Curso de Asincronismo con JavaScript

# Introducción al asincronismo

JavaScript sólo puede hacer una cosa a la vez, sin embargo; es capaz de delegar la ejecución de ciertas funciones a otros procesos. Este modelo de concurrencia se llama EventLoop.

JavaScript delega en el navegador ciertas tareas y les asocia funciones que deberán ser ejecutadas al ser completadas. Estas funciones se llaman callbacks, y una vez que el navegador ha regresado con la respuesta, el callback asociado pasa a la cola de tareas para ser ejecutado una vez que JavaScript haya terminado todas las instrucciones que están en la pila de ejecución.

Si se acumulan funciones en la cola de tareas y JavaScript se encuentra ejecutando procesos muy pesados, el EventLoop quedará bloqueado y esas funciones pudieran tardar demasiado en ejecutarse.



**API**  
Interfaz de programación de aplicaciones (Application Programming Interface). Es un conjunto de rutinas que provee acceso a funciones de un determinado software.  
**Concurrencia**  
Cuando dos o más tareas progresan simultáneamente.  
**Paralelismo**  
Cuando dos o más tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.  
**Bloqueante**  
Una llamada u operación bloqueante no devuelve el control a nuestra aplicación hasta que se ha completado. Por tanto el thread queda bloqueado en estado de espera.  
**Síncrono**  
Es frecuente emplear ‘bloqueante’ y ‘síncrono’ como sinónimos, dando a entender que toda la operación de entrada/salida se ejecuta de forma secuencial y, por tanto, debemos esperar a que se complete para procesar el resultado.  
**Asíncrono**  
La finalización de la operación I/O se señaliza más tarde, mediante un mecanismo específico como por ejemplo un callback, una promesa o un evento, lo que hace posible que la respuesta sea procesada en diferido.  
**Call Stack**  
La pila de llamadas, se encarga de albergar las instrucciones que deben ejecutarse. Nos indica en qué punto del programa estamos, por donde vamos.  
**Heap**  
Región de memoria libre, normalmente de gran tamaño, dedicada al alojamiento dinámico de objetos. Es compartida por todo el programa y controlada por un recolector de basura que se encarga de liberar aquello que no se necesita.  
**Cola o Queue**  
Cada vez que nuestro programa recibe una notificación del exterior o de otro contexto distinto al de la aplicación, el mensaje se inserta en una cola de mensajes pendientes y se registra su callback correspondiente.  
**Eventloop o Loop de eventos**  
Cuando la pila de llamadas (call stack) se vacía, es decir, no hay nada más que ejecutar, se  
procesan los mensajes de la cola. Con cada ‘tick’ del bucle de eventos, se procesa un nuevo mensaje.  
**Hoisting**  
Sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código en tiempo de compilación.  
**DOM**  
DOM permite acceder y manipular las páginas XHTML como si fueran documentos XML. De  
hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.  
**XML**  
Lenguaje de marcado creado para la transferencia de información, legible tanto para seres  
humanos como para aplicaciones informáticas, y basado en una sencillez extrema y una rígida sintaxis. Así como el HTML estaba basado y era un subconjunto de SGML, la reformulación del primero bajo la sintaxis de XML dio lugar al XHTML; XHTML es, por tanto, un subconjunto de XML.  
**Events**  
Comportamientos del usuario que interactúa con una página que pueden detectarse para lanzar una acción, como por ejemplo que el usuario haga click en un elemento (onclick), que elija una opción de un desplegable (onselect), que pase el ratón sobre un objeto (onmouseover), etc.  
**Compilar**  
Compilar es generar código ejecutable por una máquina, que puede ser física o abstracta como la máquina virtual de Java.  
**Transpilar**  
Transpilar es generar a partir de código en un lenguaje código en otro lenguaje. Es decir, un  
programa produce otro programa en otro lenguaje cuyo comportamiento es el mismo que el  
original.

# Definición Estructura Callback

La definición que el profe Oscar nos da: *“Es una función que al crearla le pasamos como parámetro una segunda función”*. Según lo que entiendo, eso no haría referencia directamente al *callback*, sino a la función que recibe como parámetro otra función.  
Una función que recibe otra función como parámetro se le denomina función de orden superior (higher-order function).  
El callback en este caso sería la función que es pasada como parámetro, mas no la función que lo recibe.

**Primero:** Los callbacks son el nombre de una convención para usar funciones que llaman a otras en JavaScript. No hay una palabra reservada llamada “callback” en el lenguaje JavaScript que haga que nuestro código sea diferente o especial,  
es más una convención.  
Tal es el caso que en lugar de llamar “callback” en el ejemplo de la clase, podemos llamarlo “suma” y funcionara igualmente.

  


**Segundo y para qué sirven?** La mayoría estamos acostumbrados a programar de manera sincrona, es decir le indicamos al código que por ejemplo defina un Valor “X” y con otro valor “Y” y realizamos un cálculo (Por ejemplo una multiplicación).

El problema radica en que por ejemplo si quisiéramos crear un programa que nos convierta nuestra moneda (pesos) a su equivalente en Bitcoin, podemos definir X (Valor de nuestro dinero) pero NO podemos definir de manera implícita “Y” (Precio del Bitcoin) porque es algo muy volátil. Entonces necesitamos obtener el precio del Bitcoin de una API, nuestro programa realiza una multiplicación de X \* Y sin embargo no tenemos Y (precio del bitcion)  
porque tenemos que esperar que el API nos conteste cual es este valor. Es ahí donde sirven los callback  
.  
Existen dos Metodos A y B  
-El método B hace el calculo de nuestros Pesos \* PrecioBitcoin  
-El método A obtiene el precio del API de Bitcoin  
Entonces el método B es llamado por el método A cuando por fin lee y obtiene el precio del Bitcoin, solo hasta entonces tiene sentido que multipliquemos nuestros valores.

# Peticiones a APIs usando Callbacks



# Múltiples Peticiones a un API con Callbacks

Es recomendable de no realizar más de 3 callback para no caer en un callback Hell, si tu proyecto tiene una función con más de 3 callback, se recomienda hacer una revisión y utilizar una mejor forma de ejecutar el código, para ello están las promesas o el Async Away



# Implementando Promesas



**Promise.all(iterable)**  
Devuelve una de dos promesas:

* una que se cumple cuando todas las promesas en el argumento iterable han sido cumplidas,
* o una que se rechaza tan pronto como una de las promesas del argumento iterable es rechazada.

Si la promesa retornada es cumplida, lo hace con un arreglo de los valores de las promesas cumplidas en el mismo orden definido en el iterable.

Si la promesa retornada es rechazada, es rechazada con la razón de la primera promesa rechazada en el iterable. Este método puede ser útil para agregar resultados de múltiples promesas

**Promise.race(iterable)**  
Devuelve una promesa que se cumple o rechaza tan pronto como una de las promesas del iterable se cumple o rechaza, con el valor o razón de esa promesa.

**Promise.reject(reason)**  
Devuelve un objeto Promise que es rechazado con la razón dada.

# Conociendo Async/await



# Resolver problema con Async/Await

