Javascript Asíncrono

Concurrencia y Paralelismo

Concurrencia y paralelismo son conceptos relacionados, pero con un importante matiz de diferencia entre ellos. Es por esto por lo que muy a menudo se confunden y se utilizan erróneamente. A saber:

- Concurrencia: cuando dos o más tareas progresan simultáneamente.
- Paralelismo: cuando dos o más tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.

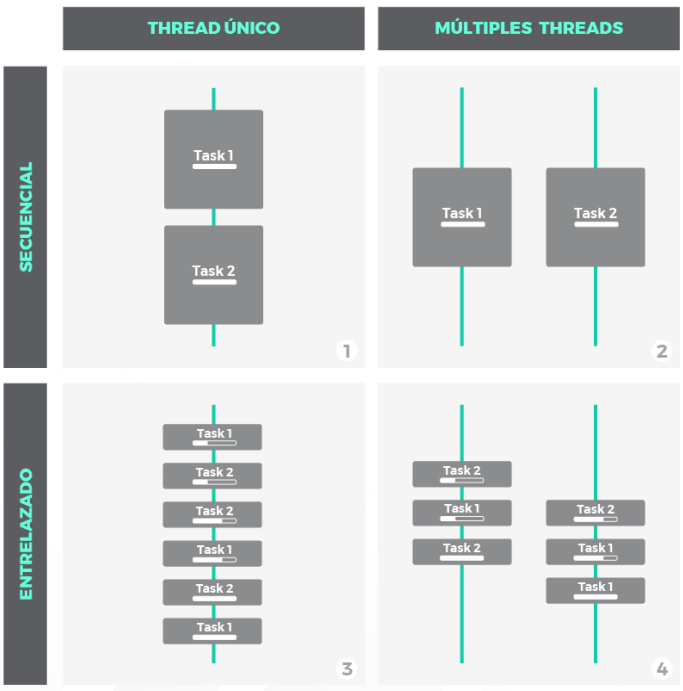
"Que varias tareas progresen simultáneamente no tiene por qué significar que sucedan al mismo tiempo."

"Mientras que la concurrencia aborda un problema más general, el paralelismo es un sub-caso de la concurrencia donde las cosas suceden exactamente al mismo tiempo."

Se cree que la concurrencia implica necesariamente más de un thread (hilo). Esto no es cierto. El entrelazado (multiplexado), por ejemplo, es un mecanismo común para implementar concurrencia en escenarios donde los recursos son limitados.

Ejemplo:

Los sistemas operativos modernos haciendo multitarea con un único core (núcleo), simplemente trocea las tareas en tareas más pequeñas y las entrelaza, de modo que cada una de ellas se ejecutará durante un breve instante. Sin embargo, a largo plazo, la impresión es que todas progresan a la vez.



- **Escenario 1**: no es ni concurrente ni paralelo. Es simplemente una ejecución secuencial, primero una tarea, después la siguiente.
- **Escenario 2**, **3** y **4**: son escenarios donde se ilustra la concurrencia bajo distintas técnicas: **Escenario 3**: muestra como la concurrencia puede conseguirse con un único *thread*. Pequeñas porciones de cada tarea se entrelazan para que ambas mantengan un progreso constante. Esto es posible siempre y cuando las tareas puedan descompuestas en subtareas más simples. **Escenario 2** y **4**: **ilustran** paralelismo, utilizando multiples *threads* donde las tareas o subtareas corren en paralelo exactamente al mismo tiempo. **A nivel de *thread*, el escenario 2 es secuencial,**

mientras que 4 aplica entrelazado.

Entonces…

JavaScript es un lenguaje de programación de un solo subproceso, lo que significa que solo puede suceder una cosa a la vez. Es decir, el motor de JavaScript solo puede procesar una declaración a la vez en un solo hilo.
Si bien los lenguajes de un solo subproceso simplifican la escritura de código porque no tiene que preocuparse por los problemas de concurrencia, esto también significa que no puede realizar operaciones largas como el acceso a la red sin bloquear el subproceso principal.

Ejemplo:

Imagínese solicitar algunos datos de una API. Dependiendo de la situación, el servidor puede tomar algún tiempo para procesar la solicitud mientras bloquea el hilo principal y hace que la página web no responda.
Ahí es donde entra en juego JavaScript asincrónico…

"Al usar JavaScript asincrónico (como devoluciones de llamada, promesas y async / await), puede realizar solicitudes de red largas sin bloquear el hilo principal."

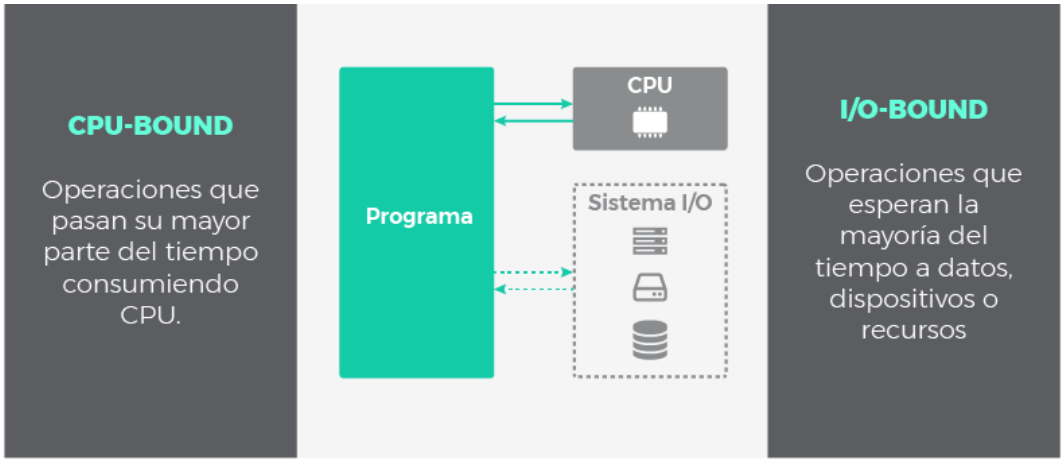
Operaciones CPU-Bound vs. I/O-Bound

CPU-bound.

- Son tareas que consume recursos de CPU.
- Estas tareas se componen de operaciones cuya carga (el código asociado a ellas) será ejecutada en nuestra aplicación.
- Se las conoce como operaciones limitadas por CPU, o en inglés, operaciones
- La naturaleza de las operaciones *CPU-bound* es intrínsecamente síncrona (o secuencial, si la CPU está ocupada no puede ejecutar otra tarea hasta que se libere) a menos que se utilicen mecanismos de concurrencia como los vistos anteriormente (entrelazado o paralelismo, por ejemplo).

I/O-Bound

- Son operaciones de entrada/salida que disparan peticiones especiales y que son atendidas fuera del contexto de nuestra aplicación.
- Ejemplo: Leer un archivo en disco, acceder a una base de datos externa o consultar datos a través de la red.
- las operaciones ***I/O-bound*** (limitadas por entrada/salida) no *corren* o se *ejecutan* en el dominio de nuestra aplicación.
- Pueden ser asíncronas, y la asincronía es una forma muy útil de concurrencia.



Entonces…

"Si incrementamos la potencia de nuestra CPU, mejoraremos el rendimiento de las operaciones *CPU-bound*, mientras que una mejora en el sistema de entrada/salida favorecerá el desempeño de las operaciones *I/O-bound*."

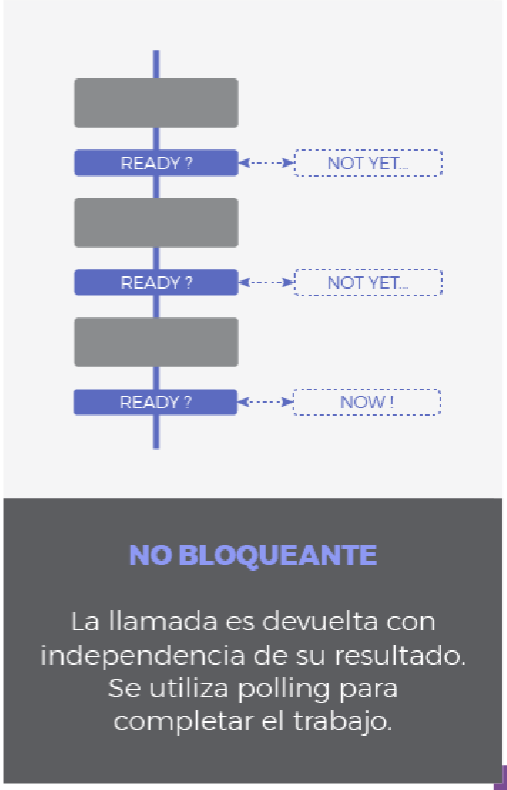
Naturaleza I/O: Bloqueante vs. No-bloqueante & Síncrono vs. Asíncrono

Estos términos no siempre son aplicados de forma consistente y dependerá del autor y del contexto. Muchas veces se utilizan como sinónimo o se mezclan para referirse a lo mismo. Una posible clasificación en el contexto *I/O* podría hacerse si imaginamos las operaciones I/O comprendidas en dos fases:

1. **Fase de Espera** a que el dispositivo esté listo, a que la operación se complete o que los datos estén disponibles.
2. **Fase de Ejecución** entendida como la propia respuesta, lo que sea que quiera hacerse como respuesta a los datos recibidos.

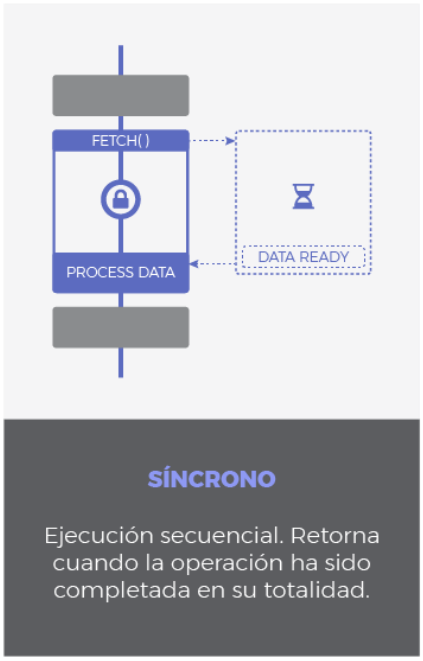
Bloqueante vs No-bloqueante hace referencia a como la fase de espera afecta a nuestro programa:

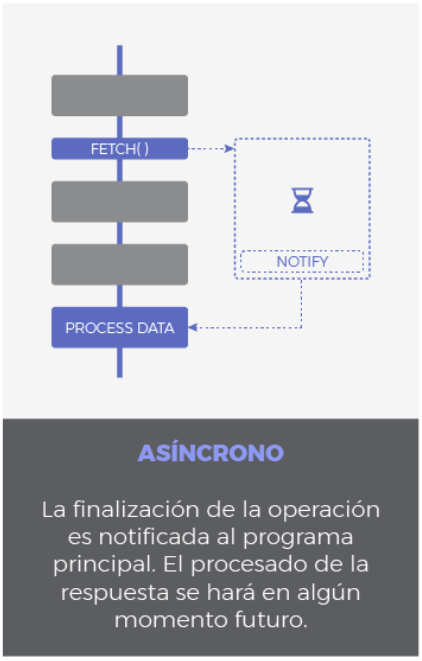
- **Bloqueante**: Una llamada u operación bloqueante no devuelve el control a nuestra aplicación hasta que se ha completado. Por tanto, el *thread* queda bloqueado en estado de espera.
- **No Bloqueante**: Una llamada no bloqueante devuelve inmediatamente con independencia del resultado. En caso de que se haya completado, devolverá los datos solicitados. En caso contrario, podría devolver un código de error.



Síncrono vs Asíncrono se refiere a cuando tendrá lugar la respuesta:

- **Síncrono**: es frecuente emplear 'bloqueante' y 'síncrono' como sinónimos, dando a entender que toda la operación de entrada/salida se ejecuta de forma secuencial y, por tanto, debemos esperar a que se complete para procesar el resultado.
- **Asíncrono**: la finalización de la operación *I/O* se señaliza más tarde, mediante un mecanismo específico como por ejemplo un *callback*, una *promesa* o un *evento*, lo que hace posible que la respuesta sea procesada en diferido. Como se puede adivinar, su comportamiento es no bloqueante ya que la llamada *I/O* devuelve inmediatamente.



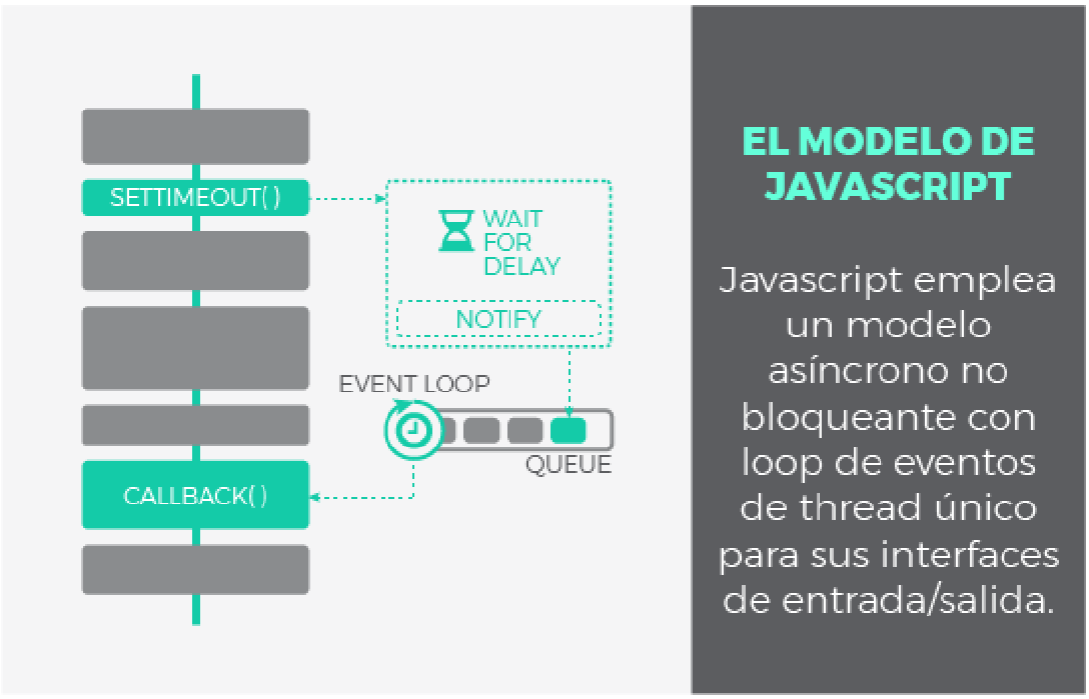


Según la clasificación anterior, podemos tener operaciones *I/O* de tipo:

- Síncronas y Bloqueantes. Toda la operación se hace de una vez, bloqueando el flujo de ejecución:
- Síncronas y No-Bloqueantes. Similar a la anterior, pero usando alguna técnica de *polling* para evitar el bloqueo en la primera fase:
- Asíncronas y No-Bloqueantes:
    1. La petición devuelve inmediatamente para evitar el bloqueo.
    2. Se envía una notificación una vez que la operación se ha completado. Es entonces cuando la función que procesará la respuesta (*callback*) se encola para ser ejecutada en algún momento en nuestra aplicación.

El Modelo de Javascript

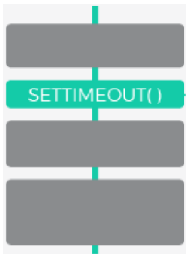
- Javascript fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones de usuario, al tiempo que se mantiene una interfaz fluida.
- Javascript utiliza un modelo **asíncrono y no bloqueante**, con un ***loop* de eventos implementado con un único *thread*** para sus interfaces de entrada/salida.
- Gracias a esta solución, Javascript es áltamente concurrente a pesar de emplear un único *thread*.
- A continuación, se muestra el aspecto de una operación *I/O* asíncrona en Javascript:



Paso a paso, podría explicarse del siguiente modo:

UNO

Petición de operación 1/0. Su naturaleza no bloqueante hace que devuelva inmediatamente El flujo del programa no se bloquea y puede continuar ejecutando tareas mientras se completa la operación asíncrona.



DOS

Se produce un cambio de contexto, la operación real se procesa fuera de nuestra aplicación. (ejemplo: esperar timer, recuperar datos, etc.). E sistema operativo es e responsable

TRES

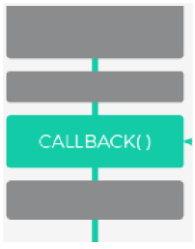
El final de la operación se señaliza asíncronamente. Una notificación en forma de mensaje se encola en la lista de mensajes pendientes a ser procesados por el entorno Javascript.

CUATRO

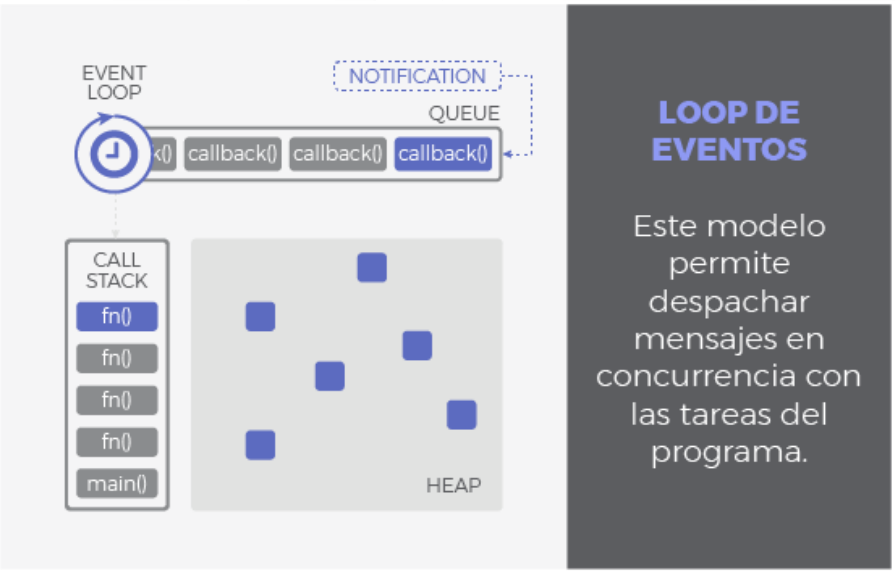
El loop de eventos es e mecanismo a cargo de procesar un mensaje cada vez. Cada mensaje debe esperar su turno.

CINCO

Una vez procesado el mensaje, su función asociada (callback) se programa para ser ejecutada. E callback hará lo que sea que queramos hacer como respuesta a la operación de entrada/salida (ejemplo: consumir datos o confirmar la operación).

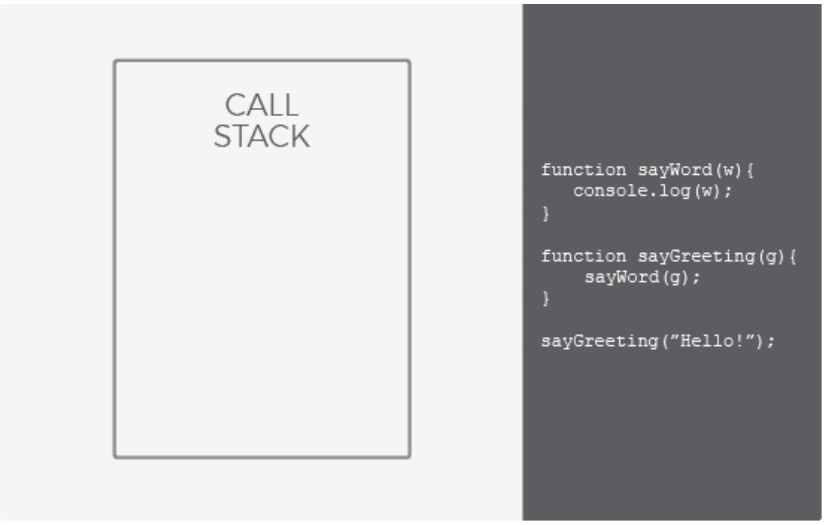


El Loop de Eventos de Javascript



Call Stack

- Se encarga de albergar las instrucciones que deben ejecutarse.
- Nos indica en que punto del programa estamos, por donde vamos.
- Cada llamada a una función de nuestra aplicación entra a la pila generando un nuevo *frame* (bloque de memoria reservada para los argumentos y variables locales de dicha función).
- Por tanto, cuando se llama a una función, su *frame* es insertado arriba en la pila, cuando una función se ha completado y devuelve, su *frame* se saca de la pila también por arriba.
- El funcionamiento es LIFO: *last in, first out*.



Heap

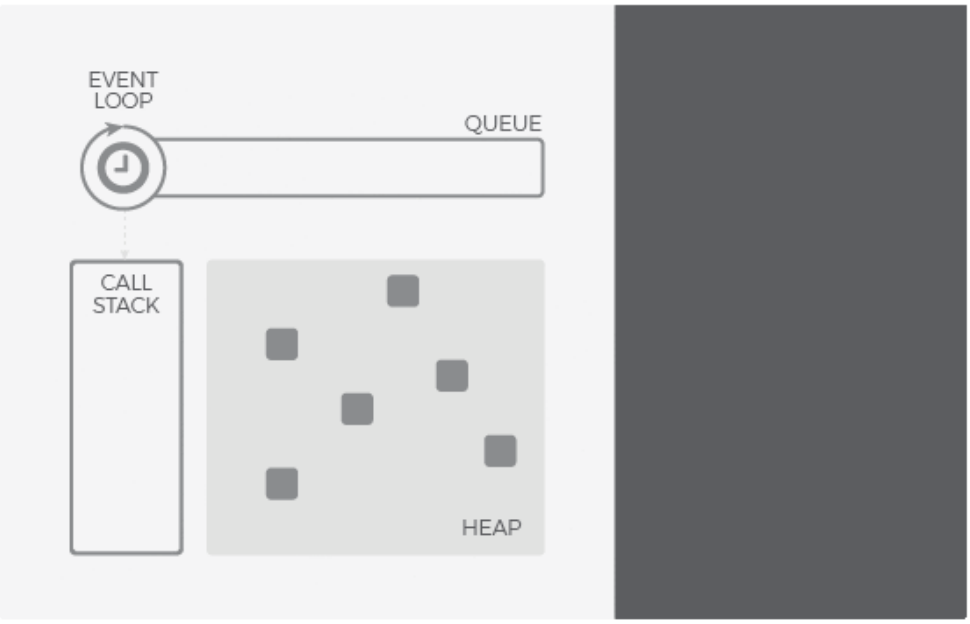
Región de memoria libre, normalmente de gran tamaño, dedicada al alojamiento dinámico de objetos.

Cola o Queue

Cada vez que nuestro programa recibe una notificación del exterior o de otro contexto distinto al de la aplicación (como es el caso de operaciones asíncronas), el mensaje se inserta en una cola de mensajes pendientes y se registra su *callback* correspondiente. Recordemos que un *callback* era la función que se ejecutará como respuesta.

Loop de Eventos

Cuando la pila de llamadas (*call stack*) se vacía, es decir, no hay nada más que ejecutar, se procesan los mensajes de la cola.



Entonces…

"La cola, es el almacén de los mensajes (notificaciones) y sus *callbacks* asociados mientras que **el *loop* de eventos es el mecanismo para despacharlos** que sigue un comportamiento síncrono."

"Los *callbacks* no son despachados tan pronto como sean encolados, por lo cual, la finalización de una operación asíncrona no puede predecirse con seguridad, sino que se atiende en modo *best effort*."

Patrones asíncronos en Javascript

Callbacks

- Los *callbacks* son la pieza clave para que Javascript pueda funcionar de forma asíncrona.
- Un *callback* no es más que **una función que se pasa como argumento de otra función**, y que será invocada para completar algún tipo de acción.
- Es el trozo de código que será ejecutado una vez que una operación asíncrona notifique que ha terminado.
- Los *callbacks* también pueden lanzar a su vez llamadas asíncronas, así que pueden anidarse tanto como se desee.

Promesas

Una promesa es un objeto que representa **el resultado de una operación asíncrona**.
Las promesas se basan en ***callbacks*** y son especiales en términos de asincronía ya que añaden un nuevo nivel de prioridad.

Async / Await

Las promesas supusieron un gran salto en Javascript al introducir una mejora sustancial sobre los *callbacks* y un manejo más elegante de nuestras tareas asíncronas.
***async* y *await* surgieron para simplificar el manejo de las promesas**. Hacen las promesas más amigables, escribir código más sencillo, reducir el anidamiento y mejorar la trazabilidad al depurar.
Sin embargo, *async* \ *await* y las promesas son lo mismo en el fondo.
La etiqueta *async* declara una función como asíncrona e indica que una promesa será automáticamente devuelta.
Por otro lado, *await* debe ser usado siempre dentro de una función declarada como *async* y esperará automáticamente (de forma asíncrona y no bloqueante) a que una promesa se resuelva.

Resumen

- La concurrencia hace que las tareas progresen simultáneamente. El paralelismo es un caso especial de concurrencia donde las tareas se ejecutan literalmente al mismo tiempo.
- Estas tareas pueden consumir CPU de forma intensiva. Se las conoce como operaciones *CPU-bound* y llevan código que se ejecuta en nuestra aplicación. Al contrario, las operaciones *I/O-bound* no son ejecutadas en el flujo de nuestro programa sino en un contexto externo. Estas operaciones persiguen el acceso a dispositivos o recursos como servidores, bases de datos, ficheros, etc.
- Las operaciones *I/O-bound* (de entrada/salida) pueden ser bloqueantes o no bloqueantes, en función de si el *thread* queda a la espera o no, y síncronas o asíncronas, según si la ejecución es secuencial o la respuesta puede darse en diferido, en algún momento en el futuro.
- Javascript está diseñado para aplicaciones web, enfocado hacia operaciones *I/O-bound*. Utiliza un modelo asíncrono y no bloqueante con un *loop* de eventos de un único *thread*.
- Este modelo permite despachar mensajes asíncronos de forma concurrente, pero cuidado, si no se utiliza convenientemente podemos reducir considerablemente el desempeño de nuestra aplicación. Mantener los *callbacks* todo lo ligeros que te sea posible, es necesario.
- Para aquellas tareas pesadas que requieran un procesamiento intensivo utiliza el paralelismo en Javascript a través de los WebWorkers.
- Los patrones asíncronos más comunes en Javascript son:
    1. *Callback*. Función que se ejecuta cuando una operación asíncrona termina, como resultado de esta.
    2. Promesa. Representa el resultado de una operación asíncrona. Se configura con dos *callbacks* para resolver la promesa con éxito o con fallo.
    3. Async/Await. Permite manejar promesas de una forma más simple. *Async* declara una función como asíncrona mientras que *await* gestiona la resolución de una promesa de forma automática. *Await* debe emplearse siempre dentro de declaraciones *async*. Atento a múltiples *await*, piensa bien el comportamiento que necesitas.