

Callbacks y APIs (Parte I)

Introducción a la asincronía

Competencias

- Definir el término de asincronía en JavaScript.
- Describir el funcionamiento del Event Loop para entender la programación asíncrona.
- Comprender el concepto de Race Condition para observar el funcionamiento de la asincronía

Introducción

Cuando desarrollamos aplicaciones web, normalmente realizamos tareas de forma síncrona, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que el orden o flujo del programa es sencillo. Pero a partir de este capítulo, el trabajo se concentrará en lograr que nuestro programa ejecute algunas tareas de manera asíncrona, ya que la programación actual a partir de ES6 no es solamente secuencial, sino que se puede trabajar de forma asíncrona. Por consiguiente, la asincronía es uno de los pilares fundamentales de JavaScript en la actualidad, por esta razón, será uno de los principales objetivos que aprenderás a lo largo de este capítulo y te ayudarán a realizar un código más ordenado y potente como todo un profesional con este lenguaje de programación.

Sincronía y Asincronía

Estos dos conceptos hacen referencia al tiempo en que se ejecutan múltiples procesos, tareas, eventos, métodos o funciones. Por un lado tenemos el término **Síncrono**, y es cuando los procesos se ejecutan uno tras otro, es decir, comienza una tarea y esta debe finalizar para que comience la siguiente y así sucesivamente con las demás tareas. Por otra parte, se tiene el término **Asíncrono**, y es cuando los procesos se ejecutan todos a la vez y no necesitan esperar a que finalicen los otros. Para apreciar mejor ambos conceptos, se puede observar la siguiente imagen con la ejecución de actividades síncronas y asíncronas.

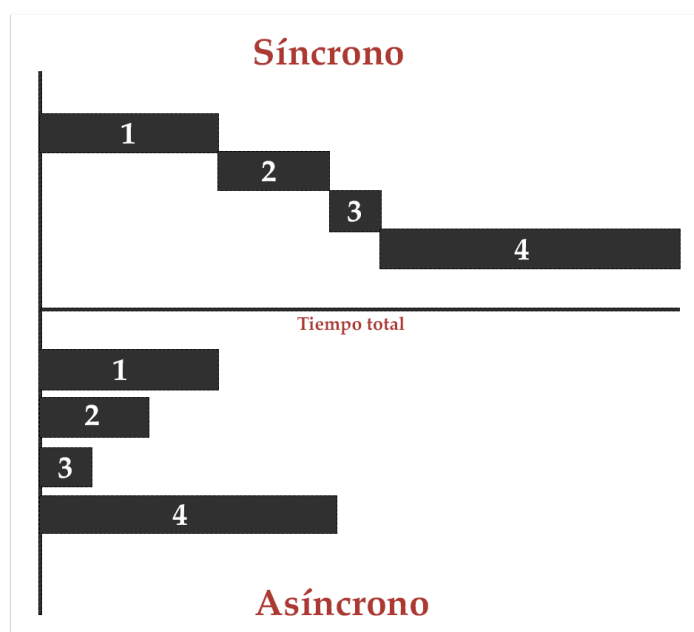


Imagen 1. Diferencias entre ejecución síncrona y asíncrona.

Fuente: Desafío Latam

Una lista de procesos se ejecuta de forma **síncrona** (o secuencial) cuando cada uno de los eventos debe ejecutarse y finalizar para que pueda iniciar el siguiente. Por ejemplo, para pagar el estacionamiento en un centro comercial cada persona debe ir a la máquina y esperar a que la gente que está delante suyo pague el ticket. Cuando la persona que está frente a la máquina retira el ticket, puede avanzar la siguiente.

Pero en el caso de una ejecución síncrona, la respuesta sucede a futuro, es decir, una operación asíncrona no esperará el resultado. Cada operación se ejecuta y devuelve inmediatamente el control al hilo (Thread), evitando el bloqueo. Finalmente, cuando cada operación termine se enviará una notificación de que ha terminado, es entonces cuando la respuesta se encola para ser procesada.

Siguiendo con el ejemplo anterior, en un centro comercial hay más de una máquina para pagar el ticket, por lo tanto, puede haber varias personas pagando al mismo tiempo, en donde cada persona recibirá la respuesta a su debido tiempo, sin interferir en el pago de las otras personas. Por ende, dentro de la ejecución de procesos asíncronos se pueden encontrar dos conceptos relacionados y muy importantes a la hora de manejar una solución, éstos son la concurrencia y el paralelismo.

Concurrencia: Esto se produce cuando dos o más tareas progresan simultáneamente. Como se muestra en la siguiente imagen.

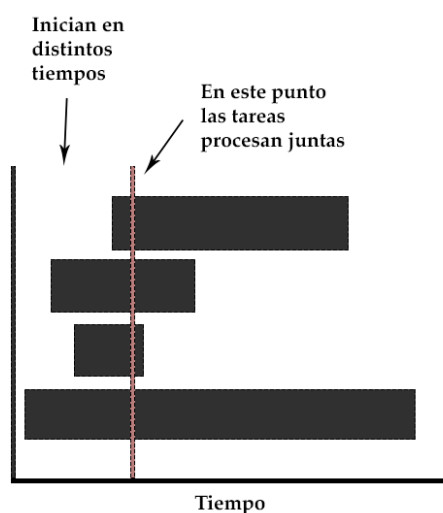


Imagen 2. Concurrencia.
Fuente: Desafío Latam

Paralelismo: Ocurre cuando dos o más tareas se ejecutan, literalmente a la vez, en el mismo instante de tiempo. Como se muestra en la siguiente imagen.

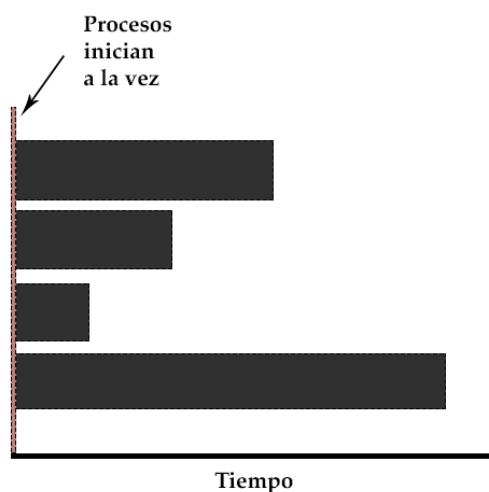


Imagen 3. Paralelismo.
Fuente: Desafío Latam

Event Loop o Bucle de Eventos

Para definir en pocas palabras el event loop, es el que se encarga de implementar las operaciones asíncronas. El event loop se puede interpretar como un manejador del Call Stack en el motor JavaScript V8 que fue desarrollado por Google para usarse en el lado del cliente (Google Chrome) y en el lado del servidor (Node). La siguiente imagen representa al Engine de Chrome en el que tenemos 2 elementos principales, Memory Heap y Call Stack.

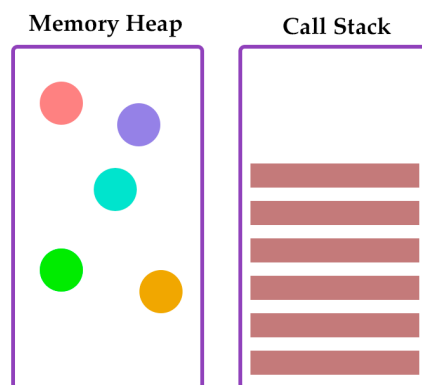


Imagen 4. Representación Engine V8 de Chrome.

Fuente: Desafío Latam

Memory Heap

El Memory Heap concentra todos los objetos y datos dinámicos, como las variables y constantes que debe sostener en la memoria durante la ejecución de las aplicaciones.

Call stack

Es una pila de procesos, parecida a una lista ordenada de tareas al que se van agregando sentencias para ser ejecutadas, donde cada proceso que agregamos va al final mientras espera a que se ejecute el resto de operaciones que le anteceden.

El event loop no solo implica poner procesos al Call Stack, sino que necesita de otros componentes en los que se determina cuál proceso se encola para llamarse, entre ellos WEB API y Callback Queue forman parte de éste.

WEB APIs

Las WEB APIs son funciones disponibilizadas por el navegador y que se pueden usar en JavaScript para comunicarnos e interactuar con el Frontend. Entre ellas están las de manipulación del DOM, geolocalización, notificaciones y muchas más. Puedes ver la lista completa de WEB APIs en el siguiente [enlace](#).

Callback Queue

La cola de devolución de llamada (Callback Queue), es una lista de funciones que le envía la WEB API y que quedan en espera a ser insertadas al Call Stack para ejecutarse. Asimismo, está basada en una estructura de datos del tipo FIFO, es decir, el primer dato en entrar es el primero en salir.

El flujo para que se ejecute una tarea se realiza de la siguiente forma, vamos a suponer que tenemos 3 funciones y las llamaremos A, B y C. Primero la función A llama a la función B, luego B llama a C y al final C muestra un mensaje en la consola. Este ejemplo mostrará cómo se ejecutarán las 3 funciones en el event loop.

Para los ejemplos de Event Loop utilizaremos una página que demuestra de manera gráfica los pasos que se van realizando desde que comienza la ejecución hasta que termina. Esto no lo haremos en la consola del navegador ya que los procesos ocurren tan rápido que son prácticamente imperceptibles, entonces para que podamos ver lo que está ocurriendo utilizaremos el sitio web [Loupe](#) que es de propiedad de [Philip Roberts](#), en él realizaremos nuestras pruebas. Hay que mencionar que esta página no acepta código ES6, por lo que se debe utilizar código en ES5 solamente.

Ejercicio guiado: Callback Queue

Para mostrar el funcionamiento del **Callback Queue** se realizará un ejemplo implementado la página mencionada anteriormente, más un código que tenga varias funciones anidadas y la última función muestre por la consola el literal "C", una función debe llamar a la otra. Por lo tanto, sigamos los siguientes pasos:

- **Paso 1:** Ingresa a la página [Loupe](#) con tu navegador favorito.
- **Paso 2:** Escribe el código mostrado a continuación en la página web, en el recuadro del lado izquierdo de color naranja claro, como se muestra en la imagen número 5.

```
function C(){  
    console.log('C')  
}  
function B(){  
    C()  
}  
function A(){  
    B()  
}  
A();
```

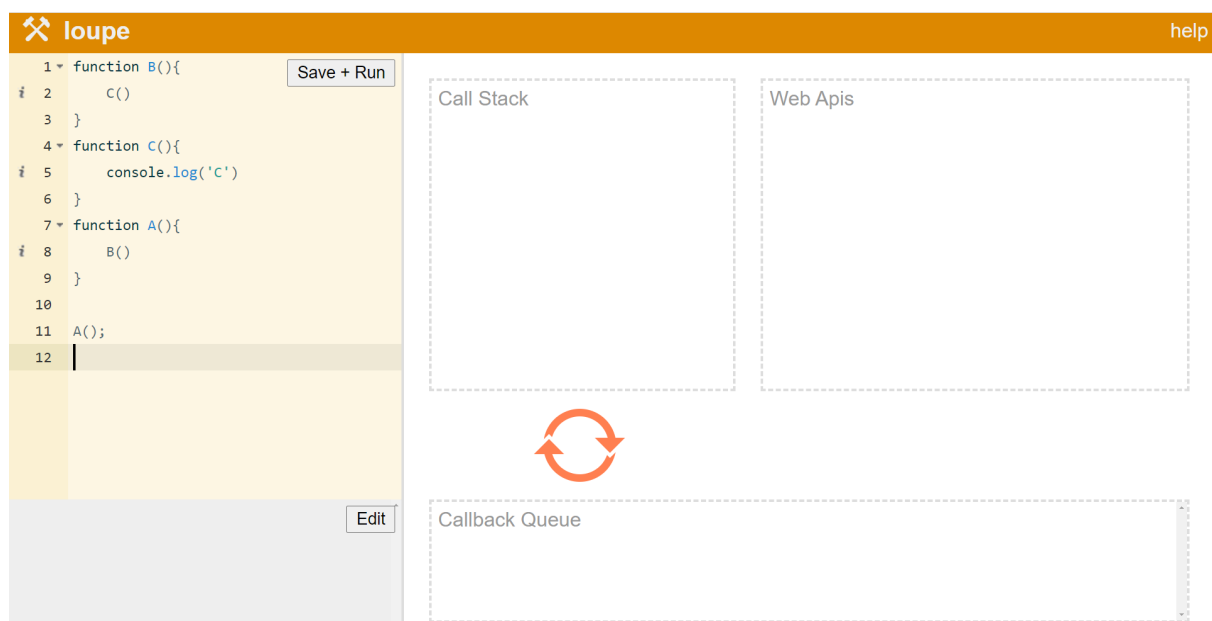


Imagen 5. Representación de event loop antes de ejecutar las funciones.
Fuente: Desafío Latam

- **Paso 3:** Sólo falta presionar “Save + Run” para iniciar el proceso y ver la respuesta que genera la página web para nuestro código. Al comenzar la ejecución de este script se agrega la función A al Call Stack como se muestra en la imagen 6.



Imagen 6. La función A se agrega a la pila de ejecución.
Fuente: Desafío Latam

- **Paso 4:** Como la función A está llamando a la función B, entonces se agrega B al Call Stack a continuación de A formando una pila de ejecución. La función A permanece en la pila porque todavía no finaliza, mientras que la función B se agrega sobre la función A en la pila. La pila en este caso tendrá una estructura del tipo LIFO (Last In, First Out) o lo que sería “último en entrar, primero en salir”. Eso se notará cuando lleguemos a la última función, donde podremos observar como la última función que parece en la pila será la primera en desaparecer.

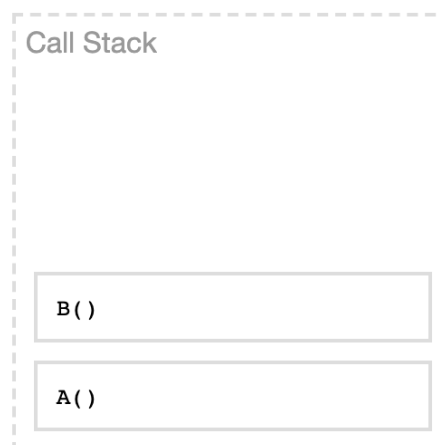


Imagen 7. Se agrega la función B a la pila.
Fuente: Desafío Latam

- **Paso 5:** Como la función B llama a la función C, entonces se agrega C a la pila de ejecución como se visualiza en la imagen número 8. Aquí vemos que continúan las funciones A y B, ya que no han finalizado.

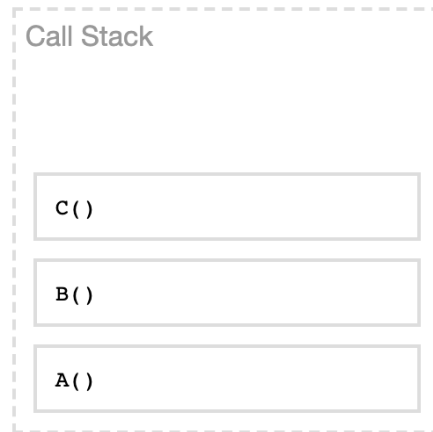


Imagen 8. Se agrega la función C a la pila.

Fuente: Desafío Latam

- **Paso 6:** En este punto se evalúa la sentencia de la función C agregando el `console.log` al stack y se procesa, quedando aún las funciones anteriores en la pila, es decir, las funciones A, B y C. Como se visualiza en la imagen 9.

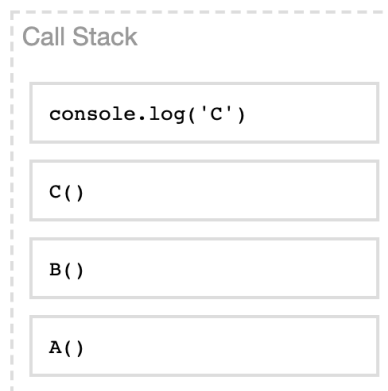


Imagen 9. Se ejecuta la función C agregando el `console.log` al stack.

Fuente: Desafío Latam

- **Paso 7:** Luego de procesar la sentencia del `console.log`, mostrando el literal "C" en la consola del navegador web, se elimina del stack quedando las funciones A, B y C aun en la pila. Como se visualiza en la imagen 10.

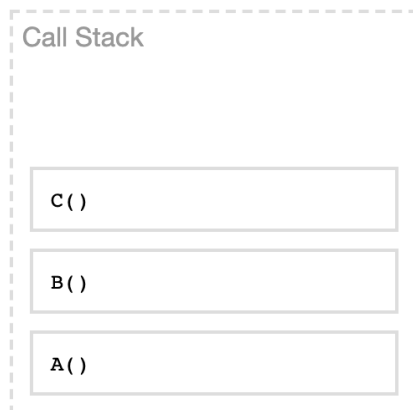


Imagen 10. Se procesa el `console.log` y se elimina del stack.

Fuente: Desafío Latam

- **Paso 8:** Una vez que termina la ejecución del `console.log` se elimina la función C del stack y retorna a la función de la que fue llamada, o sea a B, quedando aún la función A en la pila. Como se visualiza en la imagen 11.

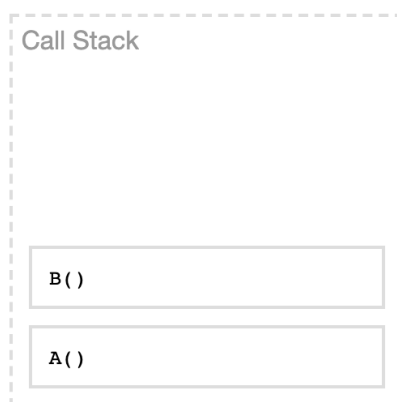


Imagen 11. La función C se elimina de la pila de ejecución.

Fuente: Desafío Latam

- **Paso 9:** Ahora la función B se elimina del stack y retorna a la función A. Siendo esta la primera función en entrar a la pila. Como se visualiza en la imagen 12.



Imagen 12. La función B se elimina de la pila de ejecución.
Fuente: Desafío Latam

- **Paso 10:** Como la función A ya no tiene nada más que realizar, entonces se elimina del stack y finaliza el proceso.

Si ponemos esta ejecución completa en una imagen, manteniendo el orden de ejecución de las funciones, se vería de la siguiente forma:

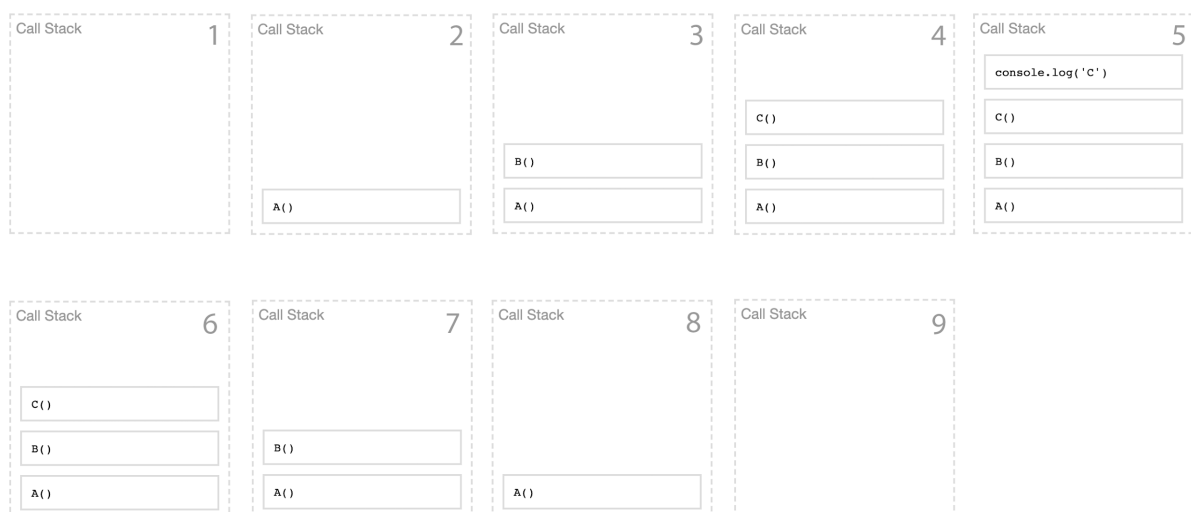


Imagen 13. Ejecución de la pila en orden.
Fuente: Desafío Latam

Como pudimos comprobar, existe un bloqueo en el stack porque se debe esperar a que una llamada termine para ejecutar la siguiente. Por ende, cuando desarrollamos algún módulo o aplicación, a veces creemos que nuestro código está perfecto, entonces vamos a ejecutarlo y nos encontramos con que pareciera que no hiciera nada, como si estuviera pegado y de repente vuelve a la normalidad. Bueno, es muy probable que justamente sea eso lo que está ocurriendo, el stack tiene demasiados procesos y debemos esperar a que se ejecuten todos para poder continuar navegando en nuestra aplicación. También ocurre en algunas ocasiones en que hay tantos procesos en el Call Stack y se siguen agregando más rápido de lo que se ejecutan algunas tareas, y esto provoca un error que nos indica que se sobrepasó la capacidad del Call Stack y por lo tanto la ejecución se rompe.

Entonces, ¿Cómo hacer que la ejecución de las funciones anteriores no sean bloqueantes para las otras funciones? La respuesta es usar funciones asíncronas y combinarlas con WEB APIs. Esto lo veremos más adelante, en el capítulo Promesas y Funciones Asíncronas.

Ejercicio propuesto (1)

Ejecuta el código mostrado a continuación en la página web [Loupe](#), y describe el comportamiento con tus propias palabras.

```
var A = [1, 2], B = [];  
  
for (var i = 0; i < A.length; i++) {  
  B.push(sumarDos(A[i]));  
};  
function sumarDos(x) {  
  return x + 2;  
};  
console.log(B)
```

Callbacks

En pocas palabras, un callback (llamada de vuelta) no es más que una función que se pasa como argumento de otra función, y que será invocada para completar algún tipo de acción.

Ahora bien, hablando desde el contexto asíncrono, un callback representa el: ¿Qué quieres hacer una vez que tu operación asíncrona termine? Por tanto, es el trozo de código que será ejecutado una vez que una operación asíncrona notifique que ha terminado. Esta ejecución se hará en algún momento futuro, gracias al mecanismo que implementa el bucle de eventos. Cuando llamamos o ejecutamos una función, normalmente esperamos a que termine de procesar todas las sentencias que tenga y nos retorne un resultado. Por ende, los callbacks se utilizan para esperar la ejecución de otros códigos antes de ejecutar el propio.

Sintaxis

```
function foo(callback) {  
  //hacer algo...  
  callback();  
}
```

Ejercicio guiado: Implementación de Callbacks

Crear una función que devuelva el nombre "John Doe", el cual tiene que mostrarse en la consola del navegador y luego de ser obtenido, se debe implementar funciones con callback. Realizar el ejercicio con ES6 y ES5 para visualizar mejor el callback y su funcionamiento. Sigamos los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo script.js. Luego, implementando ES6 realizar la función que retorne el nombre, esto lo podemos lograr creando primeramente una constante con el nombre getName para tener la función callback retornando el string "John Doe"..

```
const getName = callback => callback('John Doe')
```

- **Paso 2:** Llamar a la función y pasarle como argumento la función anónima con el argumento "name" que será la respuesta que se obtenga de la función getName, luego mostramos en consola el string.

```
const getName = callback => callback('John Doe')
getName(name => console.log(name)) // output: John Doe
```

- **Paso 3:** Al ejecutar el archivo script.js con ayuda de Node directamente desde la terminal con el comando: `node script.js`, el resultado encontrado seria:

```
John Doe
```

Veamos ahora el mismo ejemplo, pero escrito en ES5 para visualizar mejor el callback y su funcionamiento, partiendo de un nuevo archivo para no modificar el actual.

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo `script.js`. Luego, Implementando ES5 realicemos ahora la función con el nombre `getName`, la cual recibe un parámetro que será a su vez una función, y esta nueva función denominada “callback” llevará como argumento el string “John Doe”.

```
function getName(callback) {  
    callback('John Doe');  
}
```

- **Paso 2:** Posteriormente, se tiene que definir la función `callback` con el argumento “name”, que será la respuesta de la función “`getName`”, imprimiendo con un `console.log` el parámetro de la función, es decir, el string “John Doe”.

```
function callback(name){  
    console.log(name);  
}
```

- **Paso 3:** Luego, hacemos el llamado a la función “`getName`”, pasando como argumento la función “`callback`”.

```
getName(callback);
```

- **Paso 4:** Al ejecutar el archivo `script.js` con ayuda de Node directamente desde la terminal con el comando: `node script.js`, el resultado encontrado seria:

```
John Doe
```

Realicemos otro ejemplo, se solicita crear funciones con callbacks para obtener el cuadrado de un número. En este caso, utilizaremos la terminal y Node para ejecutar el archivo script creado. Por lo tanto:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo llamado `script.js`. Seguidamente, en el archivo creado en el paso anterior declaramos la función `getSquare`, a la cual se le pasan dos argumentos, uno denominado "number", quien será el valor del cual queremos obtener el cuadrado multiplicando el número por sí mismo, y otro denominado "callback" que será la función que retorna el cuadrado del número enviado.

```
const getSquare = (number, callback) => callback(number * number);
```

- **Paso 2:** Ahora hacemos el llamado a la función `getSquare`, pasando como argumento el número del cual calcularemos el cuadrado y el nombre que le daremos a la función callback, para recibir el valor y mostrarlo por la consola. Implementamos el llamado a la función varias veces para obtener distintos resultados.

```
const getSquare = (number, callback) => callback(number * number);  
getSquare(4, result => console.log(result));  
getSquare(2, result => console.log(result));  
getSquare(5, result => console.log(result));  
getSquare(12, result => console.log(result));
```

- **Paso 3:** Ejecutamos el archivo `script.js` en la terminal con ayuda de Node mediante la instrucción: `node script.js`, obteniendo como resultado:

```
16  
4  
25  
144
```

Ejercicio guiado: Callbacks anidados

Crear tres funciones que se utilizarán con un solo llamado, dos de ellas se ejecutarán dentro de la primera, y devolverán el resultado en la función callback. La primera función debe retornar el doble del argumento pasado, mientras que la segunda función debe retornar el cuadrado del argumento y por último la tercera función debe retornar el 25% del número. Para ello, ejecutamos los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crear un archivo con el nombre de script.js. Ahora, en el archivo script.js comenzamos por definir las funciones principal, multiply y calculate25Percent como constantes. La función principal multiplica el número pasado por argumento por dos (2). Mientras que la función multiply recibe como argumento el resultado de la multiplicación y la función calculate25Percent le pasamos por argumento el resultado de multiply.

```
const principal = (number) => {  
  const double = number * 2;  
}  
const multiply = (number, callback) => callback();  
const calculate25Percent = (number, callback) => callback();
```

- **Paso 2:** En la función principal, se debe ejecutar el resto del proceso, el cual corresponde a las llamadas de las otras dos funciones y mostrar el resultado en un console.log en el último llamado a la función, es decir, a la función calculate25Percent.

```
const principal = (number) => {  
  const double = number * 2;  
  // llamamos a multiply y le pasamos el resultado anterior  
  multiply(double, square => {  
    // llamamos a calculate25Percent y le pasamos el resultado de B  
    calculate25Percent(square, percent => {  
      // enviamos a la consola el resultado obtenido de calculate25Percent  
      console.log(percent)  
    })  
  })  
}  
  
const multiply = (number, callback) => callback();  
const calculate25Percent = (number, callback) => callback();
```


- **Paso 3:** La función `multiply` recibe como argumento el resultado de la multiplicación y a su vez se encarga de devolver el cuadrado del número recibido, mientras que la función `calculate25Percent`, a la que pasamos por argumento el resultado de `multiply`, devuelve el 25% del resultado recibido. Finalmente hacemos el llamado de la función principal enviando el número 5.

```
const principal = (number) => {  
  const double = number * 2  
  multiply(double, square => {  
    calculate25Percent(square, percent => {  
      console.log(percent)  
    })  
  })  
}  
  
const multiply = (number, callback) => callback(number * number);  
const calculate25Percent = (number, callback) => callback(number * 25 /  
100);  
principal(5);
```

- **Paso 4:** Ejecutamos el archivo `script.js` en la terminal con ayuda de Node mediante la instrucción: `node script.js`, obteniendo como resultado:

25

Ahora, en pocas palabras al ejecutar el código anterior enviando como parámetro el número cinco (5), lo que ocurrió fue: comienza la ejecución de la función principal y se le pasa como argumento el número 5; esta función multiplica el $5 * 2$ obteniendo 10. Luego, la función principal ejecuta la función `multiply` pasándole como argumento este resultado y que multiplica por sí mismo y devuelve como resultado 100. Posteriormente, terminada la ejecución de `multiply` pasa a ejecutar `calculate25Percent` enviándole el argumento 100 que obtuvo de `multiply`, y en el que se obtiene el 25% de éste, o sea, el 25% de 100 y nos da como resultado 25 que es lo que se mostraría en el `console.log(percent)`.

Ejercicio propuesto (2)

Desarrollar un programa con JavaScript implementando funciones con callback, que al pasarle los puntos obtenidos en dos ejercicios de un examen, sume ambos puntos e indique si hemos superado la prueba. El examen tiene una ponderación total de 10 puntos. Cada ejercicio tiene una nota máxima de 5 puntos.

Race condition

Condición de carrera (como indica su nombre en inglés) se interpreta como la ejecución de varios procesos o eventos a la vez modificando datos de forma concurrente, sin tener la certeza sobre cuáles serán los valores finales retornados por estos procesos, lo que puede producir errores o inconsistencia en los resultados esperados.

```
let value = 0;
function add1(callback) {
  callback(value += 1);
}
function add2(callback) {
  callback(value += 2);
}
```

Aquí tenemos una variable global que es modificada por dos funciones, y en el caso que estas dos (2) funciones se ejecutarán de forma asíncrona, no tendríamos la certeza de cuál se procesaría primero, y esto impacta sobre el resultado entregado. Si llega primero el resultado de la función add1 y después el de add2, los cambios del resultado sería el siguiente:

1. value es 0.
2. se ejecuta add1, resuelve la suma y devuelve 1.
3. ejecuta add2, resuelve la suma y devuelve 3.

Cuando escribimos nuestros métodos o funciones, esperamos que se ejecuten y retornen lo que esperamos, comprobemos todo esto.

Las condiciones de carrera son difíciles de identificar y depurar, ya que en ocasiones un código puede devolver el mismo resultado muchas veces durante los tests, pero en algunos, devolver un resultado distinto.

Ejercicio guiado: Demostración de Race Condition

Demostrar que en ocasiones los resultados pueden variar cuando el tiempo de ejecución influye en lo que queremos obtener. Para ello, implementaremos la función `setTimeout` (que veremos más adelante), para lograr que varias funciones se ejecuten asincrónicamente, estableciendo un tiempo aleatorio en el que se tengan que ejecutar las funciones para demostrar la asincronía y lo que puede ocurrir en el resultado entregado.

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo `script.js`. Luego, sobre el mismo archivo e implementando ES6, realizar la función que retorne el valor aleatorio entre dos números, esto lo podemos lograr creando una constante con la función llamada `randomNumber`, recibiendo dos parámetros, el número mínimo y máximo, luego utilizando el objeto `Math.random()` encontramos el número aleatorio. Este número aleatorio será el tiempo en que tardará en ejecutarse las posteriores.

```
const randomNumber = (min, max) => {  
  return parseInt(Math.random() * (max - min) + min);  
}
```

- **Paso 2:** Ahora trabajemos con una función que se encargará de ejecutar de forma asíncrona una impresión por consola, implementando el método `setTimeout()`, de acuerdo al número aleatorio recibido como parámetro.

```
const runner = (time) => {  
  setTimeout(() => {  
    console.log(`Hola, Soy el timer de ${time}ms`)  
  }, time)  
}
```

- **Paso 3:** Solo queda por hacer de esta función varias veces, por consiguiente, a cada función le pasamos un valor aleatorio obtenido al ejecutar la función `randomNumber`.

```
runner(randomNumber(500, 5000));  
runner(randomNumber(500, 5000));  
runner(randomNumber(500, 5000));  
runner(randomNumber(500, 5000));
```

- **Paso 4:** Como estamos indicando que el tiempo de ejecución sea aleatorio entre 500 ms y 5 segundos, no tenemos la certeza de cual se ejecutará primero y cuál después. Por ende, para poder revisar el comportamiento y respuesta del código, ejecuta el archivo `script.js` desde la terminal con la ayuda de Node.

```
Hola, Soy el timer de 1430ms  
Hola, Soy el timer de 3079ms  
Hola, Soy el timer de 4205ms  
Hola, Soy el timer de 4390ms
```

Al ejecutar el código tres veces, se puede observar como las respuestas varían en cada llamado de la función y en cada ejecución del código. Este es un ejemplo muy simple, pero cumple con la demostración y nos lleva a pensar por qué es tan importante conocer lo que estamos desarrollando y los flujos que deben tener nuestras aplicaciones.

¿Cómo se puede evitar la condición de carrera?

1. Evitar utilizar recursos compartidos, como variables que se modifican en más de una función.
2. Tener cuidado con el alcance de las variables.
3. Utilizar correctamente la secuencia de instrucciones y verificar que el código se ejecuta de manera asíncrona, pero obteniendo los resultados de forma secuencial.
4. Nunca asumir que si se espera una cantidad de tiempo, el código se ejecutará en orden.

Ejercicio propuesto (3)

Crear la función "separar", donde se pasen dos (2) argumentos, un arreglo de números y un callback. La función deberá devolver un objeto con dos (2) arreglos, uno con los pares y otro con los impares. Ejemplo: Si se tiene el arreglo [3,12,7,1,2,9,18]. La función debe retornar: pares: [12,2,18], impares: [3,7,1,9].

Ejercicio propuesto (4)

Crear cuatro (4) funciones con las operaciones básicas matemáticas, sumar, restar, multiplicar y dividir, a las cuales se les debe enviar dos valores, el primer valor siempre será más grande que el segundo. Este segundo valor, no puede ser 0. Los valores deben ser devueltos en callbacks.

Callbacks y Promesas

Competencias

- Codificar una función asíncrona para utilizar callbacks y setTimeout.
- Codificar una función asíncrona que permita implementar Promise / Resolve.

Introducción

Para desarrollar webs de calidad estamos obligados a generar código asíncrono para no hacer que otras porciones de código tengan que esperar para ejecutarse, y para ello contamos con una función API que nos proporciona el navegador y que no es bloqueante para el resto de funcionalidades de nuestras aplicaciones, la usaremos bastante en nuestros desarrollos, la cual lleva por nombre: [setTimeout](#). Adicionalmente al setTimeout, se han integrado nuevas funcionalidades al lenguaje JavaScript y con ello, la solución a problemas de asincronía y funciones anidadas. Hablamos de Promesas (Promise), la que nos permiten ejecutar código asíncrono y devolver una promesa, que finalizará resuelta o rechazada.

Por lo tanto, a lo largo de este capítulo aprenderemos cómo crear código de manera asíncrona implementando Promesas, logrando que el código escrito sea más ordenado y potente, resolviendo a la vez algunos problemas que generan los Callbacks mediante la adecuada implementación de las promesas.

setTimeout

El método `setTimeout()` llama a una función o evalúa una expresión después de un número específico de milisegundos. Así mismo, pertenece a una de las funciones Web APIs que tienen los navegadores y que se agrega directamente en el Call Stack.

Este método tiene doble funcionalidad:

1. El callback pasado como primer argumento se ejecutará después del tiempo establecido en el segundo argumento.
2. Su ejecución no bloquea el stack, por lo que es una función que se procesa de forma asíncrona.

Veamos la sintaxis y algunos ejemplos de implementación de `setTimeout` y cómo se comporta.

Sintaxis:

```
setTimeout(function(){}, milliseconds, param1, param2, ...)
```

El único argumento requerido es "function" y es lo que se va a ejecutar o será la expresión que se evaluará al cumplirse el tiempo, que es lo que establece el argumento *milliseconds* para indicar cuánto deberá esperar *function* para ejecutarse, este valor es opcional. Los argumentos siguientes son parámetros adicionales que se pueden pasar a *function* y también son opcionales.

Hay un caso especial para *milliseconds* y es poner este valor en cero (0) u omitirlo y lo mostraremos en un ejemplo más adelante.

Implementación de setTimeout

El primer ejemplo consiste en esperar una cantidad de tiempo para evaluar la expresión que contiene la función que se le pasó por argumento, que en este caso es mostrar en la consola un mensaje.

```
setTimeout(() => {  
    console.log('hola mundo!');  
}, 1000)
```

1. Establecemos qué será lo que ejecutará la función que le pasamos a setTimeout.
2. Le indicamos el tiempo que demorará en realizar la operación que le indicamos.
3. Luego de 1000ms, se ejecuta el código y aparece el log en la consola del navegador.




Imagen 14. Respuesta de la consola del navegador.
Fuente: Desafío Latam

En el siguiente ejemplo le indicaremos distintos tiempos de espera y veremos el resultado ordenado de forma correcta a pesar de estar escrito en desorden, ya que lo importante es el tiempo en que se ejecutan y no el orden en que se escriban. Esto es justamente lo que se demuestra en este capítulo, la asincronía. Por consiguiente, para realizar este ejemplo, se deben seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crear un archivo con el nombre de script.js. Seguidamente en el archivo script.js, se implementan las funciones con el método setTimeout, utilizando distintos tiempos de ejecución y distintos mensajes dentro de cada función. Por ejemplo, el primer setTimeout se ejecutará después de dos (2) segundos, mostrando el mensaje 'después de 2 segundos!', el segundo setTimeout se ejecutará después de tres (3) segundos mostrando el mensaje 'después de 3 segundos!' y el último setTimeout se ejecutará después de un (1) segundo mostrando el mensaje 'después de 1 segundos!'.

```
setTimeout(() => {  
    console.log('después de 2 segundos!')  
}, 2000);  
setTimeout(() => {  
    console.log('después de 3 segundos!')  
}, 3000);  
setTimeout(() => {  
    console.log('después de 1 segundos!')  
}, 1000);
```

- **Paso 2:** Ejecutamos ahora el archivo script.js desde la terminal con ayuda de Node, para ver el resultado, por lo que podremos observar como al pasar los segundos se van mostrando los mensajes uno por uno.

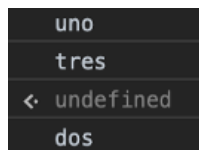
```
después de 1 segundos!  
después de 2 segundos!  
después de 3 segundos!
```

Para este ejemplo utilizaremos console.log y setTimeout para ver el orden de ejecución de las sentencias. Ahora probemos con otro ejemplo directamente en la consola del navegador web:

```
console.log('uno');  
setTimeout(() => {  
    console.log('dos');  
}, 1000);  
console.log('tres');
```

1. Se agrega un console.log.
2. Se agrega un setTimeout con una sentencia en el callback y 1000ms.
3. Se agrega otro console.log.

Al observar el resultado de la ejecución de este código, se ve que “dos” aparece al final de la ejecución, ya que para mostrarse debía esperar un segundo.



```
uno
tres
< undefined
dos
```

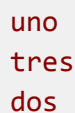
Imagen 15. Resultado de la consola del navegador.

Fuente: Desafío Latam

Al observar el siguiente ejemplo, es el mismo que el anterior, sólo que ha cambiado el tiempo a 0, ¿Cómo será el resultado de este código? Ejecutemos en la consola del navegador este script.

```
console.log('uno');
setTimeout(() => {
  console.log('dos');
}, 0);
console.log('tres');
```

Pensemos primero en que un tiempo de cero (0) significa que no hay retraso en la ejecución, por lo tanto podríamos asumir que el resultado será “uno, dos, tres”. Veamos la consola.



```
uno
tres
dos
```

Al ejecutarlo en la consola nos da como resultado “uno, tres, dos”. ¿Cómo es posible esto, si el tiempo que tiene asignado es 0, por qué se ejecuta en un tiempo distinto? Para ver una explicación más detallada, puedes ir al archivo ubicado en “Material Complementario” con el nombre de: **Material Apoyo Lectura - Funcionamiento de Event Loop con setTimeout**, y observar en detalle cómo se ejecuta desde la raíz de JavaScript el código anterior.

Ejercicio guiado: Obteniendo usuarios por ID

Simular una petición asíncrona como si se tratara de una API, retornando la información de un usuario pero en formato JSON. La información para el primer usuario será la siguiente: `data = {id: 1, name: 'John', lastName: 'Doe', age: 24}`. Mientras que la información para el segundo usuario será: `data = {id: 2, name: 'Jane', lastName: 'Smith', age: 19}`.

Por lo tanto, se debe enviar el parámetro “id” a la función para simular que se requieren los datos de un usuario en específico y que el tiempo de respuesta será de 1000 milisegundos.

Iniciemos a desarrollar este ejemplo siguiendo los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un `script.js`. Luego, en el archivo `script.js` se creará primeramente la función denominada `getUserData`, la cual recibirá dos parámetros, el número de “id” y la función “callback”. Dentro de esta función, se utilizará el método `setTimeout()` con un tiempo de 1000 milisegundos.

```
function getUserData(id, callback){  
  setTimeout(() => {//código aquí}, 1000);  
};
```

- **Paso 2:** Ahora dentro del `setTimeout`, creamos la variable que contendrá los datos solicitados dependiendo del número del “id”. Igualmente, con la función `callback` se retornará los datos del usuario.

```
function getUserData(id, callback){
  setTimeout(() => {
    let data = {};
    if(id === 1){
      data = {
        id: 1,
        name: 'John',
        lastName: 'Doe',
        age: 24
      }
    }
    if(id === 2){
      data = {
        id: 2,
        name: 'Jane',
        lastName: 'Smith',
        age: 19
      }
    }
    callback(data)
  }, 1000)
}
```

- **Paso 3:** Hacer el llamado a la función dos veces, pasando dos "id" distintos y el nombre del callback con el cual queremos mostrar el resultado.

```
getUserData(1, user => console.log(user));
getUserData(2, user => console.log(user));
```

- **Paso 4:** Ejecutar ahora el archivo script.js desde la terminal con ayuda de Node, para ver el resultado, por lo que podremos observar cómo al pasar los segundos se van mostrando los mensajes uno por uno.

```
{ id: 1, name: 'John', lastName: 'Doe', age: 24 }  
{ id: 2, name: 'Jane', lastName: 'Smith', age: 19 }
```

Como se puede ver, al pedir la información del usuario 1, la función entrega los datos (objeto) de John Doe y cuando se pide la del usuario 2, retorna los datos de Jane Smith.

Ejercicio propuesto (5)

Crear un programa con JavaScript mediante el uso de funciones con Callback y el método `setTimeout`, que permita mostrar los datos de un usuario de acuerdo al nombre o el número de identificación. Los datos serán los siguientes:

```
usuario1 = {id: 2356256, name: 'Juan', lastName: 'Duran', age: 35}  
usuario2 = {id: 27564512, name: 'Manuel', lastName: 'Perez', age: 31}  
usuario3 = {id: 17658624, name: 'Jocelyn', lastName: 'Rodriguez', age: 30}  
usuario4 = {id: 12345678, name: 'Maria', lastName: 'Garrido', age: 30}
```

Promesas (Promise)

Una promesa es un objeto que representa el estado de una operación asíncrona. Estos estados son:

- **pendiente** - cuando todavía no empieza o no ha terminado de ejecutarse.
- **resuelto** - en caso de éxito.
- **rechazado** - en caso de fallo.

El resultado de una promesa podría estar disponible **ahora** o en el **futuro**. Por otra parte, las promesas permiten ejecutar código asíncrono y devolver valores como si fueran secuenciales, ya que permiten continuar con nuevas sentencias una vez resuelta la promesa. Lo que retorna una promesa es otra promesa por lo que permite que se pueda continuar creando sentencias o expresiones utilizando el método **then** para cuando la promesa anterior sea resuelta y el método **catch** cuando la promesa sea rechazada.

Sintaxis

```
new Promise((resolve, reject) => { /* ..... */ })
```

Se requiere pasar como argumento una función callback que contenga dos (2) parámetros, puedes nombrarlos como quieras, en este caso se usarán `resolve` y `reject` que son los valores por defecto. Como se muestra a continuación:

```
const promise = new Promise((resolve, reject) => {  
  const value = true;  
  value ? resolve('Exito') : reject('Rechazado')  
})  
promise.then(resp => console.log(resp)) // output: Exito
```

Ejecutemos directamente el código anterior en la consola del navegador.

```
Exito VM21729:6
< ▶ Promise {<resolved>: undefined}
```

Imagen 16. Resultado de la consola del navegador.

Fuente: Desafío Latam

Vemos que el resultado de la promesa se resolvió y devolvió "Exito". Ahora veamos qué pasa en caso contrario. Es decir, si modificamos la variable "value" por *false*.

```
< ▶ Promise {<rejected>: "Rechazado"}
✖ ▶ Uncaught (in promise) Rechazado local-ntp.html:1
```

Imagen 17. Resultado de la consola del navegador.

Fuente: Desafío Latam

Aquí la promesa fue rechazada por lo que devolvió "Rechazado", pero adicionalmente a esto, se generó un error no controlado debido a que en nuestro código no está manejado este tipo de situaciones, por lo que veremos cómo hacerlo en el capítulo Manejo de errores de la Lectura - Callbacks y APIs (Parte II). Para mayor información también puedes revisar la siguiente [documentación](#).

Lo visto hasta el momento, es solo la sintaxis de Promise y cómo funciona de manera general, pero en el punto a continuación, si se trabajan con las promesas de forma completa, explicada paso a paso y con ejemplos de implementación reales.

Promise.all

Permite poder ejecutar un objeto iterable o arreglo de múltiples promesas, y esperar que estas se resuelvan para entregar todos los resultados de una vez. Lo que retorna es un arreglo con tantos elementos que se le hayan pasado al método *all* en el objeto iterable. Como le indicamos en el arreglo una cantidad de promesas a evaluar, todas se deben resolver para que la promesa se cumpla. Al fallar una o más promesas que se enviaron en el objeto iterable, se rechazará el Promise.all.

Sintaxis

```
Promise.all(objetoIterable)
  .then((result) => { /* ... */ })
  .catch((error) => { /* ... */ })
```

Ejercicio guiado: Promise.all

Aplicando la sintaxis anterior, se deben cumplir todas las promesas iniciadas en variables separadas, la primera se debe igualar a un número, la segunda se debe igualar a `Promise.resolve(2)`, y la tercera a una nueva promesa que retorne el número tres cuando se resuelva.

Implementar la sentencia del `Promise.all` y ver cómo se deben cumplir todas las promesas para que se retorne un arreglo con los resultados. Por consiguiente, todas las promesas se deben enviar al método “all”, y cuando estas se cumplan, se debe mostrar por terminal el resultado.

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un `script.js`. Seguidamente, en el archivo `script.js` vamos a crear primeramente las tres constantes que tendrán las promesas por individual.

```
const promise1 = 1;  
const promise2 = Promise.resolve(2);  
const promise3 = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve(3)  
  }, 1000);  
});
```

- **Paso 2:** Luego, utilizando el `Promise.all`, pasamos como arreglo las tres promesas por individual, seguidamente cuando ya se cumplan todas las promesas (then) se muestra la respuesta recibida mediante un `console.log`.

```
Promise.all([promise1, promise2, promise3]).then(response => {  
  console.log(response); // [1, 2, 3]  
});
```

- **Paso 3:** Ejecutamos ahora el archivo script.js desde la terminal con ayuda de Node, para ver el resultado:

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

Pero, ¿qué fue lo que hizo el código cuando se ejecutó en la terminal: la primera promesa sólo devuelve un uno (1). Mientras que la segunda promesa retorna una promesa resuelta con valor de respuesta dos (2). Posteriormente, la tercera promesa retorna una promesa resuelta, pero después de 1 segundo. Luego de finalizar su ejecución, todas las promesas se resuelven y retornan a `Promise.all` con la respuesta de éstas, devolviendo un arreglo con las tres respuestas.

Pero no todo es tan preciso y exacto, debido a que pueden ocurrir problemas o errores en la promesa generando errores. Esto es el caso contrario al **then**, es decir, cuando una o más promesas se rechace. Esta parte será abordada en la Lectura - Callbacks y APIs (Parte II), Capítulo: Manejo de Errores - Método Catch.

Ejercicio guiado: Combinando promesas y eventos en el DOM

Solicitan que al hacer un clic sobre un botón, se ejecute una promesa que retorne y muestre por pantalla tres mensajes:

- Primero: "Solicitando Autorización"
- Segundo: "Esperando la información"
- Tercero: "El usuario en línea es: Juan".

Estos mensajes se deben insertar en el documento utilizando el DOM dentro de una función externa.

Para generar el primer mensaje se debe utilizar una función que contenga una promesa y tarde 2.5 segundos en indicar que todo está correcto, es decir, que tiene autorización retornando "true", mientras que el segundo y tercer mensaje se origina en otra función que contiene una promesa que muestra primeramente el segundo mensaje y al cabo de 2,5 segundos muestra el tercer y último mensaje.

Ahora bien, las promesas se deben activar cuando el usuario haga un click sobre el botón denominado "Ejecutar", y mientras la respuesta sea verdadera "true" se deben mostrar los dos últimos mensajes. Para ello, ejecutamos los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un index.html y un script.js. Seguidamente, en el index.html debes escribir la estructura básica de un documento HTML como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Promesas</title>
</head>
<body>
  <h4>Usando ES6 - Promesas</h4>
  <section id="contenido"></section>
  <button id="boton">Ejecutar</button>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 2:** En el archivo script.js, lo primero es agregar un escucha al botón, para que cuando el usuario haga un click sobre él, se activen los llamados a la funciones que contienen las promesas. La primera función que debe llamar será la que retorna el mensaje de autorización con el nombre de `getData`, es decir, si el usuario está o no autorizado, en este caso y para este ejemplo, el retorno deberá ser "true". A ella, se le concatena un "then" para esperar la respuesta de la promesa, si la promesa retorna true, entonces se debe regresar el llamado a la función que permita mostrar los datos (mensajes dos y tres), esta función llevará por nombre `mostrarData()`. Ahora, esta última función también contiene una promesa, por lo tanto se debe concatenar el "then" para así poder mandar a mostrar por pantalla el último mensaje con el nombre del usuario.

```
boton.addEventListener('click',()=>{
  getData().then(autorizacion => {
    if(autorizacion){
      return mostrarData();
    };
  }).then(usuario => {
    setTexto(`El usuario en línea es: ${usuario.nombre}`);
  });
});
```

- **Paso 3:** Agregamos ahora la lógica interna de la función `getData()`. Ya que es la primera en ser invocada. En esta función se retorna una nueva promesa y dentro de la promesa se debe llamar a la función encargada de mostrar el contenido en el documento mediante el DOM `'Solicitando Autorización'`. Luego de transcurrir los 2.5 segundos del método `setTimeout`, se resuelve con true la promesa.

```
const getData = () => {
  return new Promise((resolve,reject)=>{
    setTexto('Solicitando Autorización');
    setTimeout(()=>{
      resolve(true);
    },2500);
  });
};
```

- **Paso 4:** Debemos trabajar ahora con la segunda función, esta función es convocada si y sólo si la primera promesa retorna "true". Dentro de ella se retorna una nueva promesa, en donde se llamará nuevamente a la función `setTexto` para que muestre el mensaje 'Esperando la información', luego de transcurrir 2.5 segundos se debe resolver la promesa con el dato: `{nombre:"El usuario en línea es: Juan"}`. Que será parte del último mensaje a mostrar por pantalla llamando a la función `setTexto` pero al retornar la promesa.

```
const mostrarData = () => {
  return new Promise((resolve, reject) => {
    setTexto('Esperando la información');
    setTimeout(() => {
      resolve({nombre: "Juan"});
    }, 2500);
  });
};
```

- **Paso 5:** Ya solo queda crear la función que incrustará los mensajes en el documento, específicamente en el elemento con el "id" igual a "contenido", mediante la propiedad `textContent` le igualamos el valor que traen el parámetro.

```
const setTexto = datos => {
  contenido.textContent = datos;
};
```

- **Paso 6:** Finalmente, ejecutamos el archivo index.html con nuestro navegador web, y hacemos un click sobre el botón para obtener la secuencia de resultados como se muestra en la imagen:

Usando ES6 - Promesas <input type="button" value="Ejecutar"/>	Usando ES6 - Promesas Solicitando Autorización <input type="button" value="Ejecutar"/>
Usando ES6 - Promesas Esperando la información <input type="button" value="Ejecutar"/>	Usando ES6 - Promesas El usuario en linea es: Juan <input type="button" value="Ejecutar"/>

Imagen 18. Secuencia de respuestas en el documento web.
Fuente: Desafío Latam

Ejercicio propuesto (6)

Crear un programa con JavaScript utilizando promesas que calcule un número aleatorio entre el "1" y el "100", pero que muestre el número aleatorio si y sólo si este número está comprendido entre "20" y "60", ambos valores incluidos.

Promise.race

Promise.race es bien parecido a Promise.all, sólo que para esta función basta con que una de las promesas esté resuelta para retornar el resultado de la promesa ganadora.

Ejercicio guiado: Promise.race

Realizar un ejemplo donde existan tres promesas, cada una regresará un valor distinto (1, 2 y 3 respectivamente), de acuerdo a la ejecución del tiempo que tenga cada una.

Ese tiempo debe originarse en una función externa, la cual va a retornar un número aleatorio que servirá como tiempo para los setTimeout de cada promesa, dando un valor aleatorio a cada una de ellas.

Para realizar este ejemplo, se deben seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo script.js. Luego, en el archivo script.js, creamos primeramente la función llamada `randomNumber` que retorna el número aleatorio, recibiendo dos valores (uno mínimo y uno máximo), para ayudar al método `Math.random()` a generar el número entre la diferencia de los dos números recibidos por la función.

```
const randomNumber = (min, max) => {  
  return parseInt(Math.random() * (max - min) + min);  
}
```

- **Paso 2:** Crear la primera promesa que posteriormente replicamos cambiando solo algunos valores. Esta primera promesa se construirá sobre una constante y dentro de ella se utilizará el método `setTimeout` para resolver la promesa, enviando el número 1 en este caso, es importante indicar que el valor del tiempo del `setTimeout` se solicita a la función creada anteriormente, denominada `randomNumber`, pasando el valor para el mínimo y el máximo.

```
const promise1 = new Promise((resolve, reject) => {  
  setTimeout(() => { resolve(1) }, randomNumber(500, 2000));  
});
```

- **Paso 3:** Copiar la primera promesa creada y le cambiamos los valores de las constantes, es decir de la variable, y el valor que resuelve para cumplir la promesa. En este caso, como el primer valor que regresa la promesa es uno (1), entonces en estas dos promesas se regresará el dos (2) y el tres (3) respectivamente.

```
const promise2 = new Promise((resolve, reject) => {  
  setTimeout(() => { resolve(2) }, randomNumber(500, 2000));  
});  
const promise3 = new Promise((resolve, reject) => {  
  setTimeout(() => { resolve(3) }, randomNumber(500, 2000));  
});
```

- **Paso 4:** Ya solo queda hacer el llamado implementando el `Promise.race` y pasarle como argumento las tres promesas creadas para que retorne la promesa ganadora y entonces (then) se muestre por la terminal el resultado.

```
Promise.race([promise1, promise2, promise3]).then(response => {  
  console.log(response);  
});
```

- **Paso 5:** Ejecutamos ahora el archivo `script.js` desde la terminal con ayuda de Node con el siguiente comando `node script.js`, para ver el resultado:

2

Ejercicio guiado: Promise.race, la respuesta más rápida

En un colegio, la profesora decide dar incentivos a quien responda primero las preguntas que hace. El que lo haga obtendrá décimas para la próxima evaluación. Selecciona a Carlos, María y Cristian para que respondan. Los estudiantes piensan y dan su respuesta casi al mismo tiempo, ¿quién da la respuesta más rápido?

Para realizar este ejemplo, se deben seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo `script.js`. Luego, en el archivo `script.js`, creamos primeramente la función llamada `randomNumber` que retorna el número aleatorio, recibiendo dos valores (uno mínimo y uno máximo), para ayudar al método `Math.random()` a generar el número entre la diferencia de los dos números recibidos por la función. Por consiguiente, usaremos esta función como tiempo en que se demoran los niños en responder.

```
const randomNumber = (min, max) => {  
  return parseInt(Math.random() * (max - min) + min);  
}
```

- **Paso 2:** Definir la función `responder`, que se encarga de esperar el tiempo aleatorio que tardan en responder y devuelve el nombre del niño que respondió. El tiempo será establecido entre 500 y 700 ms.

```
const responder = alumno => new Promise((resolve, reject) => {  
  setTimeout(() => { resolve(alumno) }, randomNumber(500,700))  
}))
```

- **Paso 3:** Definir una nueva función, la cual, tendrá el `Promise.race` y se encargará de mostrar la promesa que retorne más rápido una respuesta, indicando el mensaje en la consola del navegador web: `'La estrella es para:'`.

```
const pregunta = (alumnos) => {  
  Promise.race(alumnos)  
    .then(response => console.log('La estrella es para:', response))  
}
```

- **Paso 4:** Ya solo queda iniciar una variable por cada alumno llamando a la función `responder` creada anteriormente y pasando como parámetro el nombre del alumno.

```
const Carlos = responder('Carlos')
const Maria = responder('Maria')
const Cristian = responder('Cristian')
const alumnos = [Carlos, Maria, Cristian];
pregunta(alumnos)
```

- **Paso 5:** Al final ejecutamos las promesas de forma simultánea y esperamos a que llegue la respuesta con el niño que respondió más rápido.

La estrella es para: Cristian

En este caso el alumno que respondió más rápido fue Cristian.

Ejercicio propuesto (7)

Realizar un programa en JavaScript que calcule la suma o la resta o la multiplicación de dos números. Para ello se debe implementar una promesa por cada operación matemática y solo se debe mostrar el resultado de la promesa ganadora. Utiliza tiempos aleatorios para los `setTimeout` que poseen cada promesa con su operación matemática. Igualmente implementa `Promise.race` para mostrar sola la primera promesa que retorne el resultado de la operación matemática ganadora.

Async/Await en JavaScript

Competencias

- Codificar una función asíncrona utilizando ASYNC/AWAIT para obtener una respuesta directa y no una promesa.
- Identificar las restricciones de Async / Await en ES6 para evitar errores de implementación en el código.

Introducción

Hasta el momento, logramos trabajar con código de manera asíncrona implementando las Promesas, pero, como las promesas devuelven promesas y no valores, es necesario mejorar aún más el lenguaje para que cuando ejecutemos un código asíncrono, retorne el valor de lo solicitado sin tener que verificar si fue resuelta la promesa o no. Para esto, aprenderemos el uso de funciones asíncronas (async) que una vez declaradas, siempre devolverán promesas y que en conjunto con await, podamos recibir sólo los resultados. Siendo este otro tema muy importante que aprenderás a lo largo de la unidad y tendrá mucha utilidad para el desarrollo de tus proyectos desde el frontend o el backend, dando un mejor orden y por lo tanto lectura al código, facilitando el mantenimiento y futuras expansiones o mejoras.

Async / Await

Ya vimos que las promesas pueden manejar código asíncrono y se comportan como si fueran síncronos, además sabemos que devuelven una promesa cuya respuesta no es conocida hasta que se resuelve o se rechaza. Por ende, lo que viene a resolver Async/Await es permitir que las funciones que retornan promesas, devuelvan directamente los resultados en vez de promesas.

Sintaxis Async

```
async function name([param[, param[, ... param]]]) { /* ... */ }
```

Se utiliza la palabra clave **async** antes de la declaración de una función, lo que nos indica que siempre retornará una promesa.

```
async function f() {  
    return 1;  
}  
f().then(resp => console.log(resp));
```

En la función asíncrona anterior, vemos que se devuelve un valor, pero como está declarada como asíncrona retorna una promesa y por eso es posible continuar el llamado con el método **.then** al que se le pasa un callback que contiene la respuesta de la función.

```
1  
Promise { <state>: "pending" }
```

Cuando queremos utilizar las funciones para pedir datos externos a través de requests, necesitamos que se ejecute esa sentencia antes de devolver la respuesta, por lo que utilizaremos la palabra reservada **await** para esperar y retornar la promesa. El operador await solo trabaja dentro de una función async.

Sintaxis Await

```
let value = await promise;
```

Un ejemplo de la función asíncrona en conjunto con el await sería de la siguiente forma:

```
const getExternalData = () => {  
  return new Promise((resolve) => {  
    setTimeout(() => { resolve('hola mundo') }, 2000)  
  })  
}  
const getData = async () => {  
  const resp = await getExternalData();  
  console.log(resp)  
}  
getData();
```

Al declarar la función `getData` como asíncrona, ya podemos continuar después de obtener el resultado que trae los datos requeridos. Por ende, al ejecutar directamente el código anterior en la consola del navegador web, el resultado sería:

```
Promise { <state>: "pending" }  
hola mundo
```

Veamos un ejemplo obteniendo datos externos, por lo cual, implementaremos el método [fetch](#), quien proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas de una manera fácil y lógica de obtener recursos de forma asíncrona.

Ejercicio guiado: Controlando la asincronía

Crear un programa que consulte una URL que entregue fotos aleatorias de perros, siendo esta URL: <https://dog.ceo/api/breeds/image/random>. Además se debe implementar Async / Await a lo largo del ejercicio y utilizar la estructura [try...catch](#). Para ejecutar el código y atrapar un error en el caso de existir uno. Se solicita implementar el método fetch.

Para realizar este ejemplo, debemos seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un index.html y un script.js. Seguidamente, en el index.html debes escribir la estructura básica de un documento HTML como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Async/Await</title>
</head>
<body>
  <h4>Usando ES6 - Async / Await</h4>
  <script src="script.js"></script>
</body>
</html>
```

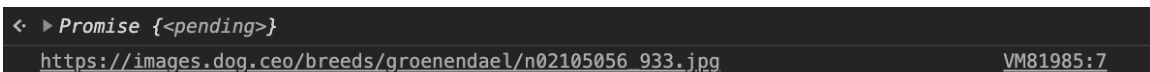
- **Paso 2:** En el archivo script.js, creamos primeramente una constante que maneje una función con la palabra reservada async para hacer a la función llamada `getDogPhoto` asíncrona, esta función se encargará de hacer la conexión y la traída de información de la URL que genera la foto aleatoria de los perros. Dentro de ella, vamos a definir una constante url con la dirección web de la cual deseamos traer la información.

```
const getDogPhoto = async () => {
  const url = 'https://dog.ceo/api/breeds/image/random';
}
```

- **Paso 3:** Ahora dentro de la función creada anteriormente, trabajaremos con la estructura try...catch, primeramente para indicar un bloque de instrucciones a intentar (try) y luego un bloque para indicar si se produce un error. En el bloque try, crearemos dos constantes, una que contenga el método fetch con el parámetro definido por la constante url, igualmente a este método le agregamos el operador await. Ahora, fetch retorna una promesa, pero el await ayudará a recibir la respuesta directamente de forma asíncrona, por consiguiente, se crea otra constante donde se almacenará la respuesta que traiga fetch pasando a JSON e implementando el operador await. Finalmente se muestra por la consola del navegador web la respuesta obtenida. En caso de algún error, se maneja en el bloque catch.

```
const getDogPhoto = async () => {  
  const url = 'https://dog.ceo/api/breeds/image/random';  
  try {  
    const response = await fetch(url);  
    const photo = await response.json();  
    console.log(photo.message);  
  } catch (err) {  
    console.log(err);  
  }  
}  
getDogPhoto();
```

- **Paso 4:** Por último, hacemos el llamado a la función, ejecutamos el archivo index.html en el navegador web y revisamos la consola para observar el resultado.



```
< ▶ Promise {<pending>}  
https://images.dog.ceo/breeds/groenendael/n02105056_933.jpg VM81985:7
```

Imagen 19. Resultado de la consola del navegador.
Fuente: Desafío Latam

Restricciones

Existen algunas restricciones para utilizar Async/Await y estas son:

1. Cualquier función puede ser asíncrona y utilizar **async**, pero hay que tener en cuenta que la función pasa a retornar una promesa, por lo que debe continuar su ejecución con **.then** para obtener el resultado.
2. Para utilizar la palabra reservada **await** debe hacerse solamente en funciones declaradas con **async**. No se puede usar en funciones regulares.
3. Hay que tener cuidado con el alcance de las funciones asíncronas, ya que por su naturaleza dejan continuar con la ejecución del resto del código.

Ejercicio propuesto (8)

Convertir el siguiente código para que se pueda usar promesas.

```
const log = (text, callback) => {
  setTimeout(() => {
    console.log(`${text}`);
    callback()
  }, 1000)
}
log('uno', () => {
  log('dos', () => {
    log('tres', () => { })
  })
})
})
```

Ejercicio propuesto (9)

Utilizando promesas, crea la función **verify** al que se le pasen dos argumentos **a** y **b**, (deben ser de tipo numérico), que los compare y resuelva la promesa si el primer número es mayor que el segundo calculando la potencia del número “a” elevado al número “b”, y rechace la promesa en caso contrario indicando que no se puede realizar la operación, muestre los datos en la consola.

Ejercicio propuesto (10)

Crea una función que permita traer y mostrar por la consola del navegador web la información de la siguiente dirección web: <https://www.feriadosapp.com/api/holidays.json>. Implementa Async...Await

Resumen

A lo largo de la lectura, se trabajó con distintos conceptos que permitieron visualizar y entender el funcionamiento de un código asíncrono. Por lo tanto:

- Se explicaron los conceptos de sincronía y asincronía, como se ejecutan e implementan en JavaScript.
- Trabajamos con el Event Loop para comprender como JavaScript desde su raíz implementa el código asíncrono.
- Logramos observar como el Race Condition puede afectar la ejecución de nuestro código.
- Se codificaron funciones asíncronas utilizando callbacks, setTimeout, Promise y ASYNC/AWAIT. Visualizando las distintas formas de crear código con cada una de estas sentencias.
- Finalmente se identificaron las restricciones de Async / Await y como debemos evitar errores en nuestro código al momento de aplicar estas funciones asíncronas.

Soluciones de los ejercicios propuestos

1. Ejecuta el código mostrado a continuación en la página web [Loupe](#), y describe el comportamiento con tus propias palabras.

El código mostrado al ser ejecutado en la página web indicada, se comporta de la manera esperada, es decir, primero muestra en la pila la comparación de la variable inicial del ciclo for con el largo del arreglo, y como la condición no se cumple, prosigue a cargar en la pila la carga de contenido a la variable "B", seguidamente muestra en el Call Stack el llamado a la función donde se sumará la variable más el número 2, retornando el resultado y repitiendo el ciclo nuevamente hasta que la condición en el ciclo for se deje de cumplir y muestra el valor de la variable "B" en la consola.

2. Desarrollar un programa con JavaScript implementando funciones con callback, que al pasarle los puntos obtenidos en dos ejercicios de un examen, sume ambos puntos e indique si hemos superado la prueba. El examen tiene una ponderación total de 10 puntos. Cada ejercicio tiene una nota máxima de 5 puntos.

```
const notas = (nota1,nota2,callback)=>{
  let total = nota1 + nota2;
  callback(total);
};

const evaluando = (total) => {
  if (total > 5) {
    console.log(`El alumno superó la prueba con: ${total} puntos`);
  } else {
    console.log(`El alumno no superó la prueba. Total puntos
alcanzados: ${total}`);
  }
};

notas(3,1,evaluando);
```

3. Crear la función "separar", donde se pasen dos (2) argumentos, un arreglo de números y un callback. La función deberá devolver un objeto con dos (2) arreglos, uno con los pares y otro con los impares. Ejemplo: Si se tiene el arreglo [3,12,7,1,2,9,18]. La función debe retornar: pares: [12,2,18], impares: [3,7,1,9].

```
const notas = (arreglo,callback)=>callback(arreglo);

const separar = (arreglo) => {
  let par = [];
  let impar = [];

  for (let index = 0; index < arreglo.length; index++) {
    if (arreglo[index] % 2 == 0) {
      par.push(arreglo[index]);
    } else {
      impar.push(arreglo[index]);
    }
  };
  return {par, impar}
};

let arreglo = [3,12,7,1,2,9,18];
console.log(notas(arreglo,separar));
```

4. Crear cuatro (4) funciones con las operaciones básicas matemáticas, sumar, restar, multiplicar y dividir, a las cuales se les debe enviar dos valores, el primer valor siempre será más grande que el segundo. Este segundo valor, no puede ser 0. Los valores deben ser devueltos en callbacks.

```
const opera = (num1,num2,callback)=>{
  if (num1 > num2 && num2 != 0){
    return callback(num1,num2);
  }else {
    return 'Los numeros no estan permitidos';
  }
};

const sumar = (num1,num2) => {
  return num1+num2;
};

const resta = (num1,num2) => {
  return num1-num2;
};
```



```
};  
const multiplica = (num1,num2) => {  
    return num1*num2;  
};  
const divide = (num1,num2) => {  
    return num1/num2;  
};  
  
console.log(opera(3,0,resta));
```

5. Crear un programa con JavaScript mediante el uso de funciones con Callback y el método `setTimeout`, que permita mostrar los datos de un usuario de acuerdo al nombre o el número de identificación.

```
function getUserData(id,name, callback){  
    setTimeout(() => {  
  
        let data = {}  
  
        if(id === 2356256 && name == 'Juan'){  
            data = {id: 2356256, name: 'Juan', lastName: 'Duran', age:  
35}  
        } else if(id === 27564512 && name == 'Manuel'){  
            data = {id: 27564512, name: 'Manuel', lastName: 'Perez',  
age: 31}  
        } else if(id === 17658624 && name == 'Jocelyn'){  
            data = {id: 17658624, name: 'Jocelyn', lastName:  
'Rodriguez', age: 30}  
        } else if(id === 12345678 && name == 'Maria'){  
            data = {id: 12345678, name: 'Maria', lastName: 'Garrido',  
age: 30}  
        } else {  
            data = 'El usuario no existe';  
        }  
  
        callback(data)  
    }, 1000)  
}  
  
getUserData(17658624, 'Jocelyn', user => console.log(user));
```

6. Crear un programa con JavaScript utilizando promesas que calcule un número aleatorio entre el "1" y el "100", pero que muestre el número aleatorio si y sólo si este número está comprendido entre "20" y "60", ambos valores incluidos.

```
const aleatorio = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let numAleatorio = Math.floor(Math.random()*100);
      resolve(numAleatorio);
    }, 1000);
  });
};

aleatorio().then(resultado => {
  if(resultado >= 20 && resultado <= 60){
    console.log(` ${resultado}`)
  }else {
    console.log(`El resultado: ${resultado} no se encuentra entre los números 20 y 60`);
  }
});
```

7. Realizar un programa en JavaScript que calcule la suma o la resta o la multiplicación de dos números. Para ello se debe implementar una promesa por cada operación matemática y solo se debe mostrar el resultado de la promesa ganadora. Utiliza tiempos aleatorios para los setTimeout que poseen cada promesa con su operación matemática. Igualmente implementa Promise.race para mostrar sola la primera promesa que retorne el resultado de de la operación matemática ganadora.

```
const randomNumber = (min, max) => {
  return parseInt(Math.random() * (max - min) + min);
}

const suma = (num1, num2) => new Promise((resolve, reject) => {
  setTimeout(() => {
    let sumaNum = `La suma es: ${num1+num2}`;
    resolve(sumaNum)
  }, randomNumber(500, 2000));
});

const resta = (num1, num2) => new Promise((resolve, reject) => {
  setTimeout(() => {
```

```
    let restaNum = `La resta es: ${num1-num2}`;  
    resolve(restaNum)  
  }, randomNumber(500, 2000));  
});  
  
const multiplica = (num1,num2) => new Promise((resolve, reject) => {  
  setTimeout(() => {  
    let multiplicaNum = `La multiplicación es: ${num1*num2}`;  
    resolve(multiplicaNum)  
  }, randomNumber(500, 2000));  
});  
  
const divide = (num1,num2) => new Promise((resolve, reject) => {  
  setTimeout(() => {  
    let divideNum = `La división es: ${num1/num2}`;  
    resolve(divideNum)  
  }, randomNumber(500, 2000));  
});  
  
Promise.race([suma(5,2), resta(5,2), multiplica(5,2),  
divide(5,2)]).then(response => {  
  console.log(`Operación Ganadora=> ${response}`);  
});
```

8. Convierte este código para que se pueda usar promesas.

```
const promesa1 = () => new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('uno')  
  }, 1000);  
});  
  
const promesa2 = () => new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('dos')  
  }, 2000);  
});  
  
const promesa3 = () => new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('tres')  
  }, 3000);  
});
```

```
promesa1().then(res=>console.log(res));  
promesa2().then(res=>console.log(res));  
promesa3().then(res=>console.log(res));
```

9. Utilizando promesas, crea la función `verify` al que se le pasen dos argumentos `a` y `b`, (deben ser de tipo numérico), que los compare y resuelva la promesa si el primer número es mayor que el segundo calculando la potencia del número “a” elevado al número “b”, y rechace la promesa en caso contrario indicando que no se puede realizar la operación, muestre los datos en la consola.

```
const verify = (a,b) => new Promise ((resolve, reject) => {  
  if (a > b) {  
    resolve(Math.pow(a,b));  
  } else {  
    reject('La operación no es permitida');  
  }  
});  
  
verify(5,2).then(res => console.log(res));
```

10. Crear una función que permita traer y mostrar por la consola del navegador web la información de la siguiente dirección web: <https://www.feriadosapp.com/api/holidays.json>. Implementa `Async...Await`

```
const getDatos = async () => {  
  const url = 'https://www.feriadosapp.com/api/holidays.json';  
  try {  
    const response = await fetch(url);  
    const datos = await response.json();  
    console.log(datos.data);  
  } catch (err) {  
    console.log(err);  
  }  
}  
  
getDatos();
```