CLASS(E)

Módulo 5. Asincronía I

Callbacks

Callbacks

Otra técnica que desbloquea el uso de funciones de orden superior es el uso de callbacks.

Los callbacks son funciones que reaccionan a un evento. Por ejemplo:

- Al recibir la respuesta a una petición HTTP
- Al detectar la acción de un usuario (onClick, onKeyUp, etc.)
- Al pasar un intervalo de tiempo

Callbacks

En definitiva, las callbacks nos permiten reaccionar a eventos que no sabemos cuándo van a suceder.

Es decir, eventos asíncronos.

Callbacks

Por la propia naturaleza del entorno web, la asincronía es inescapable para los programadores de JavaScript.

Intervalos

Intervalos

Javascript dispone de dos herramientas para gestionar intervalos de tiempo:

- setTimeout
- setInterval

setTimeout

Ejecuta una función (llamada callback function) al cabo de x milisegundos.

`setTimeout(callback, ms)`

Ejemplos →

```
setTimeout(function(){
     console.log("Ha pasado un segundo")
}, 1000)

setTimeout(function(){
     console.log("Ha pasado medio segundo")
}, 500)
```

setTimeout

¿En qué orden se imprimen los números?

```
console.log("uno")

setTimeout(function(){
     console.log("dos")
}, 1000)

console.log("tres")
```

setTimeout

Genera asincronía: se ejecutan dos ramas de código por separado.

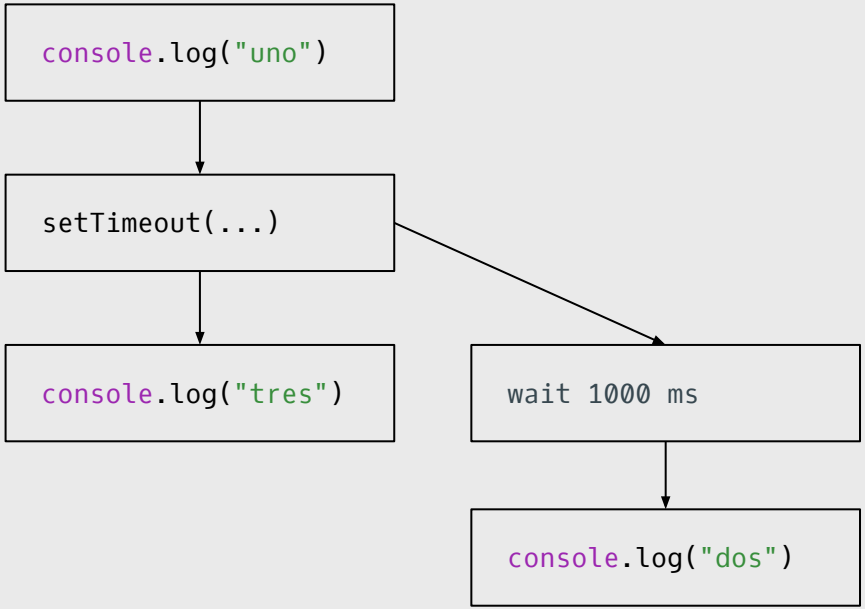
```
console.log("uno")

setTimeout(function(){
     console.log("dos")
}, 1000)

console.log("tres")

// uno
// tres
// dos (al cabo de un segundo)
```

setTimeout




```
console.log("uno")

setTimeout(function(){
     console.log("dos")
}, 1000)

console.log("tres")

// uno
// tres
// dos (al cabo de un segundo)
```

setInterval

Ejecuta una función cada x milisegundos.

```
setInterval(function, ms)
```

```
setInterval(function(){
    console.log("Me llamo cada vez que pasan dos segundos")
}, 2000)
```

Ejercicio intervalos

Crea una variable contador que empiece valiendo `0`.

Incrementa el contador cada segundo e imprímelo.

Ejercicio intervalos II

Imprime ‘ping’ cada 500ms de forma infinita.

No utilices setInterval

setTimeout

Genera asincronía: se ejecutan dos ramas de código por separado.

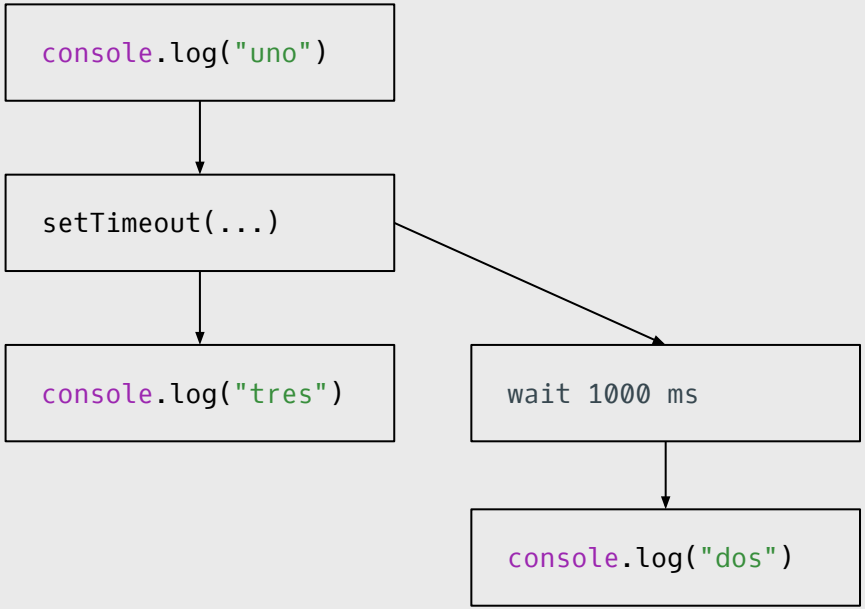
```
console.log("uno")

setTimeout(function(){
    console.log("dos")
}, 1000)

console.log("tres")

// uno
// tres
// dos (al cabo de un segundo)
```

setTimeout




```
console.log("uno")

setTimeout(function(){
     console.log("dos")
}, 1000)

console.log("tres")

// uno
// tres
// dos (al cabo de un segundo)
```

Callbacks

¿Cómo ordenamos los logs?

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

Callbacks

¿Cómo ordenamos los logs?

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

Callbacks

¿Cómo imprimimos “tres” un segundo después de “dos”?

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

Callbacks

¿Cómo imprimimos “tres” un segundo después de “dos”?

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

Ejercicio asincronía I

Crea una función `throwDice` que devuelva el resultado de tirar un dado de 6 caras al cabo de 1000ms.

- La función devuelve el resultado a través de un callback.

Ejercicio asincronía II

Crea una función `getPlayerScore` que devuelva el resultado de tirar dos datos (eg: [3,5]).

- Utiliza la función `throwDice` del ejercicio anterior para calcular los valores de las tiradas.
- Devuelve el resultado utilizando un callback.

Ejercicio asincronía III

Crea una función startGame que devuelva los resultados de las tiradas de 3 jugadores (eg: [[2, 2], [4, 6], [5, 1]]).

- No utilices bucles.
- Utiliza la función `getPlayerScore` del ejercicio anterior para calcular los valores de las tiradas.
- Devuelve el resultado utilizando un callback.

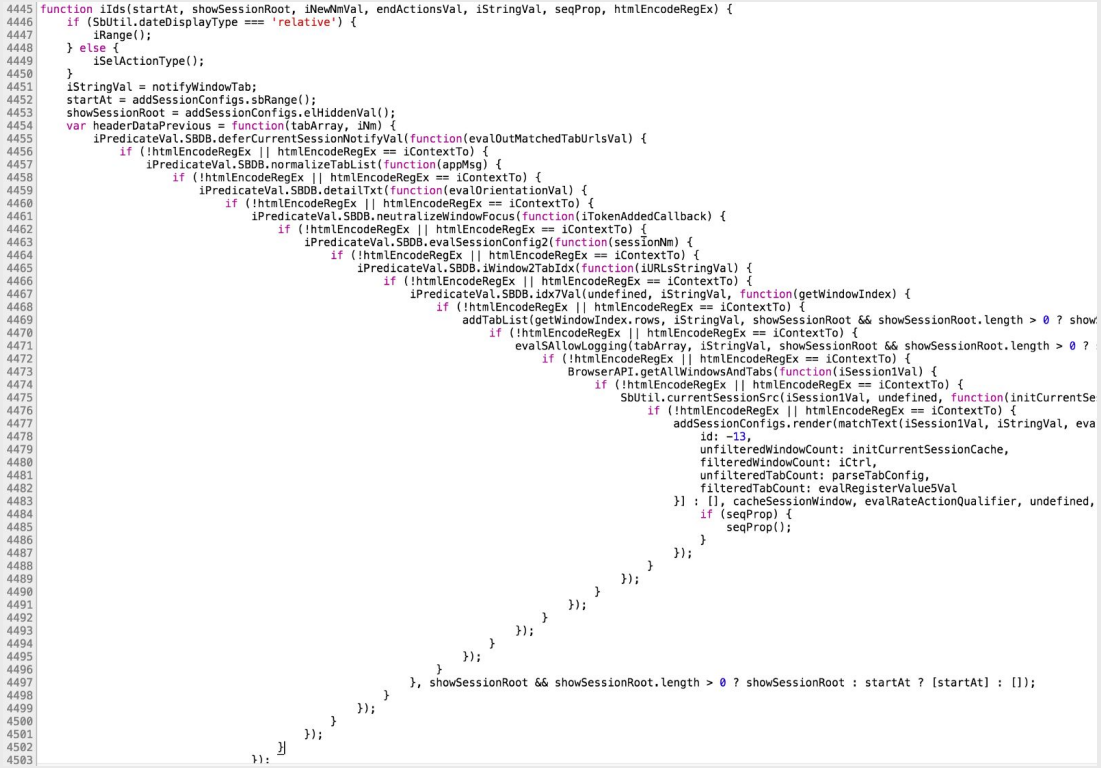
Conclusión: ¡la asincronía es contagiosa!

Callbacks

¿Cuál es el problema?

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

Callback Hell



```
function iIds(startAt, showSessionRoot, iNewNmVal, endActionsVal, iStringVal, seqProp, htmlEncodeRegEx) {
    if (SbUtil.dateDisplayType === 'relative') {
        iRange();
    } else {
        iSelActionType();
    }
    iStringVal = notifyWindowTab;
    startAt = addSessionConfigs.sbRange();
    showSessionRoot = addSessionConfigs.elHiddenVal();
    var headerDataPrevious = function(tabArray, iNm) {
        iPredicateVal.SBDB.deferCurrentSessionNotifyVal(function(evalOutMatchedTabUrlsVal) {
            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                iPredicateVal.SBDB.normalizeTabList(function(appMsg) {
                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                        iPredicateVal.SBDB.detailTxt(function(evalOrientationVal) {
                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                iPredicateVal.SBDB.neutralizeWindowFocus(function(iTokenAddedCallback) {
                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                        iPredicateVal.SBDB.evalSessionConfig2(function(sessionNm) {
                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                iPredicateVal.SBDB.iWindow2TabIdx(function(iURLsStringVal) {
                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                        iPredicateVal.SBDB.idx7Val(undefined, iStringVal, function(getWindowIndex) {
                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                addTabList(getWindowIndex.rows, iStringVal, showSessionRoot && showSessionRoot.length > 0 ? show
                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                        evalSAllowLogging(tabArray, iStringVal, showSessionRoot && showSessionRoot.length > 0 ?
                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                BrowserAPI.getAllWindowsAndTabs(function(iSession1Val) {
                                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                        SbUtil.currentSessionSrc(iSession1Val, undefined, function(initCurrentSe
                                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                                addSessionConfigs.render(matchText(iSession1Val, iStringVal, eva
                                                                                                    id: -13,
                                                                                                    unfilteredWindowCount: initCurrentSessionCache,
                                                                                                    filteredWindowCount: iCtrl,
                                                                                                    unfilteredTabCount: parseTabConfig,
                                                                                                    filteredTabCount: evalRegisterValue5Val
                                                                                                }] : [], cacheSessionWindow, evalRateActionQualifier, undefined,
                                                                                                    if (seqProp) {
                                                                                                        seqProp();
                                                                                                    }
                                                                                                });
                                                                                            }
                                                                                        });
                                                                                    }
                                                                                });
                                                                            }
                                                                        });
                                                                    }
                                                                });
                                                            }
                                                        });
                                                    }
                                                }, showSessionRoot && showSessionRoot.length > 0 ? showSessionRoot : startAt ? [startAt] : []);
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
```

Una posible solución es la paralelización

Ejercicio asincronía IV 🔥

Crea una función startGame que reciba un número `N` de jugadores y devuelva un array con los resultados de N jugadores (eg: [[2, 2], [4, 6], [5, 1], [4, 3]]).

- Utiliza la función `getPlayerScore` del ejercicio anterior para calcular los valores de las tiradas.
- Devuelve el resultado utilizando un callback.
- Puedes llamar varias funciones getPlayerScore en paralelo.

A veces no es posible paralelizar...

Gestión de errores

Gestión de errores

Javascript tiene varios errores nativos.

Error name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number “out of range” has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

Gestión de errores

Podemos lanzar errores manualmente mediante throw.

Se parará la ejecución del código y se mostrará el error en consola.

```
throw SyntaxError
```

Gestión de errores

Se puede throwear cualquier valor.

Se parará la ejecución del código y se mostrará el valor en consola como error.

```
throw value // String, Number, Boolean or Object
```

Gestión de errores

Podemos instanciar nuestros propios objetos de error.

```
const error = new Error("Custom error 28031")
console.log(error.message) // Custom error 28031

// Podemos lanzar el nuevo objeto de error
throw error
```

Gestión de errores

Podemos evitar que un error pare la ejecución de código mediante un bloque try/catch.

```
try {
  console.log("lorem ipsum")
} catch(err) {
// El interior de este bloque se ejecutará en caso de error
  console.error(err)
}
```

Gestión de errores

Podemos rethrowear el error condicionalmente si sólo queremos que determinados errores paren la ejecución.

```
function getUser(id) {
    try {
        return readDB(id) // librería externa
    } catch(err) {
        if(err.message instanceof KeyDoesNotExist) {
            throw new Error("Missing user.", {cause: err})
        }

        throw new Error("Error reading database", {cause: err})
    }
}
```

Gestión de errores

El bloque `finally` nos permite ejecutar código haya o no error.

```
try {
    socket.read()
} catch(err) {
    console.error(err)
} finally {
    // El socket se cerrará en cualquier caso
    socket.close()
}
```

En el código asíncrono la gestión de errores se complica

Errores en asincronía

¿Qué pasa si ejecutamos este código?

```
try {
    setTimeout(() => {
        console.log("Esto debería fallar")
    }, 500)
} catch(err) {
    console.log("Ha habido un error!")
}
```

Errores en asincronía

El catch nunca va a capturar el error.

Primero se evalúa el try/catch entero y luego, en otra rama, se evalúa el callback del setTimeout.

```
try {
    setTimeout(() => {
        console.olg("Esto debería fallar")
    }, 500)
} catch(err) {
    console.log("Ha habido un error!")
}
```

¿Cómo se gestionan los errores con callbacks?

Errores en asincronía

En programación por callbacks, el error se pasa al propio callback.

El consenso en las librerías de JavaScript es pasar el error como primer parámetro.

```
const fs = require('fs')

fs.readdir('.', (err, files) => {
  if (err)
    console.error(err)
  else
    console.log(files)
})
```

Errores en asincronía

Si err es null, la operación ha sido exitosa.

```
const fs = require('fs')

fs.readdir('.', (err, files) => {
  if (err)
    console.error(err)
  else
    console.log(files)
})
```

Ejercicio filesystem 🔥

Escribe una función `calculateDirSize` en `node` que…

- Reciba un directorio como parámetro.
- Calcule la suma total del tamaño de los ficheros que contiene.
- `NO` utilice promesas, async/await o métodos sync de fs.

Puedes utilizar los módulos fs y path

- Información en la siguiente diapositiva →

```
calculateDirSize("./folder", (err, size) => {
  console.log('Total:', (size / 1024).toFixed(2), 'Kb')
})
```

Ejercicio filesystem 🔥

fs.readdir

- Lista de los elementos de un directorio.

path.join

- Concatenar segmentos de una ruta

fs.stat (información sobre un fichero)

- Consultar si una ruta es directorio o fichero.
- Consultar el tamaño de un fichero.

Ejercicio asyncMap 🔥🔥🔥

Crea una función `asyncMap`:

- Recibe una lista y devuelve el resultado de aplicarles una función asíncrona, de las que devuelven el resultado por callback.
- asyncMap devuelve una lista con todos los resultados por callback una vez estén todos calculados.
- **Cuidado**: ¡la función que reciba asyncMap debe llamar al callback siempre! Si no, asyncMap nunca devolverá un resultado. Cuidado con ifs que no llamen al callback.

Ejercicio filesystem II 🔥

Utiliza la función `asyncMap` del ejercicio anterior para reimplementar `calculateDirSize`.

Ejercicio filesystem III

Modifica `calculateDirSize` para que calcule el tamaño de una carpeta y `todos sus subdirectorios` recursivamente.

Asincronía y funciones de orden superior

Si combinamos asincronía y funciones de orden superior podemos resolver fácilmente algunos problemas habituales en el desarrollo web.

Throttle

throttle nos permite transformar una función para que sólo pueda ser llamada una vez cada x milisegundos.

```
function spam(){
  console.log("SPAM!")
}

const throttledSpam = throttle(spam, 500)

throttledSpam() // SPAM!
throttledSpam() // no effect
throttledSpam() // no effect
setTimeout(throttledSpam, 600) // SPAM!
```

Throttle

throttle se utiliza bastante en front.

Ejemplo: Para evitar llamar a un event listener de window.resize o scroll 500 veces cuando el usuario interactúa con la página.

Ejercicio throttle

Implementa la función `throttle`.

Debounce

debounce nos permite transformar una función para que se ejecute pasados x milisegundos desde la última vez que fue llamada.

Si llamamos a la función otra vez antes de que se ejecute, se reinicia el temporizador.

```
function spam(){
  console.log("SPAM!")
}

const debouncedSpam = debounce(spam, 500)

debouncedSpam()
debouncedSpam()
debouncedSpam()
// A los 500 ms -> SPAM!

setTimeout(debouncedSpam, 600) // A los 500 ms -> SPAM!
```

Debounce

Debounce se utiliza mucho para evitar la sobrecarga de peticiones a un servidor.

Ejemplo: una web ofrece un buscador que filtra resultados en tiempo real. Es muy costoso pedir nuevos resultados al servidor por cada letra que escriba cada usuario. Debounce nos permite enviar una petición sólo cuando se ha dejado de escribir.

Ejercicio debounce

Implementa la función debounce.

¿Qué pasa si queremos múltiples reacciones a un evento?

Observables

Observables

Sabemos que podemos suscribir varias reacciones en Javascript.

```
window.addEventListener("resize", () => console.log("reaction1"))
window.addEventListener("resize", () => console.log("reaction2"))
```

Observables

```
const onResize = (ev) => console.log("resizing")
window.addEventListener("resize", onResize)
window.removeEventListener("resize", onResize)
```

También podemos desuscribirlas.

¿Cómo funciona esto?

Observables

El patrón observable nos permite:

- Tradicionalmente (OOP)
  - Desacoplar objetos dependientes.
- Javascript
  - Suscribir varios callbacks a un mismo agente.

Observables

Un productor.

Múltiples consumidores.

Se comunican mediante eventos.

Observables

Un observable tiene tres métodos:

- `subscribe` (a.k.a on) → asigna un callback a un evento.
- `unsubscribe` (a.k.a off) → desasigna un callback a un evento.
- `emit` → ejecuta todos los callbacks asignados al evento.

Ejercicio observables

Crea un nuevo fichero "./eventManager.js" que actúe como un observable.

- Guarda el estado sobre los callbacks asignados en ese fichero.
- Exporta un objeto con tres funciones: on, off, emit.

```
const eventManager = require("./eventManager")

eventManager.on("detonate", () => console.log("explosion"))
eventManager.on("detonate", () => console.log("smoke"))
eventManager.emit("detonate")
// explosion
// smoke
```

Ejercicio observables II

No deberíamos poder detonar una bomba más de una vez.

Añade un método once al eventManager que suscriba un callback que se ejecutará sólo la primera vez que se emita ese evento.

Observables

Los observables nos permiten desacoplar componentes.

Esta es la premisa principal de la arquitectura orientada a eventos.

Observables

En el ejercicio anterior hemos implementado un observable en forma de singleton.

Lo más habitual es utilizar clases.

Observables

Crear observables mediante clases es muy sencillo: extendemos de una clase Observable.

```
class Metronome extends Observable {
  constructor(tempo){
    super()
    this.counter = 0
    setInterval(() => this.tick(), tempo)
  }
  tick(){
    this.emit("tick", this.counter++)
  }
}

const m = new Metronome(1000)
m.on("tick", t => console.log("tick:", t))
```

Observables

La clase observable tiene un método emit que podemos utilizar desde el hijo.

```
class Metronome extends Observable {
  constructor(tempo){
    super()
    this.counter = 0
    setInterval(() => this.tick(), tempo)
  }
  tick(){
    // <- la clase Observable tiene un método emit
    this.emit("tick", this.counter++)
  }
}

const m = new Metronome(1000)
m.on("tick", t => console.log("tick:", t))
```

Observables

La clase observable tiene un método on que podemos utilizar para suscribirnos a los eventos a nivel de instancia.

```
class Metronome extends Observable {
  constructor(tempo){
    super()
    this.counter = 0
    setInterval(() => this.tick(), tempo)
  }
  tick(){
    // <- la clase Observable tiene un método emit
    this.emit("tick", this.counter++)
  }
}

const m = new Metronome(1000)
// podemos suscribirnos en la instancia
m.on("tick", t => console.log("tick:", t))
```

Ejercicio observables III

Implementa la clase observable con sus tres métodos:

- on
- off
- emit