.prototype.valueOf().
String.prototype.@@iterator()

Devuelve un nuevo objeto Iterator que itera sobre los puntos de código de un valor de cadena, devolviendo cada punto de código como un valor de cadena.

Ejemplos
Conversión de cadenas
Es posible usar String como una alternativa más confiable de toString(), ya que funciona cuando se usa en null, undefined y en símbolos. Por ejemplo:

```
let outputStrings = []
for (let i = 0, n = inputValues.length; i < n; ++i) {
  outputStrings.push(String(inputValues[i]));
}
```

Interpolación

Interpolación de variables

En ECMAScript se introduce una interesante mejora en la manipulación general de , sobre todo respecto a la legibilidad de código.

Hasta ahora, si queríamos concatenar el valor de algunas variables con textos predefinidos por nosotros, teníamos que hacer algo parecido a esto:

```
const sujeto = "frase";

const adjetivo = "concatenada";

"Una " + sujeto + " bien " + adjetivo; // 'Una frase bien concatenada'
```

A medida que añadimos más variables, el código se hace bastante menos claro y más complejo de leer, especialmente si tenemos que añadir arrays, introducir comillas simples que habría que escapar con \' o combinar comillas simples con dobles, etc...

Para evitarlo, se introducen las backticks (comillas hacia atrás), que nos permiten interpolar el valor de las variables sin tener que cerrar, concatenar y abrir la cadena de texto continuamente:

```
const sujeto = "frase";

const adjetivo = "concatenada";

`Una ${sujeto} mejor ${adjetivo}`; // 'Una frase mejor concatenada'
```

Esto es una funcionalidad muy simple, pero que mejora sustancialmente la calidad de código generado. Eso sí, recuerda que se introduce en ECMAScript 6, con todo lo que ello conlleva.

Plantilla de cadena de caracteres (Template String)

Resumen

Las plantillas de cadena de texto (template strings) son literales de texto que habilitan el uso de expresiones incrustadas. Es posible utilizar cadenas de texto de más de una línea, y funcionalidades de interpolación de cadenas de texto con ellas.

Sintaxis

`cadena de texto`

`línea 1 de la cadena de texto

línea 2 de la cadena de texto`

`cadena de texto ${expresión} texto`

tag `cadena de texto ${expresión} texto`

Descripción

Las plantillas de cadena de texto se delimitan con el caracter de comillas o tildes invertidas (` `) (grave accent) , en lugar de las comillas simples o dobles. Las plantillas de cadena de texto pueden contener marcadores, identificados por el signo de dólar, y envueltos en llaves (${expresión}). Las expresiones contenidas en los marcadores, junto con el texto entre ellas, son enviados como argumentos a una función. La función por defecto simplemente concatena las partes para formar una única cadena de texto. Si hay una expresión antes de la plantilla de cadena de texto (i.e. tag), llamamos a esta plantilla de cadena de texto "plantilla de cadena de texto con etiqueta". En este caso, la expresión de etiqueta (típicamente una función) es llamada a partir de la cadena resultante de procesar la plantilla de cadena de texto, que luego puede ser manipulada antes de ser devuelta. En caso de querer escapar una comilla o tilde invertida en una plantilla de cadena de texto, pon un backslash \ antes de la comilla o tilde invertida.

`\`` === '`' // --> verdadero

Cadenas de más de una línea

Los caracteres de fin de línea encontrados son parte de la plantilla de cadena de texto. En el caso de cadenas de texto normales, esta es la sintaxis necesaria para obtener cadenas de más de una línea:

//ES5

console.log("línea 1 de cadena de texto\n\

```
línea 2 de cadena de texto");

// "línea 1 de cadena de texto

// línea 2 de cadena de texto"


//ES6

// Para obtener el mismo efecto con cadenas de texto multilínea, con ES6 es posible escribir:

console.log(`línea 1 de la cadena de texto

línea 2 de la cadena de texto`);

// "línea 1 de la cadena de texto

// línea 2 de la cadena de texto"
```

Interpolación de expresiones

Para combinar expresiones dentro de cadenas de texto normales, se usa la siguiente sintaxis:

```
var a = 5;

var b = 10;

console.log("Quince es " + (a + b) + " y\nno " + (2 * a + b) + "."); // "Quince es 15 y // no 20."
```

Ahora, con las plantillas de cadena de texto, tenemos una sintaxis que habilita la misma funcionalidad, con un código más agradable a la vista y fácil de leer:

```
var a = 5; var b = 10;

console.log(`Quince es ${a + b} y\nno ${2 * a + b}.`); // "Quince es 15 y // no 20."
```

Anidamiento de plantillas

En ciertos momentos, anidar una plantilla es la forma más fácil y quizás más legible de tener cadenas configurables. Dentro de una plantilla con tildes invertidas, es sencillo permitir tildes invertidas interiores simplemente usándolos dentro de un marcador de posición ${ } dentro de la plantilla. Por ejemplo, si la condición a es verdadera: entonces devuelva este literal con plantilla.

En ES5:

```
var classes = 'header'

classes += (isLargeScreen() ?

" : item.isCollapsed ?

' icon-expander' : ' icon-collapser');
```

En ES6 con plantillas de cadena de texto y sin anidamiento:

```
const classes = `header ${ isLargeScreen() ? " :

(item.isCollapsed ? 'icon-expander' : 'icon-collapser') }`;
```

En ES5 con plantillas de cadena de texto anidadas:

```
const classes = `header ${ isLargeScreen() ? " :

`icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;
```

Plantillas de cadena de texto con postprocesador

Una forma más avanzada de plantillas de cadenas de texto son aquellas que contienen una función de postprocesado. Con ellas es posible modificar la salida de las plantillas, usando una función. El primer argumento contiene un array con las cadenas de texto de la plantilla ("Hola" y "mundo" en el ejemplo). El segundo y subsiguientes argumentos con los valores procesados (ya cocinados) de las expresiones de la plantilla (en este caso "15" y "50"). Finalmente, la función devuelve la cadena de texto manipulada. El nombre "tag" de la función no es nada especial, se puede usar cualquier nombre de función en su lugar.

```
var persona = 'Mike';
var edad = 28;

function myTag(strings, expPersona, expEdad) {
var str0 = strings[0]; // "Ese "
var str1 = strings[1]; // " es un "
// Hay tecnicamente un String después
// la expresión final (en nuestro ejemplo)
// pero es vacia (""), asi que se ignora.
// var str2 = strings[2];

var strEdad;
if (expEdad > 99) {
strEdad = 'viejo';
} else {
strEdad = 'joven';
}
// Podemos incluso retornar un string usando una plantilla de cadena de texto
return `${str0}${expPersona}${str1}${strEdad}`;
}

var salida = myTag`Ese ${ persona } es un ${ edad }`;
console.log(salida);
// Ese Mike es un joven
```

Cadenas en crudo (raw)

La propiedad especial raw, disponible en el primer argumento de las plantillas de cadenas de texto postprocesadas, nos permite acceder a las cadenas de texto tal como fueron ingresadas, sin procesar secuencias de escape

```
function tag(strings, ...values) {

console.log(strings.raw[0]);

// "linea 1 de cadena de texto \\n línea 2 de cadena de texto"

}

tag`línea 1 de cadena de texto \n línea 2 de cadena de texto`;
```

Adicionalmente, el método String.raw() https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String/raw permite crear cadenas de texto en crudo tal como serían generadas por la función por defecto de plantilla, concatenando sus partes.

```
var str = String.raw`¡Hola\n${2+3}!`;

// "¡Hola\n5!"

str.length; // 9

str.split('').join(',');

// "¡,H,o,l,a,\,n,5,!"
```

Seguridad

Las plantillas de cadenas de texto NO DEBEN ser construidas por usuarios no confiables, porque tienen acceso a variables y funciones.

```
`${console.warn("this es",this)}`; // "this es" Window


let a = 10;

console.warn(`${a+=20}`); // "30"
```

```
console.warn(a); // 30
```