08. CALLBACKS JAVASCRIPT

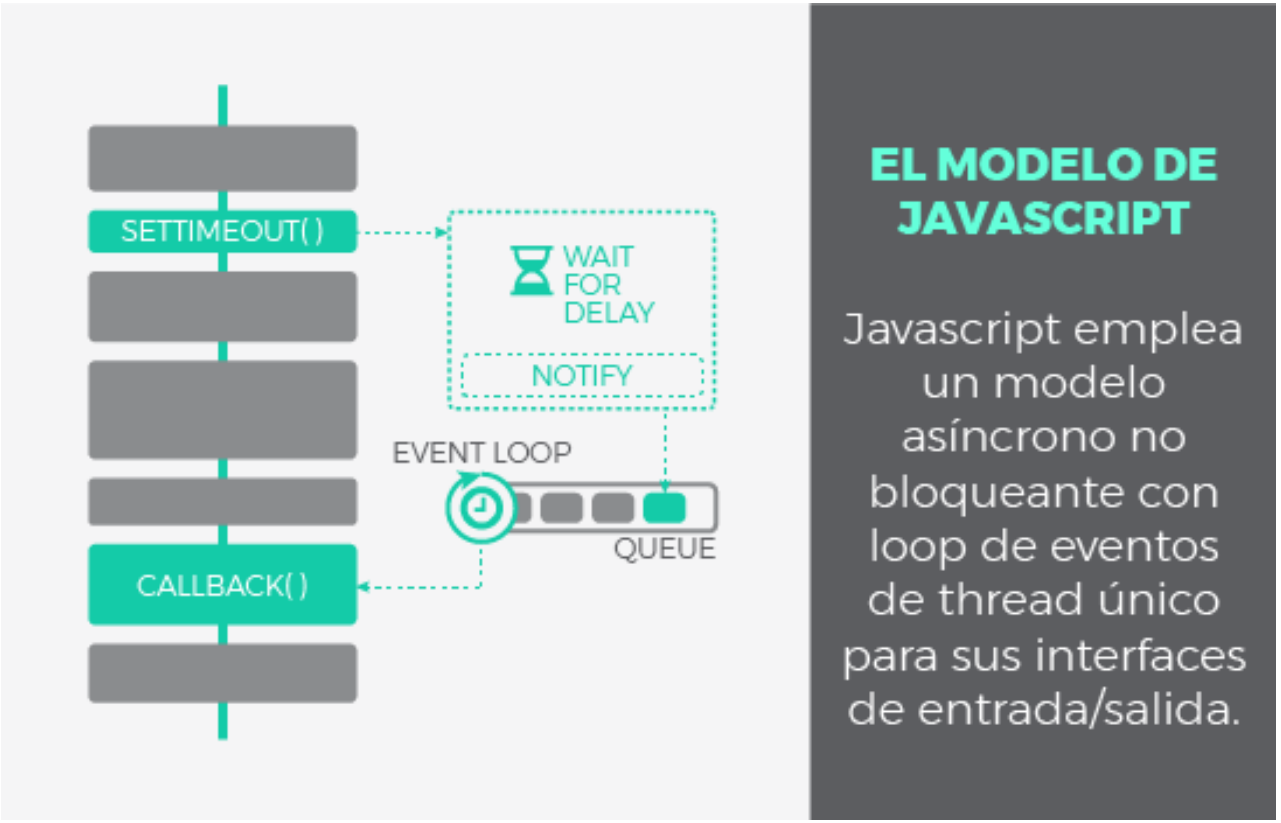
IES ESTACIÓ CURS 2021- 2022

Índex

1. Introducció a la Asíncronia en JavaScript.

- Javascript va ser dissenyat per a ser executat en navegadors, treballar amb peticions sobre la xarxa i processar les interaccions d'usuari, al mateix temps que es manté una interfície fluida.
- Ser bloquejant o síncron no ajudaria a aconseguir aquests objectius, és per això que Javascript ha evolucionat intencionadament pensant en operacions de tipus I/O.
- Per aquesta raó, Javascript utilitza un model asíncron i no bloquejant, amb un loop d'esdeveniments implementat amb un únic thread per a les seues interfícies de E/S.
- Gràcies a aquesta solució, Javascript és altament concurrent malgrat emprar un únic thread.
- Vegem l'aspecte d'una operació I/O asíncrona en Javascript:

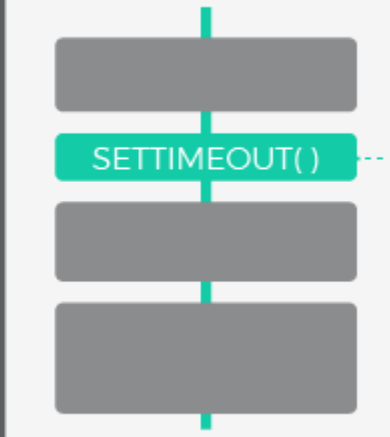
1. Introducció a la Asíncronia en JavaScript.



1. Introducció a la Asíncronia en JavaScript.

UNO

Petición de operación I/O. 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.



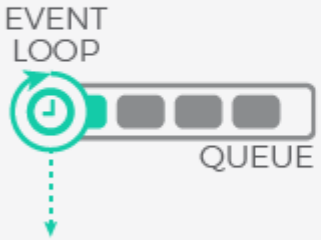
DOS

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

1. Introducció a la Asíncronia en JavaScript.

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 el mecanismo a cargo de procesar un mensaje cada vez. Cada mensaje debe esperar su turno.

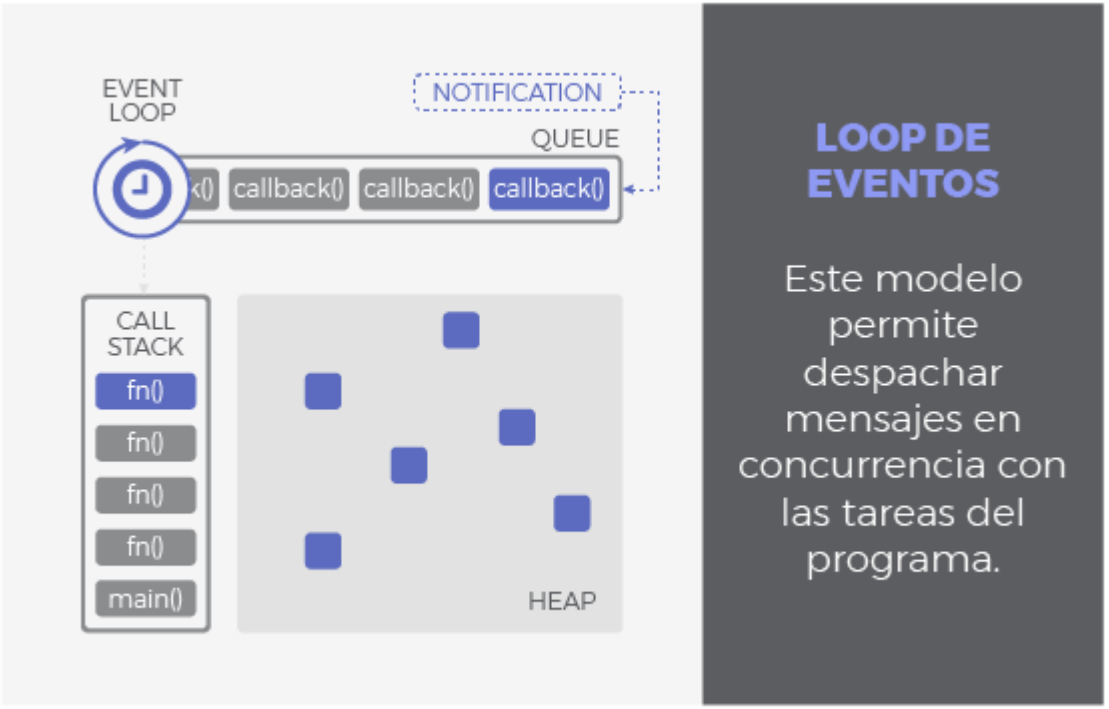
1. Introducció a la Asíncronia en JavaScript.

CINCO

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

2. El Loop de esdeveniments.

- ¿Com s'executa un programa en Javascript? ¿Com gestiona la nostra aplicació de manera concurrent les respostes a les crides asíncrones?
- Això és exactament el que el model basat en un loop d'esdeveniments ve a respondre:



2. El Loop de esdeveniments.

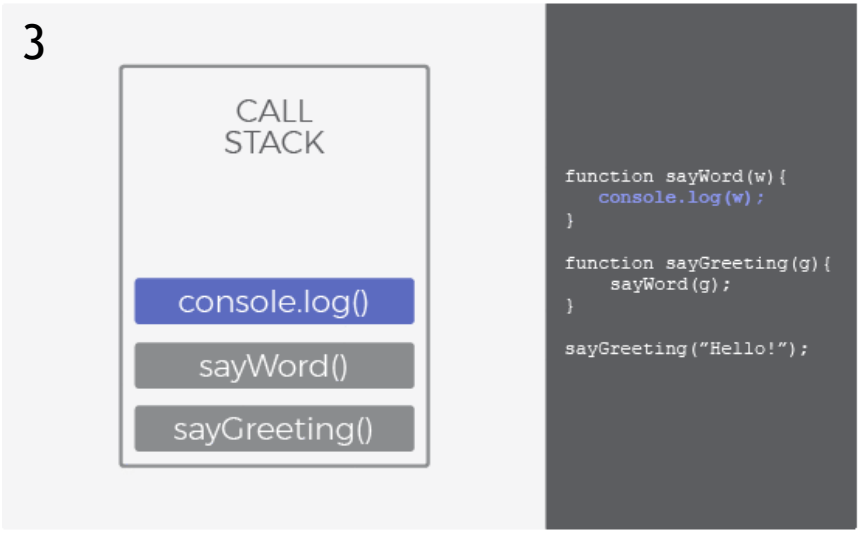
- **Call Stack.**
  - S'encarrega d'albergar les instruccions que han d'executar-se.
  - Ens indica en què punt del programa estem, per on anem.
  - Cada cridada a una funció de la nostra aplicació, entra a la pila generant un nou frame (bloc de memòria reservada per als arguments i variables locals d'aquesta funció).
  - Per tant, quan es crida a una funció, la seua frame és inserida a dalt en la pila, quan una funció s'ha completat i retorna, la seua frame es trau de la pila també per dalt.
  - El funcionament és **LIFO**: last in, first out.
  - D'aquesta manera, les cridades a funció que estan dins d'una altra funció contenidora són apilades damunt i seran ateses primer.

2. El Loop de esdeveniments.

1

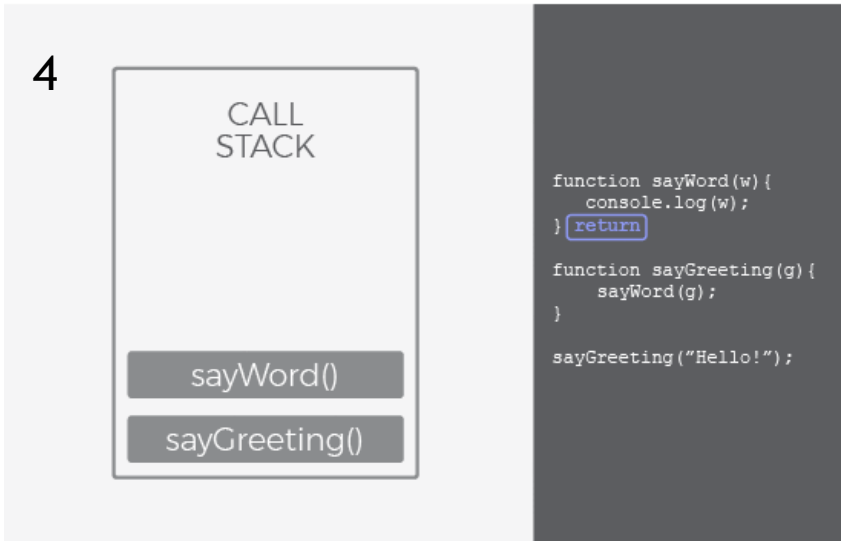
2

3



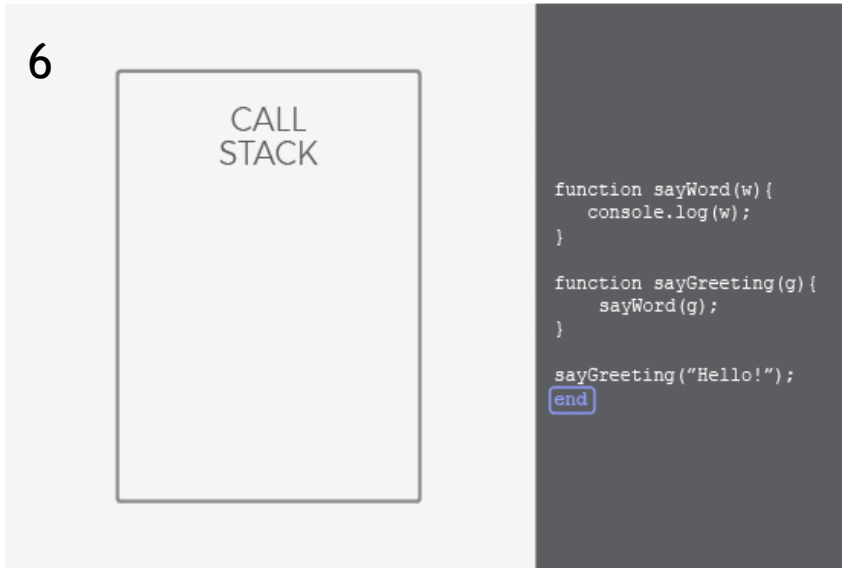
2. El Loop de esdeveniments.

4



5

6



2. El Loop de esdeveniments.

- **Heap.**
  - Regió de memòria lliure, normalment de gran grandària, dedicada a l'allotjament dinàmic d'objectes.
  - És compartida per tot el programa i controlada per un recol·lector de fem que s'encarrega d'alliberar allò que no es necessita.
- **Queue.**
  - Cada vegada que el nostre programa rep una notificació de l'exterior o d'un altre context diferent del de l'aplicació (com és el cas d'operacions asíncrones), el missatge s'insereix en una cua de missatges pendents i es registra el seu **callback** corresponent.
  - Recordem que un **callback** era la funció que s'executarà com a resposta.

2. El Loop de esdeveniments.

- **Loop d'Esdeveniments.**
  - Quan la pila de crides (call stack) està buida, és a dir, no hi ha res més a executar, es processen els missatges de la cua.
  - Amb cada 'tick' del bucle d'esdeveniments, es processa un nou missatge.
  - Aquest processament consisteix a cridar al **callback** associat a cada missatge el que donarà lloc a un nou frame en la pila de crides.
  - Aquest frame inicial pot derivar en molts més, tot depén del contingut del **callback**.
  - Un missatge s'acaba de processar quan la pila torna a estar buida de nou.
  - A aquest comportament se'l coneix com a 'run-to-completion’.

2. El Loop de esdeveniments.

- **Loop d'Esdeveniments.**
  - D'aquesta manera, podem entendre la cua com el magatzem dels missatges (notificacions) i les seues callbacks associats.
  - El loop d'esdeveniments és el mecanisme per a despatxar-los.
  - Aquest mecanisme segueix un comportament síncron: cada missatge ha de ser processat de manera completa perquè puga començar el següent.
  - Una de les implicacions més rellevants d'aquest bucle d'esdeveniments és que els **callbacks** no seran despatxats tan prompte com siguen encolats, sinó que han d'esperar el seu torn.
  - Aquest temps d'espera dependrà del número de missatges pendents de processar (per davant en la cua) així com del temps que es tardarà en cadascun d'ells.
  - Encara que puga semblar obvi, això explica la raó per la qual la finalització d'una operació asíncrona no pot predir-se amb seguretat, sinó que s'atén en manera “best effort”.

2. El Loop de esdeveniments.

- **Loop d'Esdeveniments.**
  - El loop d'esdeveniments no està lliure de problemes, i podrien donar-se situacions compromeses en els següents casos:
    - La pila de crides no es buida, ja que la nostra aplicació fa ús intensiu d'ella. No hi haurà tick en el bucle d'esdeveniments i per tant els missatges no es processen.
    - El flux de missatges que es van encolant és major que el de missatges processats. Massa esdeveniments alhora.
    - Un callback requereix processament intensiu i acapara la pila. De nou bloquegem els ticks del bucle d'esdeveniments i la resta de missatges no es despatxen.

2. El Loop de esdeveniments.

- **Loop d'Esdeveniments.**
  - El més probable és que un coll de botella es produïsca a conseqüència d'una mescla de factors.
  - En qualsevol cas, acabarien retardant el flux d'execució.
  - I per tant retardant el renderitzat, el processament d'esdeveniments, etc.
  - L'experiència d'usuari es degradaria i l'aplicació deixaria de respondre de manera fluida.
  - Per a evitar aquesta situació, recorda sempre mantenir els **callbacks** tan lleugers com siga possible.
  - En general, evita codi que acapare la CPU i permeta que el loop d'esdeveniments s'execute a bon ritme.

3. Callbacks.

- Els **callbacks** són la peça clau perquè Javascript puga funcionar de manera asíncrona.
- La resta de patrons asíncrons en Javascript està basat en **callbacks** d'una manera o un altre.
- Un no és més que una funció que es passa com a argument d'una altra funció, i que serà invocada per a completar algun tipus d'acció.
- En el nostre context asíncron, un **callback** representa el ¿Què vols fer una vegada que la teua operació asíncrona acabe?
- Per tant, és el tros de codi que serà executat una vegada que una operació asíncrona notifique que ha acabat.
- Aquesta execució es farà en algun moment futur, gràcies al mecanisme que implementa el bucle d'esdeveniments.

3. Callbacks.

- Fixa't en el següent exemple senzill utilitzant un callback:

```
setTimeout(function(){
  console.log("Hola Mundo con retraso!");
}, 1000)
```

- Si ho prefereixes, el **callback** pot ser assignat a una variable amb nom en lloc de ser anònim:

```
const myCallback = () => console.log("Hola Mundo con retraso!");
setTimeout(myCallback, 1000);
```

- **setTimeout** és una funció asíncrona que programa l'execució d'un **callback** una vegada ha transcorregut, com a mínim, una determinada quantitat de temps (1 segon en l'exemple anterior).
- A tal fi, dispara un **timer** en un context extern i registra el **callback** per a ser executat una vegada que el **timer** acabe.
- En resum, retarda una execució, com a mínim, la quantitat especificada de temps.

3. Callbacks.

- És important comprendre que, fins i tot si configurem el retard com 0ms, no significa que el **callback** vaja a executar-se immediatament.
- Exemple:

```
setTimeout(function(){
  console.log("Esto debería aparecer primero");
}, 0);
console.log("Sorpresa!");

// Sorpresa!
// Esto debería aparecer primero
```

- Recorda, un **callback** que s'afig al **loop d'esdeveniments** ha d'esperar el seu torn.
- En l'exemple, el **callback** del **setTimeout** ha d'esperar el primer **tick**.
- No obstant això, la pila està ocupada processant la línia **console.log("Sorpresa!").**
- El **callback** es despatxarà una vegada la pila quede buida, en la pràctica, quan Sorpresa! Haja sigut logueado.

3.1 Callback Hell.

- Els **callbacks** també poden llançar al seu torn crides asíncrones, d'esta forma que poden anidar-se tant com es desitge.
- **Inconvenient**, podem acabar amb codi com aquest:

```
setTimeout(function(){
  console.log("Etapa 1 completada");
  setTimeout(function(){
    console.log("Etapa 2 completada");
    setTimeout(function(){
      console.log("Etapa 3 completada");
      setTimeout(function(){
        console.log("Etapa 4 completada");
        // Podríamos continuar hasta el infinito...
      }, 4000);
    }, 3000);
  }, 2000);
}, 1000);
```

- Aquest és un dels inconvenients clàssics dels **callbacks**, a més de la indentació, resta llegibilitat, dificulta el seu manteniment i afig complexitat ciclomática.
- Al **Callback Hell** també se'l coneix com **Pyramid of Doom** o **Hadouken**.