

Evolucion de Javascript

Objetivo: Conocer que es lo que sucede cuando ejecutamos JavaScript

Conceptos Importantes:

Paralelismo: varios objetos realizando una acción cada uno **simultáneamente** (síncrono)

Ejemplo: 2 colas de personas que puede usar 2 máquinas de café, por lo tanto cada cola usa su máquina de café exclusiva.

Concurrencia: un solo objeto, con varias tareas “activas” entre las que va **alternando** (asíncrono)

Ejemplo: 2 colas de personas que desean usar una única máquina de café, por lo tanto las personas de ambas colas se deben ir turnando para usar la única máquina de café.

Síncrono (Bloqueante): Debo esperar que la tarea actual finalice para poder continuar con la siguiente.

Ejemplo: Cola para entrar en la sala de cine.

Asíncrono (No Bloqueante): Puedo ejecutar una segunda tarea mientras la primera se está ejecutando

Ejemplo: Pedirle la orden al mesero de un restaurante.

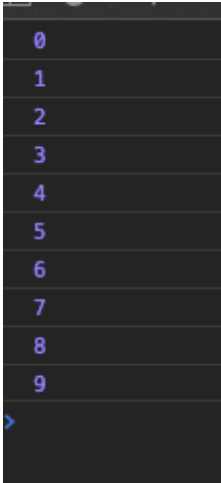
Javascript sincrono o asincrono?

¿Javascript sincrono o asincrono?. Esta es una de las preguntas más habituales cuando uno empieza a tener cierta experiencia en programación con JavaScript. ¿Cómo funciona el lenguaje en sí? ¿Es sincrono o es asincrono? Normalmente casi todo el mundo suele responder que el lenguaje es asincrono por sus peticiones Ajax etc. . Sin embargo las cosas no son tan sencillas como nos parece en un primer momento. Vamos a echarle un vistazo a cómo JavaScript funciona realmente. Para ello es suficiente con construir un bucle for.

```
<html>
<head>
<meta charset="utf-8">
<title>JS</title>
</head>
  <script type="text/javascript">
    for (let i=0;i<10;i++) {
      console.log(i);
    }
  </script>
<body>
```

```
</body>  
</html>
```

Vemos los números por la consola.

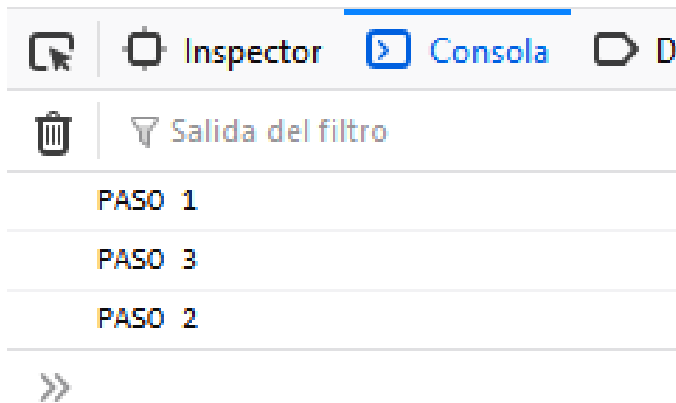


El resultado no nos sorprende para nada . Podemos ahora construir un programa que nos da la sensación de ser asíncrono en el cual pulsamos un botón y solicitamos que después de un intervalo de tiempo se muestre un mensaje por la consola.

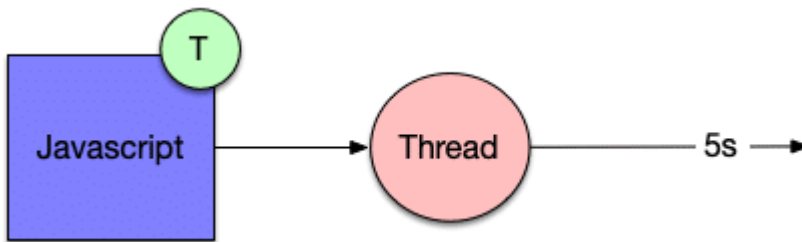
```
<html>  
<head>  
<meta charset="utf-8">  
<title>JS</title>  
</head>  
<script type="text/javascript">  
  function mensaje() {  
    console.log("PASO 1");  
    setTimeout(function() {  
      console.log("PASO 2");  
    }, 5000);  
    console.log("PASO 3");  
  }  
</script>  
<body>  
<input type="button" value="asincrono" onclick="mensaje()"/>  
</body>  
</html>
```

Si lo ejecutamos el comportamiento será el esperado en este caso al cabo de 5 segundos se muestra el resultado por la consola.

asíncrono



¿Así pues hemos ejecutado un código “asíncrono”? . Muchas personas piensan que cuando este código se ejecuta hay un Thread de JavaScript que se abre para realizar esta tarea al cabo de 5 segundos.



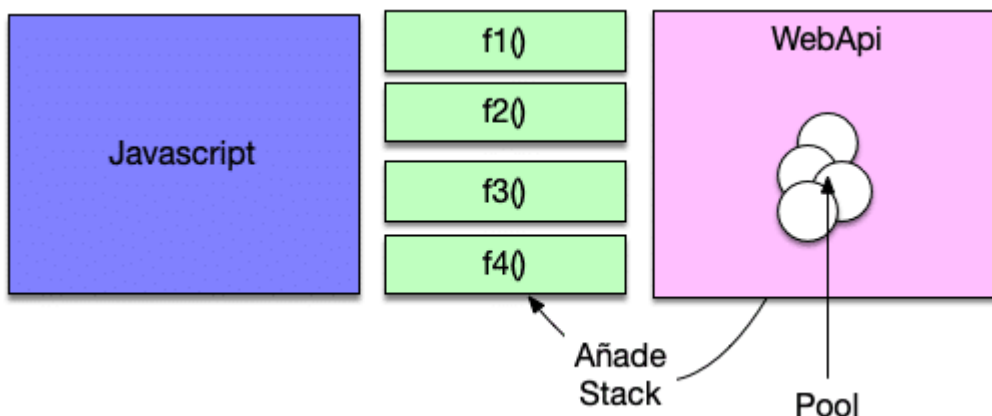
Algo similar a cuando realizamos este tipo de operaciones en el mundo Java. Lamentablemente la realidad es diferente y a veces cuesta entenderla. Para ello vamos a añadir un sencillo botón que nos ejecute una función de JavaScript con un alert a nuestro código



Si pulsamos el botón del setTimeout a los 5 segundos veremos el mensaje por la consola.

Ahora bien si pulsamos ese botón e inmediatamente después pulsamos el botón que nos muestra el alert la realidad es que NO veremos el mensaje de la consola pasados los 5 segundos. **De hecho no lo veremos nunca hasta que aceptemos el mensaje del alert.** ¿Qué está ocurriendo? . ¿Es JavaScript sincrónico o asíncrono? . La realidad es que el motor de Javascript es sincrónico y solo dispone de un Thread de ejecución. Si nosotros bloqueamos el Thread con un alert a la espera de que confirmemos el motor de Javascript no será capaz de ejecutar nada más y se quedará esperando eternamente.

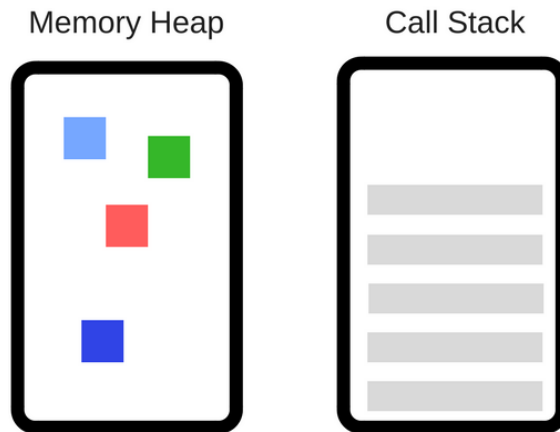
¿Entonces cómo funciona una petición Ajax que todos sabemos que es asíncrona? Ahora que tenemos claro que el motor de JavaScript es sincrónico. Las cosas no son tan complicadas como parece. El navegador tiene un pool de Thread a nivel de WebAPI fuera del runtime de JavaScript que es el que se encarga de realizar estas tareas asíncronas cómo puede ser una petición Ajax . Cuando esa petición Ajax termine el pool registra en la pila de llamadas de JavaScript una nueva función con el resultado de la operación. Esta función será ejecutada por el motor de JavaScript en su event loop de forma sincrónica. Es decir en cuando no tenga otra cosa que hacer.



Por lo tanto tengamos siempre en cuenta que el motor de JavaScript siempre es sincrónico y todas las peticiones que consideremos asíncronas se hacen a través del pool de Thread de WebAPI que es algo externo. Este al terminar nos devolverá un resultado.

Engine Java Script

Miremos un dibujo que representa el runtime de v8 (el runtime que usa chrome y node)



el engine consiste de dos elementos principales

- Memory Heap: es donde se realiza la aloca  n de memoria
- Call Stack: es donde el runtime mantiene un track de las llamadas a las funciones

Call Stack

Para los que no sepan un stack (tambi  n llamado pila) es una estructura simple, similar a un arreglo en el que solo se puede agregar items al final (push) , y remover el   ltimo (pop).

El proceso que realiza el call stack es simple, cuando se est   a punto de ejecutar una funci  n, esta es a  adida al stack. Si la funci  n llama a su vez, a otra funci  n, es agregada sobre la anterior. Si en alg  n momento de la ejecuci  n hay un error, este se imprimir   en la consola con un mensaje y el estado del call stack al momento en que ocurri  .

Javascript es un lenguaje single threaded. Esto quiere decir que durante la ejecuci  n de un script existe un solo thread que ejecuta el c  digo. Por lo tanto solo se cuenta con un call stack

Veamos un ejemplo:

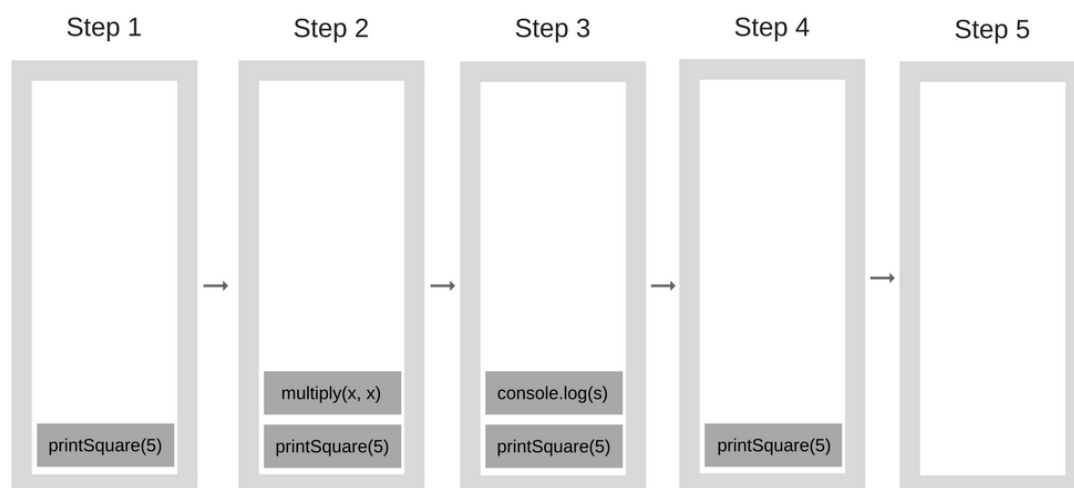
```
function multiply (x, y) {  
  return x * y;  
}
```

```
function printSquare (x) {  
  var s = multiply(x, x);  
  console.log(s);  
}
```

```
printSquare(5);
```

Los estados del call stack serían:

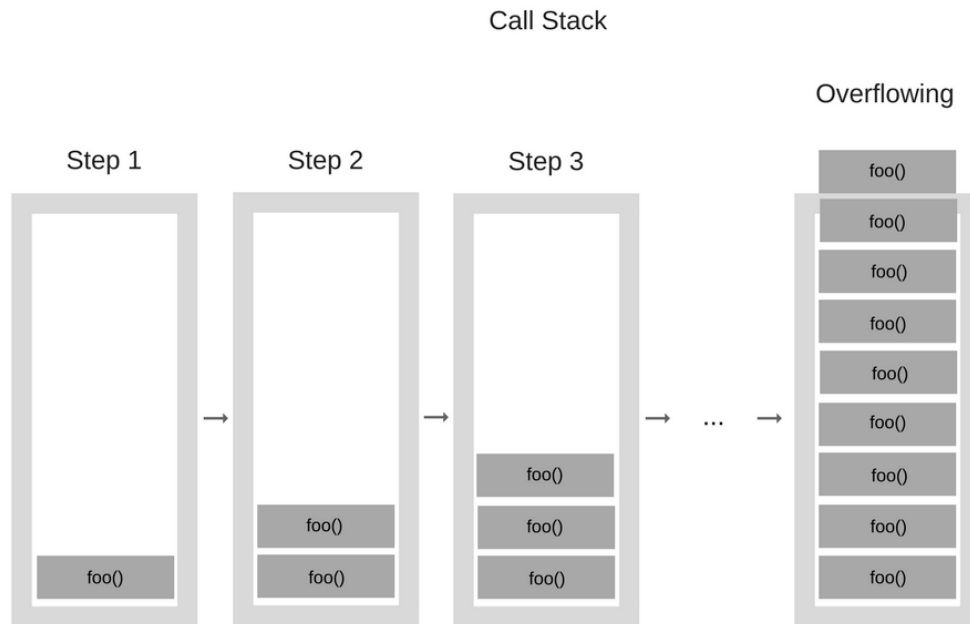
Call Stack



Y que pasa si tenemos una función de esta manera:

```
function foo() {  
  foo();  
}
```

```
foo();
```



Lo que sucedería es que en algún momento la cantidad de funciones llamadas excede el tamaño del stack , por lo que el navegador mostrará este error:

✖ ▶ Uncaught RangeError: Maximum call stack size exceeded

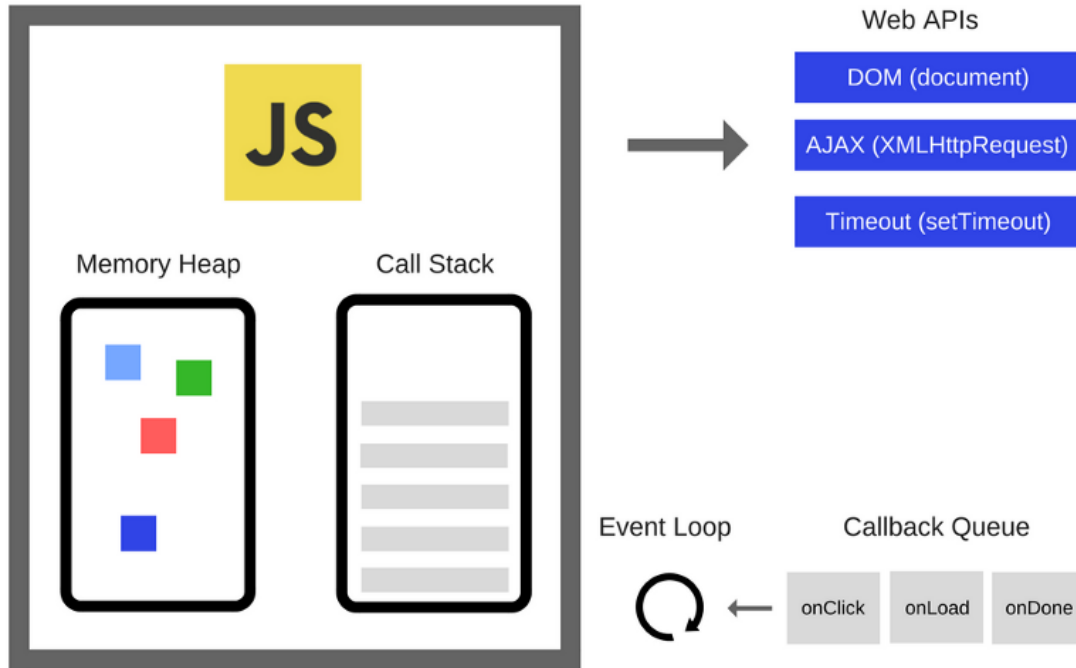
La situación anterior podemos subsanarla gracias al Event Loop.

Event Loop (Bucle de Eventos)

Web APIs

Javascript (del lado del cliente en particular) tiene muchas [APIs disponibles](#), estas no son parte del lenguaje Javascript en sí, más bien están construidas sobre Javascript, proveyéndonos de súper poderes extra para usar en nuestro código Javascript, alguna de ellas van desde setTimeout y XMLHttpRequest hasta WebGL o FileReader.

Por lo tanto este es el gráfico que muestra una visión más abarcativa de javascript. En este se puede ver el runtime, más las Web APIs y el callback queue (cola de devolución de llamada) del cual hablaremos más adelante.



Al haber un solo thread es importante no escribir código bloqueante para la UI no quede bloqueada.

Pero ¿Cómo hacemos para escribir código no bloqueante?

La solución son **callbacks asincrónicas**. Para esto combinamos el uso de **callbacks** (funciones que pasamos como parámetros a otras funciones) con las WEB API's.

```
function mensaje() {
  console.log("PASO 1");
  setTimeout(function paso2() {
    console.log("PASO 2");
  }, 5000);
  console.log("PASO 3");
}
```

```
/*
* Resultados:
* => PASO 1
* => PASO 3
* => PASO 2
*/
```


Como pueden ver la ejecución no se queda bloqueada en **setTimeout()** ya que imprime la instrucción que le sigue primero) ¿Pero entonces cómo es que posible que esto sea así si solo existe un solo thread? ¿Cómo es que la ejecución continua y al mismo tiempo el setTimeout hace la cuenta regresiva para ejecutar la función pasada como callback?

Esto es porque, como mencione anteriormente, el **setTimeout NO** es parte del runtime. Sino que es provista por el navegador como WEB APIs (o en el caso de Node por c++ apis). Los cuales SI se ejecutan en un thread distinto.

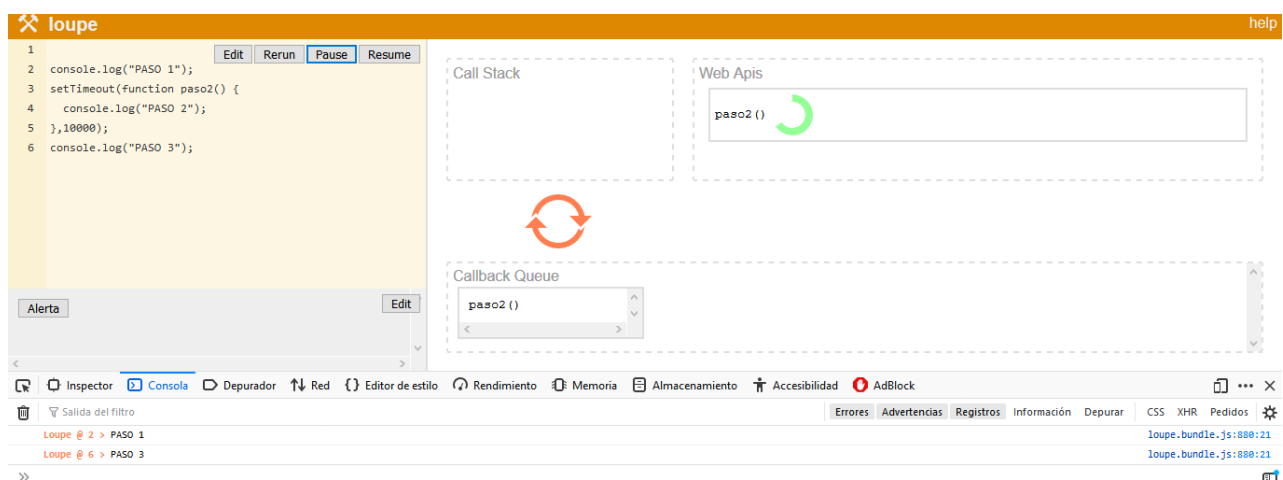
¿Como se maneja esto con una única call stack?

Existe otra estructura donde se guardan las funciones que deben ser ejecutadas luego de cierto evento (timeout, click, mouse move), en el caso del código de ejemplo de arriba se guarda que, cuando el timeout termine se debe ejecutar la función `paso2()`. Tener en cuenta que cuando sucede el evento, esta estructura no es la que la ejecuta y tampoco las agrega al call stack ya que sino podría pasar que la función se ejecutará en medio de otro código. Lo que hace es enviarla a la Callback Queue.

Lo que hace el event loop es fijarse el call stack, y si está vacío (es decir no hay nada ejecutandose) envía la primera función que esté en la callback queue al call stack y comienza a ejecutarse.

Luego de terminar la cuenta regresiva del `setTimeout()` (que no es ejecutada en el runtime de javascript), `timeoutCallback()` será enviada a la callback queue. El event loop chequeara el Call Stack, si este está vacío enviará `paso2()` al call stack para su ejecución.

El flujo en imágenes de todo este trabalenguas seria:



Visitar <http://latentflip.com/loupe/> para ver de forma visual como trabaja Event Loop

Ver Video de Philip Roberts: ¿Que diablos es el "event loop" (bucle de eventos) de todos modos?
<https://youtu.be/8aGhZQkoFbQ>

De esta manera se logra que el código sea no bloqueante, en vez de un `setTimeout` podría ser una llamada a un servidor, en donde habría que esperar que se procese nuestra solicitud y nos envíe una respuesta, el cual sería tiempo ocioso si no contáramos con callbacks asincrónicos, de modo que el runtime pueda seguir con otro código. Una vez que la respuesta haya llegado del servidor y Call Stack esté vacío, se podrá procesar la respuesta (mediante la función pasada como callback) y hacer algo con ella, por ejemplo mostrarla al usuario.

Funciones Flecha =>

Alternativa sintáctica que permite **reducción de la sintaxis, un código más limpio, y una corrección necesaria** del contexto y valor de `'this'`.

Sintaxis Básica

```
(param1, param2, ..., paramN) => { sentencias }
```

```
(param1, param2, ..., paramN) => expresion
```

```
// Equivalente a: () => { return expresion; }
```

```
// Los paréntesis son opcionales cuando hay un solo parámetro: singleParam => { statements }
```

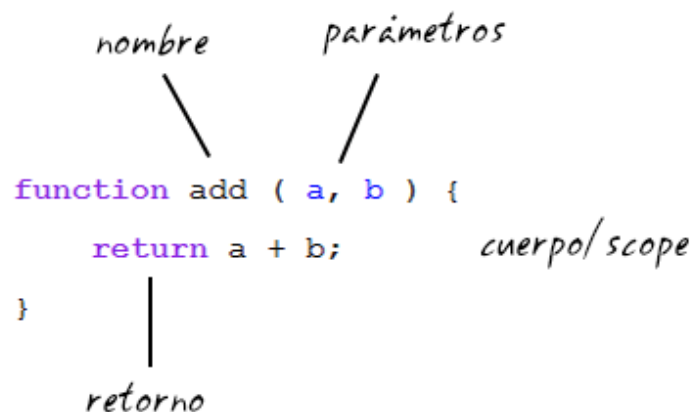
```
(singleParam) => { sentencias } singleParam => { sentencias }
```

```
// Una función sin parámetros requiere paréntesis:
```

```
() => { sentencias }
```

Ejemplo

Función JavaScript Tradicional



```
function add ( a, b ) {  
    return a + b;  
}
```

Diagram labels:

- nombre* points to `add`.
- parámetros* points to `(a, b)`.
- cuerpo/ scope* points to the curly braces `{ ... }`.
- retorno* points to `return a + b;`.

Con las funciones flecha, buscamos simplificar todo lo anterior de un modo mucho más directo y declarativo:

- Eliminamos la palabra reservada *function* y nos limitamos a recoger los parámetros mediante los paréntesis tradicionales.
- Podemos eliminar las llaves que delimitan el *scope* *abriéndolo* con una flecha.
- Podemos eliminar la palabra reservada *return*.

nombre parâmetros flecha / scope
 \ | /
 var add = (x, y) => x + y;
 |
 retorno directo

```
var p = new Persona();
```

SALIDA:1
2
3

..... así sucesivamente incrementando en 1

Función Callback

Un [*callback*](#) (llamada de vuelta) es una función que recibe como argumento otra función y la ejecuta. Es una de las técnicas y formas más común para el control de la **asincronía** dentro de *JavaScript*, *dado que* a menudo se utilizan para continuar con la ejecución del código después de que se haya completado una operación a síncrona.

Es importante tener en cuenta que cuando pasamos un *callback* solo pasamos la definición de la función y no la ejecutamos en el parámetro. Así, **la función contenedora elige cuándo ejecutar el *callback*.**

Un ejemplo muy común de *callback* es una función escuchadora de un evento.

```
function showAlert(){  
    alert('Alerta');  
}  
button.addEventListener('click', showAlert);
```

En este ejemplo, *showAlert* es un *callback*. También podemos escribir el *callback* como función anónima:

```
button.addEventListener('click', function(){  
    alert('Alerta');  
});
```

Ejemplo Callback

```
<html>  
<head>  
<meta charset="utf-8">  
<title>JS</title>  
</head>  
<body>  
<script>  
/**
```

```
* Simulacion de asincronia en JavaScript (Funcion A).
*
* @param Array    list    Lista de numeros.
* @param Function callback Recibe la funcion B.
*/
const asincronia = (list, callback) => {
    //se valida que el parametro list sea un arreglo
    //y no este vacio
    if (list instanceof Array && list.length > 0) {
        let suma = list.map(valor => Math.pow(valor, 2));
        return callback(null, suma);
    }
    //si no se cumple la condicion se manda un error.
    else {
        let error = new Error("Error de ejecución:Lista Vacía ");
        return callback(error, null);
    }
};
//===== Consiguiendo la respuesta correcta =====//
console.log("Consiguiendo la respuesta correcta");
asincronia([2, 3, 4, 5], (error, result) => (error) ?
console.error(error) : console.log(result));

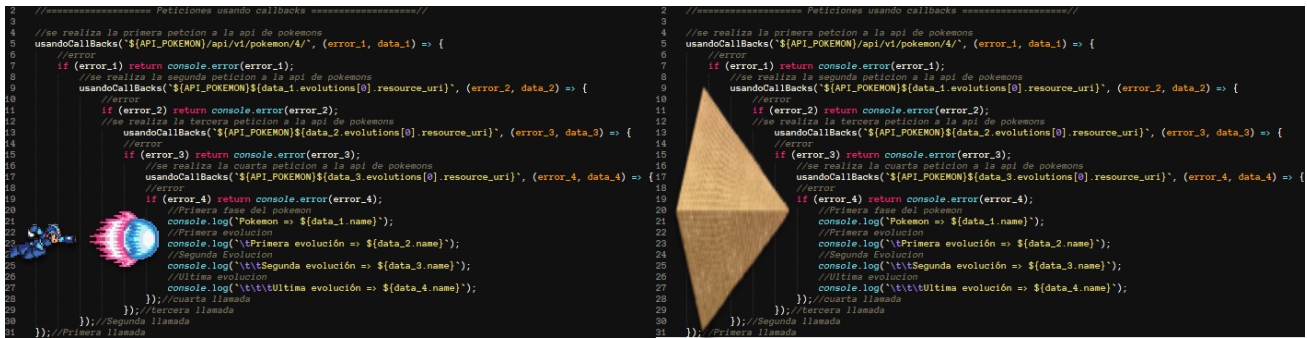
//===== Consiguiendo el error =====//
console.log("Consiguiendo el error");
asincronia([], (error, result) => (error) ? console.error(error) :
console.log(result));
</script>
</body>
```

Desventaja de Callbacks

Callback Hell

Anidamiento de N niveles de Callbacks

Callbacks dentro de otros Callbacks



Este problema surge por los niveles de profundidad que se generan en el código lo cual dificultará la lectura y el mantenimiento de este.

Con las nuevas especificaciones de [EcmaScript 6](#) surge una nueva forma para el manejo de la *asincronía*, las *promesas*.

Promesas

Una [Promise](#) (promesa en castellano) es un objeto que representa la terminación o el fracaso de una operación asíncrona. Nos permite menos líneas y mucha menos indentación en el código

Explicaremos las promesas con un ejemplo:

Supongamos que vamos a comprar comida a un restaurante de comida rápida, cuando terminamos de pagar por nuestra comida nos dan un ticket con un número, cuando llamen a ese número podemos entonces ir a buscar nuestra comida.

Ese ticket que nos dieron es nuestra promesa, ese ticket nos indica que eventualmente vamos a tener nuestra comida, pero que todavía no la tenemos. Cuando llaman a ese número para que vayamos a buscar la comida entonces quiere decir que la promesa se completó. Pero resulta que una promesa se puede completar correctamente o puede ocurrir un error, ¿Qué error puede ocurrir en nuestro caso? Por ejemplo puede pasar que el restaurante no tenga más comida, entonces cuando nos llamen con nuestro número pueden pasar dos cosas.

1. Nuestro pedido se resuelve y obtenemos la comida.
2. Nuestro pedido es rechazado y obtenemos una razón del por qué.

Pongamos esto en código:

```
const ticket = getFood();

ticket
  .then(food => eatFood(food))
  .catch(error => getRefund(error));
```

Cuando tratamos de obtener la comida (getFood) obtuvimos una promesa (ticket), si esta se resuelve correctamente entonces recibimos nuestra comida (food) y nos la comemos (eatFood). Si nuestro pedido es rechazado entonces obtenemos la razón (error) y pedimos que nos devuelvan el dinero (getRefund).

Crear una promesa

Las promesas se crean usando un constructor llamado Promise y pasándole una función que recibe dos parámetros, resolve y reject, que nos permiten indicarle a esta que se **resolvió** o se **rechazó**.

```
const promise = new Promise((resolve, reject) => {
  const number = Math.floor(Math.random() * 10);

  setTimeout(
    () => number > 5
      ? resolve(number)
      : reject(new Error('Menor a 5')),
    1000
  );
});

promise
  .then(number => console.log(number))
  .catch(error => console.error(error));
```

Lo que acabamos de hacer es crear una nueva promesa que se va a completar luego de 1 segundo, si el número aleatorio que generamos es mayor a 5 entonces se resuelve, si es menor a 5 entonces es rechazada y obtenemos un error.

Estados de las promesas

Esto nos lleva a hablar del estado de una promesa, básicamente existen 3 posibles estados.

- **Pendiente**
- **Resuelta**
- **Rechazada**

Una promesa originalmente está **Pendiente**. Cuando llamamos a resolve entonces la promesa pasa a estar **Resuelta**, si llamamos a reject pasa a estar **Rechazada**, usualmente cuando es rechazada obtenemos un error que nos va a indicar la razón del rechazo. Cuando una promesa se *resuelve* entonces se ejecuta la función que pasamos al método .then, si la promesa es *rechazada* entonces se ejecuta la función que pasamos a .catch, de esta forma podemos controlar el flujo de datos.

También es posible pasar una segunda función a .then la cual se ejecutaría en caso de un error en vez de ejecutar el .catch

Encadenando promesas

```
checkWeather('buenos aires')
  .then(weather => {
    if (weather === 'well') {
      return checkFlights('buenos aires');
    }
    throw new Error('el clima es malo');
  })
  .then(flights => buyTicket(flights[0]))
  .then(ticket => {
    console.log('ticket n° %d', ticket.number);
  })
  .catch(error => console.error(error));
```

En el ejemplo anterior usamos muchos `.then` y llamamos a varias funciones que devuelven promesas. Este patrón se llama **promise chaining** o encadenamiento de promesas.

Básicamente nos evita anidar código (como nos pasacundo usamos callbacks), en vez de eso, una promesa puede devolver otra promesa y llamar al siguiente `.then` de la cadena.

Promesas en paralelo

Es muy común que necesitemos realizar múltiples funciones asíncronas al mismo tiempo, por ejemplo para obtener varios datos de un API. Para eso la clase `Promise` tiene un método estático llamado `Promise.all` el cual recibe un único parámetro, una lista de promesas las cuales se ejecutan simultáneamente, si alguna de estas es rechazada entonces toda la lista lo es, pero si todas se resuelven entonces podemos obtener una lista de todas las respuestas.

```
Promise.all([readFile('./archivo1.txt'),
readFile('./archivo2.txt')])
  .then(finalData => console.log(finalData))
  .catch(error => console.error(error));
```

Lo que hacemos en el ejemplo de arriba es leer 2 archivos al mismo tiempo, si en algún momento ocurrió un error lo mostramos como tal en consola, si todo se resuelve bien entonces escribimos en consola la lista de contenidos de archivos.

Carrera de promesas

Antes hablamos de ejecutar varias promesas en paralelo y obtener una respuesta cuando todas se completan, existe otro método que nos permite correr varias al tiempo, pero solo obtener el resultado de la primer promesa. Gracias a esto es posible mandar múltiples peticiones HTTP a un API y luego recibir una sola respuesta, la primera. Este método se llama `Promise.race`.

```
Promise.race([readFile('./archivo1.txt'),
readFile('./archivo2.txt')])
  .then(resolve)
```



```
.then(readFile)
.then(data => console.log(data))
.catch(error => console.error(error));
```

Como vemos en el ejemplo otra vez leemos 2 archivos, pero esta vez solo obtenemos el contenido de 1, el que primero se termine de leer. O si alguno se completó con un error entonces entramos al catch y mostramos el error en consola.

Conclusiones

Trabajar con promesas nos facilita mucho el control de flujos de datos asíncronos en una aplicación, además las promesas son la base para luego poder implementar características más avanzadas de JavaScript como **Async/Await** que nos facilitan aún más nuestro código.

Async/Await (ECMAScript 7)

Una función asíncrona se declara con `async`, y no es más que una función que devuelve una promesa. Lo importante es que no tenemos que declarar el objeto `Promise` en ningún momento.

Estas funciones asíncronas, a su vez, pueden llamar a otras funciones que devuelven promesas. En lugar de `.then`, `Promise.all` y demás, podemos utilizar `await`. Esta instrucción detiene la ejecución de una función hasta que la promesa se ha resuelto, resolviendo así todo en una única línea.

Es importante recordar que la instrucción `await` sólo puede utilizarse dentro de una función declarada como `async`.

En el siguiente ejemplo, podemos ver cómo creamos una función asíncrona para realizar dos llamadas a la API de Marvel. Esta función devuelve el resultado de ambas llamadas, y se pinta por pantalla en otra función asíncrona.

```
async function getHulkAndSpiderman() {
  const hulkResponse = await fetch('https://swapi.co/api/marvel/1');
  const hulk = await hulkResponse.json();

  const spidermanResponse = await
  fetch('https://swapi.co/api/marvel/2/');
  const spiderman = await spidermanResponse.json();

  return { hulk, spiderman };
}

async function init() {
```

```
const hulkAndSpiderman = await getHulkAndSpiderman ();
const pre = document.createElement('pre');

pre.innerText = JSON.stringify(hulkAndSpiderman, null, 2);
document.body.appendChild(pre);
}

init();
```

Las promesas tienen un bloque específico para capturar errores. ¿Y cómo hago esto yo ahora? Fácil: con un bloque try/catch de toda la vida:

```
async function getLukeAndTatooine() {
  const hulkResponse = await fetch('https://swapi.co/api/marvel/1');
  const hulk = await hulkResponse.json();

  const spidermanResponse = await
  fetch('https://swapi.co/api/marvel/2/');
  const spiderman = await spidermanResponse.json();

  return { hulk, spiderman };
}

async function init() {
  try {
    const hulkAndSpiderman = await getHulkAndSpiderman ();
    const pre = document.createElement('pre');

    pre.innerText = JSON.stringify(hulkAndSpiderman, null, 2);
    document.body.appendChild(pre);
  } catch(error) {
    alert(error);
  }
}

init();
```

Conclusiones

Cómo hemos podido comprobar, los operadores `async/await` proporcionan una forma mucho más simple para trabajar con promesas, permitiendo trabajar de forma síncrona.