Callbacks y APIs

Introducción a la asincronía

{desafío} latam_

Utilizar elementos de programación asíncrona para resolver un problema simple distinguiendo los diversos mecanismos para su implementación acorde al lenguaje Javascript.

Utilizar el objeto XHR y la API Fetch para el consumo de una API externa y su procesamiento acorde al lenguaje Javascript.

{desafío} latam_

- Unidad 1:
  ES6+ y POO
- Unidad 2:
  Herencia
- Unidad 3:
  Callbacks y APIs

Te encuentras aquí

¿Qué aprenderás en esta sesión?

- *Explica el concepto de programación asíncrona reconociendo el problema que resuelve y los mecanismos disponibles en Javascript.*
- *Distingue las diferencias en la utilización de los distintos mecanismos de programación asíncrona (callbacks, promises, async/await) reconociendo sus ventajas y desventajas.*

{desafío} latam_

¿Quieres comentar lo aprendido en la clase anterior?

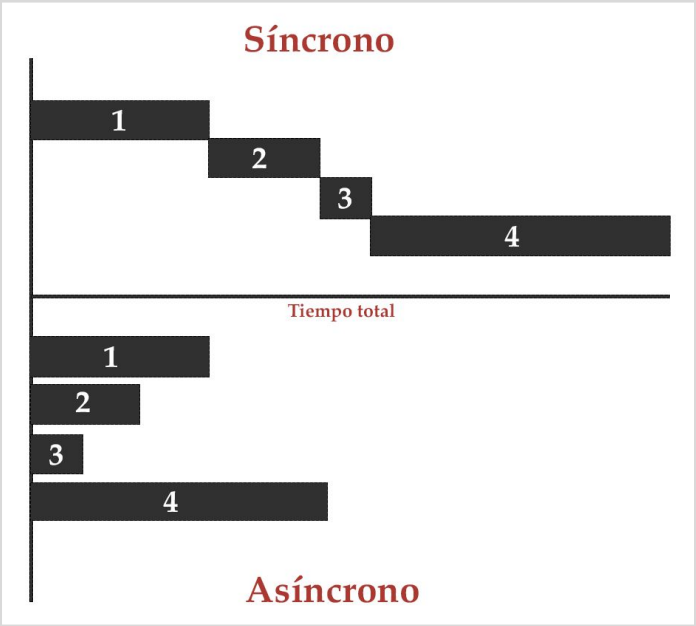
{desafío} latam_

/* Sincronía y Asincronía */

Sincronía y Asincronía

Síncrono, cuando los procesos se ejecutan uno tras otro.

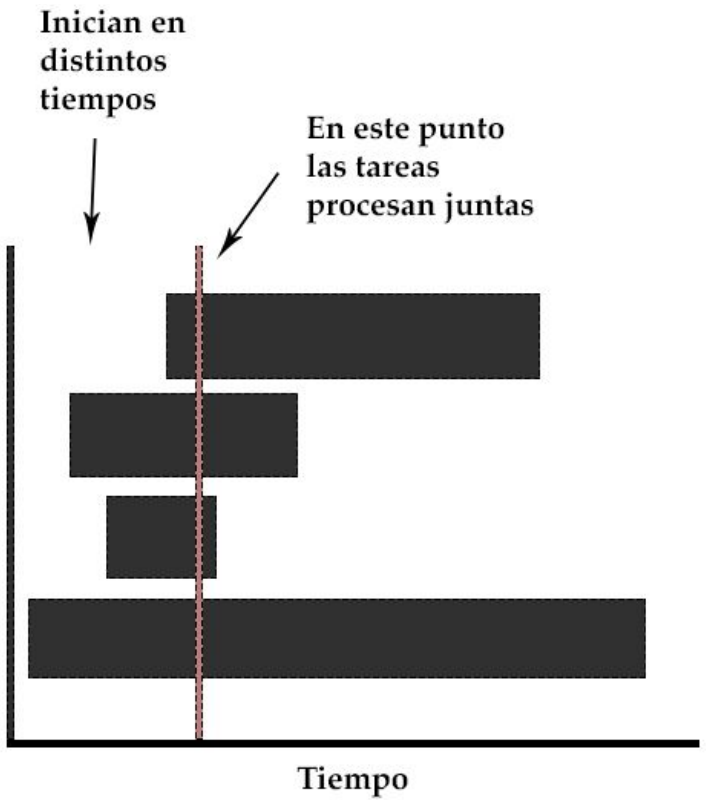
Asíncrono, cuando los procesos se ejecutan todos a la vez y no necesitan esperar a que finalicen los otros.



/* Concurrencia */

Concurrencia

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



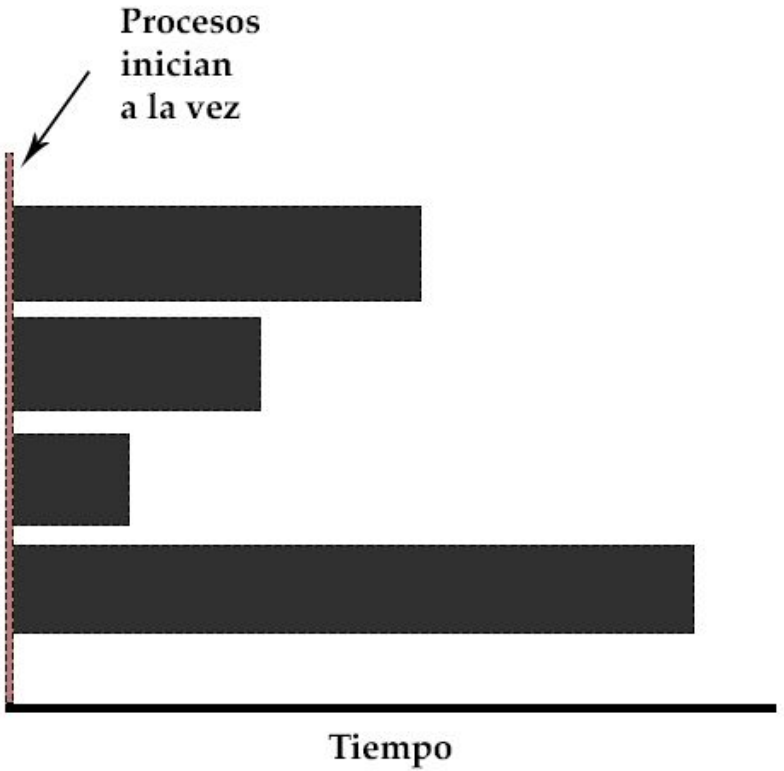
Fuente: Desafío Latam

{desafío} latam_

/* Paralelismo */

Paralelismo

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

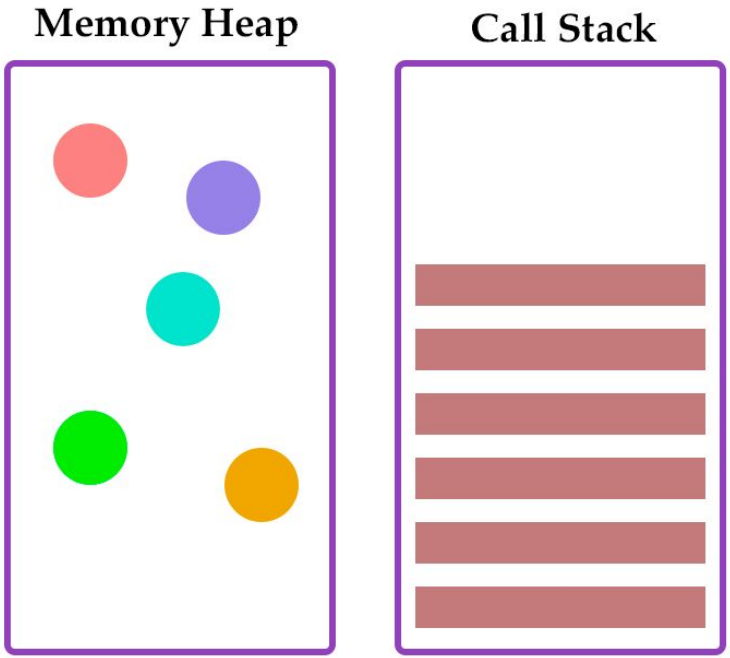


Fuente: Desafío Latam

/* Event Loop o Bucle de Eventos */

Event Loop o Bucle de Eventos

Se encarga de implementar las operaciones asíncronas



Fuente: Desafío Latam

/* Memory Heap */

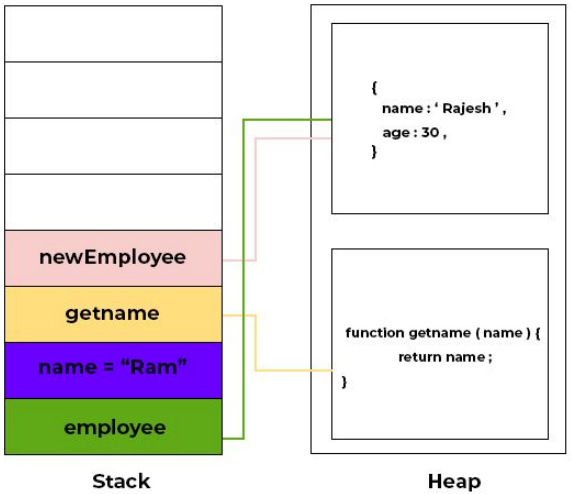
Memory Heap

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.

```
const employee = {
    name : ' Rajesh ' ,
    age : 30 ,
};
const name = "Ram" ;

function getname ( name ) {
    return name ;

}
const newEmployee = employee ;
```

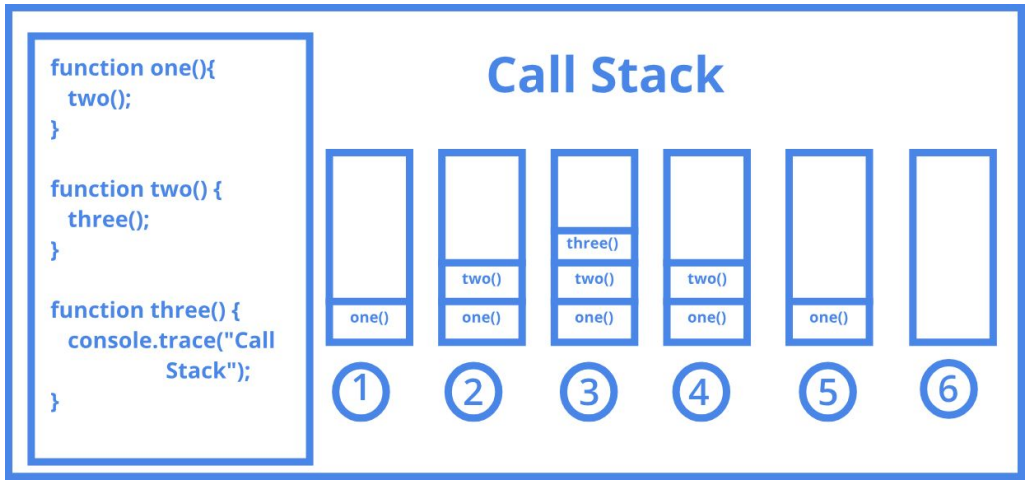


Fuente: https://www.geeksforgeeks.org/

/* Call stack */

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 este.

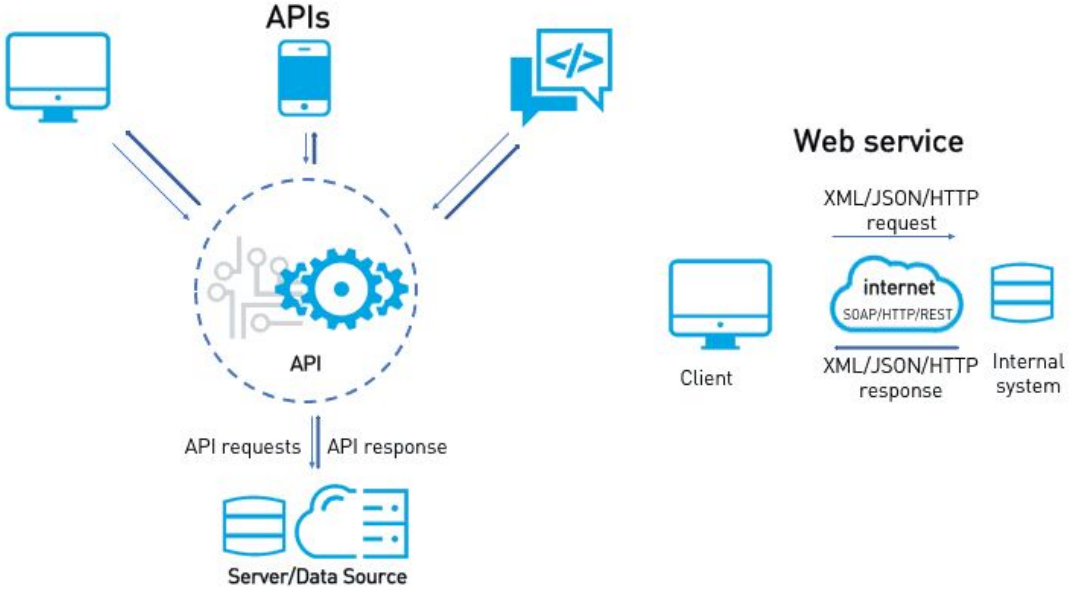


Fuente: www.medium.com

/* WEB APIs */

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.



{desafío}
latam_

/* Callback Queue */

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.

Ejercicio guiado: Callback Queue

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:

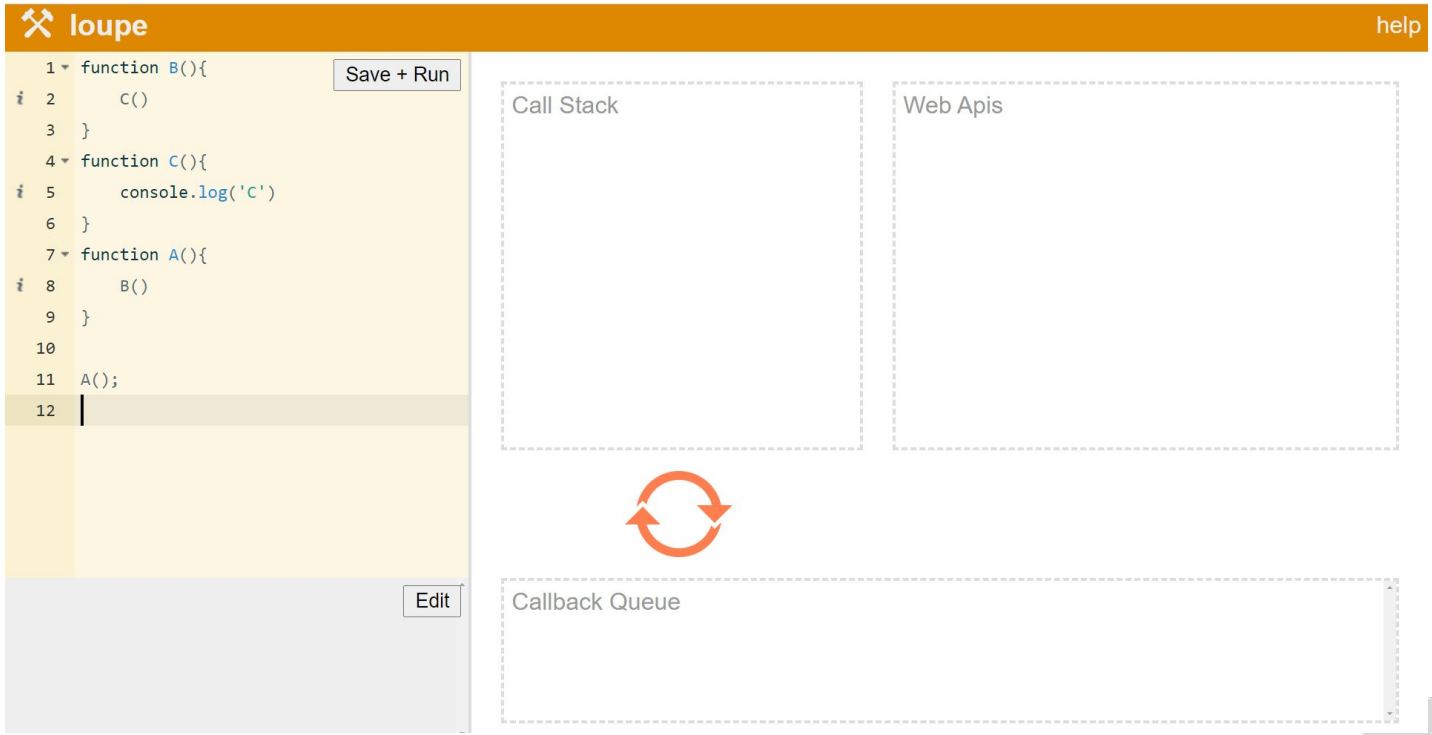
Paso 1: Ingresa a la página Loupe con tu navegador favorito.

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 de la slide siguiente.

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

Ejercicio guiado

Callback Queue



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

{desafío} latam_

Ejercicio guiado

Callback Queue

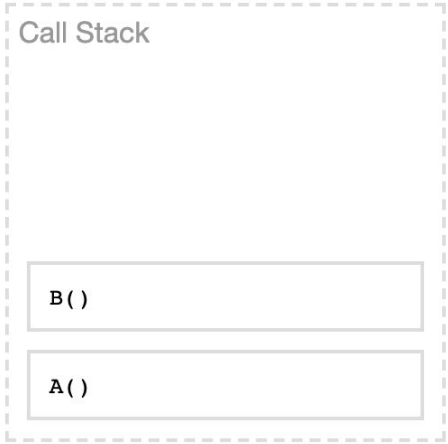
Paso 3: Solo falta presionar “Save + Run” para iniciar el proceso y ver la respuesta que genera la página web para nuestro código. Al comenzar la ejecución de este script se agrega la función A al Call Stack como se muestra en la imagen.

Fuente: Desafío Latam

Ejercicio guiado

Callback Queue

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.



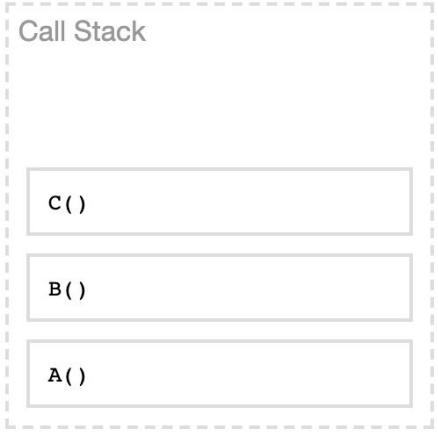
Fuente: Desafío Latam

{desafío} latam_

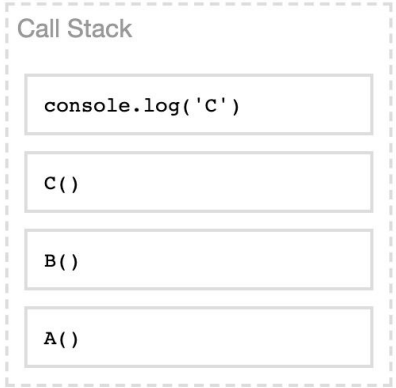
Ejercicio guiado

Callback Queue

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 de arriba. Aquí vemos que continúan las funciones A y B, ya que no han finalizado.



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 de abajo.



Fuente: Desafío Latam

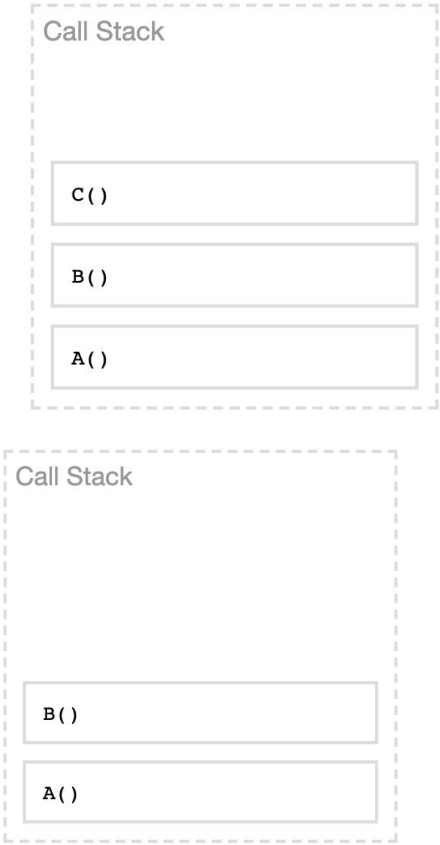
{desafío} latam_

Ejercicio guiado

Callback Queue

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 aún en la pila. Como se visualiza en la imagen de arriba.

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 de abajo.



Fuente: Desafío Latam

{desafío} latam_

Ejercicio guiado

Callback Queue

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.

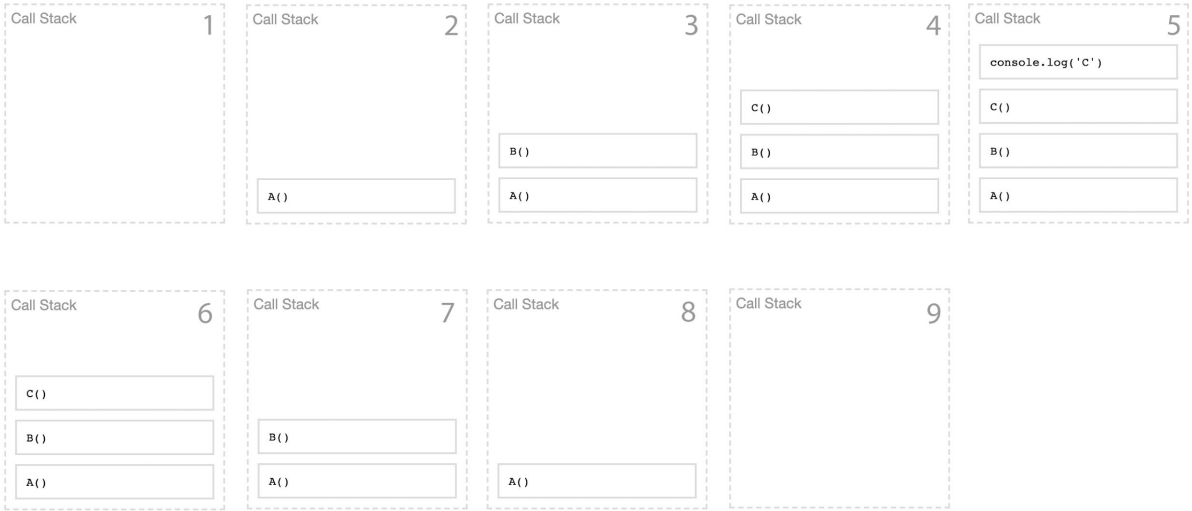
Paso 10: Como la función A ya no tiene nada más que realizar, entonces se elimina del stack y finaliza el proceso.

Fuente: Desafío Latam

Ejercicio guiado

Callback Queue

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



Fuente: Desafío Latam

/* Callbacks */

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();
}
```

/* Race condition */

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);
}
```

Race condition

Aquí tenemos una variable global que es modificada por dos funciones, y en el caso de que estas dos (2) funciones se ejecutaran 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 serían los siguientes:

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.

Race condition

¿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.

/* Async / Await */

Async / Await

- Permite que las funciones que retornan promesas, devuelvan directamente los resultados en vez de promesas.

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

- Sintaxis Async:

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

- La palabra reservada await la utilizaremos para esperar y retornar la promesa. El operador await solo trabaja dentro de una función async.
- Sintaxis Await

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

{desafío} latam_

/* Restricciones */

Restricciones

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.

Para utilizar la palabra reservada **await** debe hacerse solamente en funciones declaradas con async. No se puede usar en funciones regulares.

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.

¿Qué concepto te costó más comprender?

{desafío} latam_

Resumiendo

- Proceso **Síncrono**, cuando los procesos se ejecutan uno tras otro.
- Proceso **Asíncrono**, cuando los procesos se ejecutan todos a la vez y no necesitan esperar a que finalicen los otros.
- La concurrencia se produce cuando dos o más tareas progresan simultáneamente.
- Las WEB APIs son funciones disponibilizadas por el navegador y que se pueden usar en JavaScript para comunicarnos e interactuar con el Frontend.

Próxima sesión...

- *Callbacks y promesas*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales