Clase 14 - Asincronía y Promesas

***Asincronía***

**Programación sincrónica**

Hasta el momento, los programas que estuvimos escribiendo funcionan realizando una tarea a la vez. Escribimos scripts donde se declaran variables y valores, objetos, arrays, funciones de diverso tipo, eventos, y demás; entendiendo que el navegador interpreta y ejecuta nuestro código en orden, una línea a la vez, y que cada instrucción para realizarse debe esperar a que la instrucción anterior haya terminado. En resumen, nuestro programa sólo puede ejecutar una tarea a la vez, y cada tarea debe esperar a que se haya terminado la anterior para continuar.

Esto es lo que entendemos como un modelo de programación *sincrónico*. Nuestro programa funciona de forma *lineal*, ejecutando una acción después de la otra, y nosotros como desarrolladores también lo pensamos así; por lo que es coherente que escribamos nuestro script en cascada, de arriba hacia abajo, entendiendo que ese es el orden de sucesos a esperar.

Sin embargo, ¿qué sucede cuando queremos que nuestro programa realice más de una acción a la vez? En las aplicaciones modernas es común que nuestro programa interactúe con otras aplicaciones y servicios. Supongamos que nuestra aplicación debe hacer una petición de información a una base de datos para cargar resultados en el DOM, y que ésta petición tarda 10 segundos. En un modelo sincrónico, tras iniciar la petición nuestro programa quedaría inmóvil durante 10 segundos hasta que esta termine para poder continuar con el resto de los procesos, porque sólo podemos realizar una tarea a la vez, y cada tarea es bloqueante de la siguiente.

**Programación asincrónica**

En un modelo sincrónico las cosas suceden una a la vez. Si llamamos una función de larga ejecución, ésta retorna sólo cuando haya terminado la acción y pueda devolver el resultado. Esto detiene el programa por el tiempo que tarda la acción.

Un **modelo asincrónico** permite que múltiples cosas sucedan a la vez. Al comenzar una acción, nuestro programa sigue en ejecución; y cuando la acción termina nuestro programa es informado de ésto y tiene acceso al resultado. Retomando el ejemplo anterior, en un modelo asincrónico hacemos una petición a la base de datos y ,durante los 10 segundos que tarda en resolverse, nuestro programa sigue activo pudiendo realizar otras tareas, y cuando la petición finaliza somos informados de ésto con el resultado y volvemos a sincronizar los datos con la ejecución.

Si tuviéramos que hacer dos peticiones, en un modelo sincrónico la segunda petición empezaría solamente cuando termine la primera, y tendríamos el resultado de ambas luego de finalizar la segunda; en un modelo asincrónico ambas peticiones pueden realizarse en paralelo, y sólo debemos sincronizar los resultados cuando terminen.

***síncrono***

*1° pet. result 1° - 2° pet. result 2°*



***asincrónico***

*1° pet. 2° pet. resultado 1° resultado 2°*

**

Sabiendo que nuestro programa continúa en ejecución realizando otras tareas en el modelo asincrónico mientras se esperan los resultados de las peticiones, lo que tenemos que saber resolver es *cuándo y cómo sincronizar los resultados con nuestra ejecución vigente*.

Por un lado, la asincronía facilita el manejo de programas que realizan múltiples acciones a la vez, y por otro puede dificultar la comprensión de aquellos programas que tienden a seguir una única línea de acción, como los que venimos construyendo hasta ahora. Entender cómo funciona la asincronía en este contexto y cómo resolver situaciones comunes es el propósito de esta clase.

**setTimeout**

Para entender mejor las implicancias de esto vamos a la práctica con una función que nos permite realizar acciones asincrónicamente, *setTimeout*. La función recibe dos parámetros: primero una función de *callback*, y segundo un valor numérico que representa *milisegundos*.

| setTimeout(fn, time) |
| --- |

Esto significa que la función que pasamos por primer parámetro se ejecuta *luego* de que transcurra el tiempo definido en el segundo parámetro. Por ejemplo:

| setTimeout(()=> {  console.log("Proceso asincrónico")  }, 3000) |
| --- |

Con esto veríamos el console.log ejecutarse con un delay de tres segundos. Lo interesante está en ver cómo trabaja esto dentro del marco sincrónico al que estamos acostumbrados.

Pensando de forma sincrónica, en el siguiente ejemplo esperaríamos ver lo siguiente, en el siguiente orden:

* “Inicia proceso”
* “Mitad de proceso” (tras 2 segundos)
* “Fin proceso”

| console.log("Inicia proceso")  setTimeout(()=> {  console.log("Mitad de proceso")  }, 2000)  console.log("Fin proceso") |
| --- |

Sin embargo, la salida se produce de la siguiente forma:

| // Inicia proceso  // Fin proceso  // Mitad de proceso - tras 2 segundos |
| --- |

Esto sucede porque *setTimeout* funciona de forma asincrónica. Es un proceso que se ejecuta en paralelo al resto de las instrucciones sincrónicas, y no es bloqueante de éstas. Significa que al ejecutarse el *setTimeout* se entiende que es asincrónico, su ejecución va en paralelo, y se continúa con el resto de las instrucciones sincrónicas. Por eso es que los dos console.log se ejecutan antes, y por último vemos el resultado del setTimeout que va en el medio.

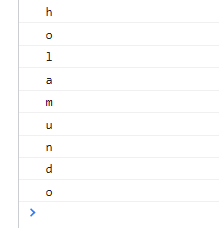
Podemos utilizar *setTimeout* para generar interacciones interesantes de forma automática. Por ejemplo, estilar algún elemento del DOM tras un evento, y removerle una clase tras cierto tiempo.

| const btn = document.querySelector('#boton')  const popup = document.querySelector('#popup-mensaje')  btn.addEventListener('click', () => {  popup.classList.add('popup-active')  setTimeout(() => {  popup.classList.remove('popup-active')  }, 2500)  }) |
| --- |

Este ejemplo agrega una clase a un elemento tras clickear un botón dispara un setTimeout que remueve esa clase del elemento tras 2500 milisegundos. Es una forma de automatizar un evento interactivo.

Para casos sencillos asociados a eventos esto puede resultar muy conveniente. Sin embargo el problema está cuando tratamos combinar de forma compleja ambos tipos de procesos.

Por ejemplo, veamos el siguiente código. De forma sincrónica, esperamos ver en vertical un console.log de cada letra de ambos strings, uno después del otro:



| for (let letra of "hola") {  console.log(letra)  }  for (let letra of "mundo") {  console.log(letra)  } |
| --- |

Sin embargo, ¿qué pasa cuando dentro de los iteradores agregamos un setTimeout a cada console.log? A la vista parece algo ambiguo, y quizás esperamos ver un console.log de cada letra con un delay entre ellos. Lo que termina sucediendo, es que el iterador ejecuta de forma *sincrónica* los llamados al setTimeout por cada letra, y estos se resuelven en orden con el delay correspondiente de forma asincrónica:

| for (let letra of "hola") {  setTimeout(() => {  console.log(letra)  }, 1000)  }  for (let letra of "mundo") {  setTimeout(() => {  console.log(letra)  }, 3000)  } |
| --- |

Algo importante que hay que entender, es que esto no tiene que ver precisamente con el tiempo de ejecución del proceso, sino con qué orden ocupa en el listado de órdenes a ejecutar. ¿A qué nos referimos con esto? Si fuese realmente por el tiempo de ejecución, podemos suponer que un *setTimeout* con 0 milisegundos se ejecuta de forma inmediata, sin irrumpir el orden sincrónico del programa. Pero vemos que el efecto sigue siendo igual que antes:

| console.log("Inicia proceso")  setTimeout(()=> {  console.log("Mitad de proceso")  }, 0)  console.log("Fin proceso") |
| --- |

Para entender esto, hay que entender qué es el *callstack* y el *event loop*.

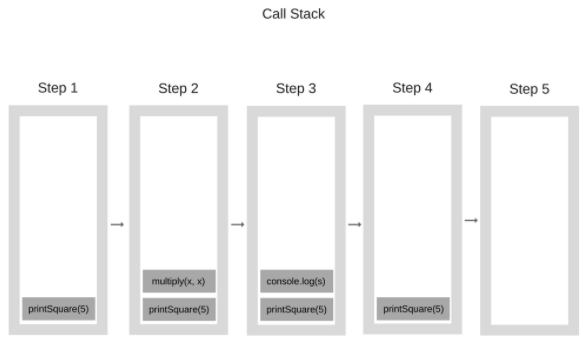
**Call Stack**

Un stack, o pila, es una lista donde se *apilan* las distintas tareas a ejecutar por nuestro programa. Javascript es un lenguaje *single threaded*, o de un único hilo, lo que significa que tiene un único *stack* o pila de ejecución. De ahí que la ejecución es implícitamente sincrónica, porque tiene una única pila donde ir listando las tareas a ejecutar.

El proceso que realiza el call stack es simple, cuando se está a punto de ejecutar una función, esta es añadida al stack. Si la función llama a su vez, a otra función, es agregada sobre la anterior. Por ejemplo:

| function multiply (x, y) {  return x \* y;  }  function printSquare (x) {  let s = multiply(x, x);  console.log(s);  }  printSquare(5); |
| --- |

Los estados de la call stack serían:



En resumen, podemos pensar la *call stack* como la lista de tareas a ejecutar que tiene Javascript durante el funcionamiento de nuestro programa. Cada nueva instrucción se agrega en el orden que corresponde al stack y el motor de JS la va resolviendo una a la vez.

Pero qué pasa si llamamos a un timeout o hacemos un request a un servidor. Al ser un solo thread, hay un solo call stack y por lo tanto solo se puede ejecutar una cosa a la vez. Es decir el navegador debería congelarse, no podría hacer más nada hasta que la llamada termine de ejecutarse. Sin embargo esto no es así, javascript es asincrónico y no bloqueante. Esto es gracias al **Event Loop.**

**Event Loop**

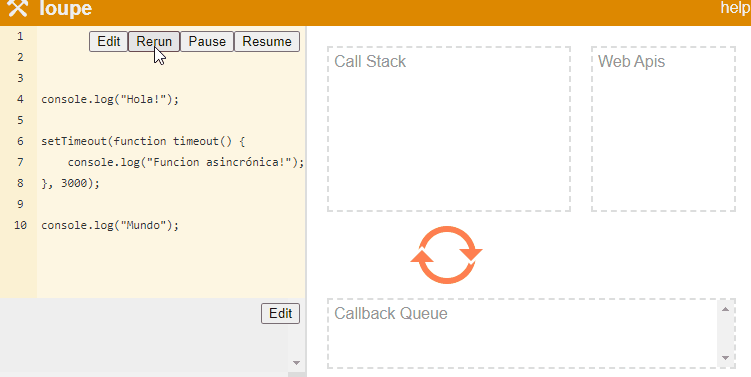
Javascript no cuenta nativamente con cosas como *setTimeout, DOM* o *HTTP requests*. Estas son llamadas *web APIs* y son provistas por el navegador, pero no están dentro del runtime de JS.

Muchas de estas funciones asincrónicas se ejecutan en un *stack* diferente. El ***Event Loop*** es la herramienta que permite la sincronización entre nuestro *callstack* con estas tareas asincrónicas que funcionan en un thread aparte. Las tareas asincrónicas se van resolviendo como callbacks, y hay una estructura diferente donde se van listando a medida que se van resolviendo. Este espacio se denomina *callback queue.*

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

***LOUPE***

Todo esto es más fácil de entender visualmente, con un gráfico que describa este funcionamiento. Para esto existe *Loupe (* [*Loupe*](http://latentflip.com/loupe) *)* que nos permite ver pausadamente y en detalle el funcionamiento de este mecanismo. Podemos escribir código sincrónico y asincrónico y ver (y entender) cómo es el funcionamiento de estas piezas en conjunto:



Sabiendo cómo funciona el *motor* de Javascript para organizar las tareas sincrónicas y asincrónicas, queda más claro cómo es posible que una instrucción como la siguiente tiene el resultado que tiene:

| console.log("Inicia proceso")  setTimeout(()=> {  console.log("Mitad de proceso")  }, 0)  console.log("Fin proceso")  // Inicia proceso  // Fin proceso  // Mitad de proceso |
| --- |

Por más que hagamos el timeout con 0 milisegundos, éste se envía al stack de *web apis* primero, luego al *callback queue*, y finalmente el *event loop* lo incorpora al *callstack* para su ejecución, luego de los console.log anteriores.

**setInterval**

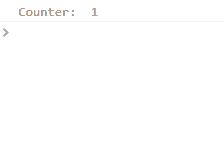
Similar al setTimeout, tenemos la función de *setInterval*, que tiene la misma sintaxis, pero la unidad de tiempo es un *intervalo* para la repetición de la función asociada:

| setInterval(() => {  console.log("Tic")  }, 1000) |
| --- |

Ésto es útil cuando queremos ejecutar funciones de forma reiterativa. La función se ejecutará repetidamente tras los milisegundos indicados hasta que indiquemos su detención, o se cierre la aplicación.

**clearInterval y clearTimeout**

Hay casos donde vamos a querer remover un intervalo o un timeout asignado previamente. En el caso de un intervalo, usamos la función *clearInterval()* para detener el mismo, y en el caso de haber llamado un *setTimeout()* podemos detener su futura ejecución invocando un *clearTimeout()*.

Cuando llamamos un *setInterval()* éste retorna una referencia al intervalo generado, el cual podemos almacenar en una variable. Es esta referencia la que debemos pasar a la función *clearInterval* para que la limpieza tenga efecto:

| let counter = 0  const interval = setInterval(() => {  counter++  console.log("Counter: ", counter)  if (counter >= 5) {  clearInterval(interval)  console.log("Se removió el intervalo")  }  }, 1000) |
| --- |

Lo mismo funciona con los timeout. Si guardamos en una variable la referencia al timeout generado, podemos usarla para removerlo luego por algún motivo que determinemos. En el siguiente caso, el timeout generado nunca llega a ejecutarse:

| console.log("Inicio")  const fin = setTimeout(() => {  console.log("fin")  }, 2000)  clearTimeout(fin) |
| --- |

***Promesas***

Trabajar con asincronía se vuelve mucho más sencillo cuando podemos representarla con valores. Una promesa es un objeto de Javascript que representa un evento a futuro. Es una acción asincrónica que se puede completar en algún momento y producir un valor, y notificar cuando esto suceda.

Podemos crear promesas a través de su constructor *new Promise*. Su sintaxis es algo compleja, ya que recibe una función por parámetro que a su vez recibe por parámetro las funciones de *resolve* y *reject*.

| new Promise( (resolve, reject) => {  //cuerpo de la promesa  }) |
| --- |

***Resolve y Reject***

Una promesa nos devuelve inmediatamente un valor que puede modificarse en el tiempo, ya que representa ésto mismo: la *promesa* de generar un valor a futuro. Es decir, una promesa cuenta **con tres estados posibles:** *pending, fulfilled, rejected*.

En principio, una promesa se retorna con estado *pending*, entendiendo que el valor a generar aún no fue resuelto. Pero las promesas contemplan la posibilidad de ser *resueltas* o *rechazadas*. Es decir, tienen la capacidad de generar distintos resultados según un caso u otro.

| const eventoFuturo = () => {  return new Promise( (resolve, reject) => {  //cuerpo de la promesa  } )  }  console.log( eventoFuturo() ) // Promise { <pending> } |
| --- |

Esta función retorna una promesa que *no se resuelve*. Por lo tanto, veremos que el valor que genera es un objeto Promise con estado pendiente. El valor de retorno de la promesa se define a través del llamado a las funciones de *resolve* o *reject*:

* Si el cuerpo de la promesa llama a *resolve()*, la promesa cambiará su estado a *fulfilled*, con el valor enviado a *resolve()*.
* Si la promesa llama a *reject()*, cambiará su estado a *rejected* con el valor enviado al *reject()*.

| const eventoFuturo = (res) => {  return new Promise( (resolve, reject) => {  if (res === true) {  resolve('Promesa resuelta')  } else {  reject('Promesa rechazada')  }  })  }  console.log( eventoFuturo(true) ) // Promise { 'Promesa resuelta' }  console.log( eventoFuturo(false) ) // Promise { <rejected> 'Promesa rechazada' } |
| --- |

En este ejemplo podemos ver como cambia de estado la promesa con distintos valores. Según el llamado de la función la promesa se verá resuelta o rechazada. Sin embargo, lo que vemos por consola es el objeto Promise que retorna la función, y con lo que nos interesa trabajar en realidad es con el *valor* de resolución de la promesa. Miremos el mismo caso agregando un delay con setTimeout:

| const eventoFuturo = (res) => {  return new Promise( (resolve, reject) => {  setTimeout( () => {  res ? resolve('Promesa resuelta') : reject('Promesa rechazada')  }, 2000)  })  }  console.log( eventoFuturo(true) ) // Promise { <pending> }  console.log( eventoFuturo(false) ) // Promise { <pending> } |
| --- |

En este caso, el console.log es sincrónico y vemos que la promesa está en *pending* en ambos llamados. Su resolución se generará dentro de 2 segundos. Las promesas tienen un mecanismo para trabajar esta asincronía y poder ejecutar funciones cuando cambie su estado, es decir cuando se *resuelva* o se *rechace.*

**THEN y CATCH**

Al llamado de una función que retorne una promesa, podemos concatenar el método *.then()* o *.catch()*, los cuales reciben una función por parámetro con la cual se captura el valor de la promesa.

* **.then()** : Si la promesa es *resuelta*, es decir se llama al *resolve()* en ella, su valor de retorno se captura dentro del *.then()*, recibiendo por parámetro de su función ese valor.
* **.catch()** : si la promesa es *rechazada*, su valor se captura dentro de un *.catch()* siguiendo la misma lógica.

Significa que lo que queramos ejecutar cuando la promesa se resuelva o rechace, debemos definirlo dentro de un .then() o .catch(), según el caso:

| eventoFuturo(true)  .then( (response) => {  console.log(response) // Promesa resuelta  })  eventoFuturo(false)  .catch( (error) => {  console.log(error) // Promesa rechazada  }) |
| --- |

Se aprecian los console.log tras 2 segundos de delay, y lo que vemos es precisamente el *valor* que retornan el *resolve* o *reject* de la promesa. Significa que una promesa puede retornar cualquier tipo de valor, hacerlo condicionalmente pudiendo resolverse o rechazarse, y ofrecernos una estructura con la cual podemos responder a cualquier caso capturando sus valores de retorno, respetando la asincronía.

Dado que una promesa puede tener varios estados posibles según el caso, podemos concatenar varios *.then()* o *.catch()* en un mismo llamado, y caeremos en el caso que corresponda según cómo se haya resuelto la promesa:

| eventoFuturo(true)  .then( (response) => {  console.log(response) // Promesa resuelta  })  .catch( (error) => {  console.log(error)  })  eventoFuturo(false)  .then( (response) => {  console.log(response)  })  .catch( (error) => {  console.log(error) // Promesa rechazada  }) |
| --- |

Es decir que para cada promesa podemos definir una *estructura* para trabajar los distintos casos posibles. Cada promesa sólo puede **resolverse o rechazarse una única vez**. Si se resuelve, caeremos en el primer *.then()* de la secuencia, y si se rechaza los .then() se omiten hasta caer en el primer *.catch()* que capture el error. Es un mecanismo de control claro y ordenado para trabajar la asincronía y los posibles valores a recibir.

**Finally**

Las promesas nos ofrecen también un método *finally()* que recibe una función la cual se ejecutará *siempre* al finalizar la secuencia, sin importar si se haya resuelto o no la promesa:

| eventoFuturo(true)  .then( (response) => {  console.log(response)  })  .catch( (error) => {  console.log(error)  })  .finally( () => {  console.log("Fin del proceso")  })  // Promesa resuelta  // Fin del proceso |
| --- |

**Ejemplo aplicado: simular petición de datos**

Aprovechando lo que vimos de asincronía y promesas, podemos simular una acción típica de una web app, como es la petición de datos a algún servidor y generar alguna interacción con el resultado.

En el siguiente ejemplo, tenemos un array de productos vacíos. Al cargar la aplicación queremos simular un *delay* para la actualización de estos datos, y lo haremos llamando una promesa que retorna el array de productos real. Cuando captemos su resolución, en el *.then()* actualizamos nuestro array y llamamos una función para generar la vista del resultado:

| // función que tras 3 segundo retorna un array de objetos  const BD = [  {id: 1, nombre: 'Producto 1', precio: 1500},  {id: 2, nombre: 'Producto 2', precio: 2500},  {id: 3, nombre: 'Producto 3', precio: 3500},  {id: 4, nombre: 'Producto 4', precio: 3500},  ]  const pedirProductos = () => {  return new Promise( (resolve, reject) => {  setTimeout(() => {  resolve(BD)  }, 3000)  })  } |
| --- |

| // inicializamos con un array vacío  let productos = []  const renderProductos = (arr) => {  // función que genere la vista de los productos  }  // asincrónicamente pedimos los datos y generamos la vista  pedirProductos()  .then((res) => {  productos = res  renderProductos(productos)  }) |
| --- |