



**Clase 4.** Programación Backend

# ***Manejo de Archivos en Javascript***



## ***OBJETIVOS DE LA CLASE***

- Utilizar la forma sincrónica y asincrónica para interactuar con los archivos.
- Ver ejemplos prácticos, ventajas y desventajas de cada uno de los modos de programación.
- Como se utiliza callback en programación asincrónica.
- Conocer el módulo que usa Node.js para acceder al sistema de archivos.

# ***CRONOGRAMA DEL CURSO***

Clase 3



**Programación  
sincrónica y  
asincrónica**

---

---

---

Clase 4



**Manejo de Archivos en  
Javascript**

---

---

---

Clase 5



**Administradores de  
Paquetes - NPM**

---

---

---

***Repasando...***

- Es una **alternativa** compacta a la expresión de función tradicional.
- Nos permite definir **funciones de una sola instrucción**, esto gracias al **retorno implícito**.
- Debemos tener **cuidado** con este tipo de funciones ya que no cuentan con **scope** o **contexto**.
- Sin scope, no tendremos acceso a objetos **this** ni **arguments**.

## ***Funciones Flecha***

**Veamos un ejemplo rápido...** 

- El **sincronismo** no es más que la ejecución secuencial de nuestras instrucciones. Cada una de estas debe terminar su ejecución antes de pasar a la siguiente.
- A diferencia del **asincronismo** que se refiere la ejecución de instrucciones en un segundo plano al hilo principal.
- Para este último necesitaremos **manejadores** que recibirán el **resultado**.

# ***Sincronismo y Asincronismo***

**Veamos un ejemplo rápido...** 

- Recordemos que los **callbacks** son funciones que podemos usar cómo parámetro de otra función.
- Normalmente son usados como manejadores para instrucciones **asincrónicas**.
- Las **promesas** son objetos que encapsulan una operación, y que nos permite manejar (mediante callbacks) una resolución exitosa o fallida de la operación en cuestión normalmente **asincrónicas**.

## ***Callbacks y promesas***

**Veamos un ejemplo rápido...** 

- La función **setTimeout** nos permite establecer un **temporizador** que ejecuta un bloque de instrucciones de forma **asíncrona** después de transcurrido el tiempo establecido.
- La función **setInterval** nos permite ejecutar un bloque de instrucciones de forma reiterada con un pausa de tiempo fijo entre cada llamada. Esto también sería **asíncrono**.

## ***setTimeout & setInterval***

**Veamos un ejemplo rápido...** 





***¿Alguna pregunta hasta  
ahora?***



# ***Asincronismo y callbacks***

Realizar un programa no bloqueante utilizando timers y callbacks

*Tiempo aproximado: 20-25 minutos*



# ***Asincronismo y callbacks***

Desarrollar una función ‘mostrarLetras’ que reciba un string como parámetro y permita mostrar una vez por segundo cada uno de sus caracteres.

Al finalizar, debe invocar a la siguiente función que se le pasa también como parámetro:

```
const fin = () => console.log('terminé')
```

Realizar tres llamadas a ‘mostrarLetras’ con el mensaje ‘¡Hola!’ y demoras de 0, 250 y 500 mS verificando que los mensajes de salida se intercalen.

# ***Archivos***



# ***Introducción***



- En todo sistema, es posible que nos topemos con la necesidad de que algunos **datos persistan más allá de la ejecución del programa**.
- Una de las opciones con las que contamos es el uso de archivos.
- Según el caso, existen ventajas y desventajas en utilizar el sistema de archivos como medio de almacenamiento de información.



# ***Ventajas del uso de archivos***



- Son fáciles de usar.
- No requieren el uso de programas externos para su creación, lectura o edición.
- En ocasiones, pueden ser abiertos y editados desde programas de edición de texto simples como un bloc de notas (¡siempre que se trate de texto!).
- Son fáciles de compartir o enviar a otros usuarios/programas.



# ***Desventajas del uso de archivos***



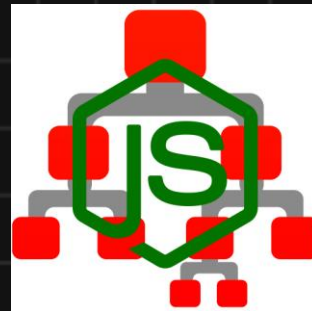
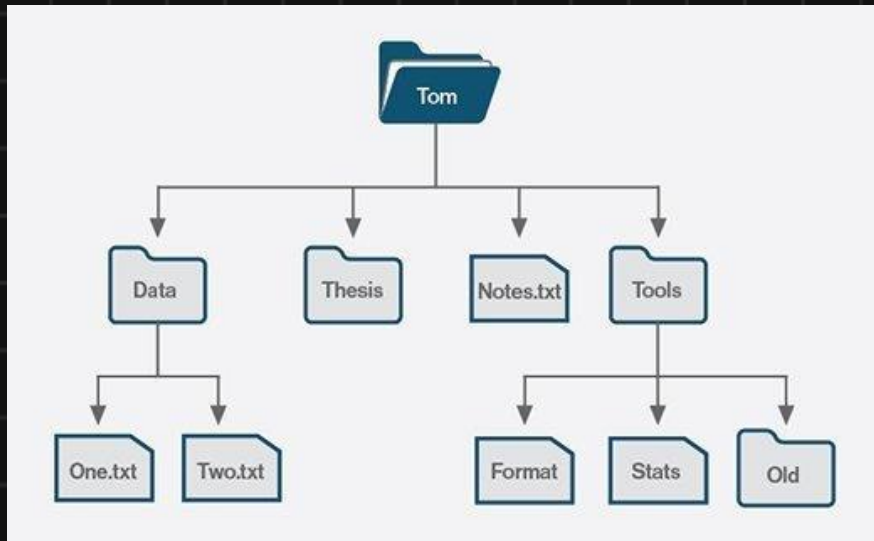
- Consultas sobre algún dato puntual entre todos los datos almacenados (y no podemos guardar todo el lote de datos en memoria).
- Ediciones de datos puntuales (que no requieren sobrescribir el archivo por completo).
- Lecturas que combinen datos obtenidos de varios archivos (nuevamente, suponiendo que no podemos guardar todos los datos en memoria).
- Probablemente sea mejor considerar el uso de un motor de base de datos.



***¿Alguna pregunta hasta  
ahora?***



# *Manejo de Archivos en NodeJS*



# ***Módulo nativo file system: fs***

- **fs** es la abreviatura en inglés para file system o sistema de archivos y es, además, uno de los módulos más básicos y útiles de Node.js.
- En Node.js es posible manipular archivos a través de fs (crear, leer, modificar, etc.).
- La mayoría de las funciones que contiene este módulo pueden usarse tanto de manera sincrónica como asincrónica.

**Aclaración:** Hay que tener en cuenta que esto sólo aplica a Node.js, desde el navegador no es posible manipular archivos dado que sería muy inseguro.

# ***Uso de fs en nuestro código***



Para poder usar este módulo solo debemos **importarlo** con la función *require* al comienzo de nuestro archivo fuente:

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

***FS: modo sincrónico***

# ***Operaciones Sincrónicas***



- Las funciones sincrónicas terminan con “Sync”
- Son **operaciones bloqueantes** que **devuelven** un **resultado**

*Podemos listar algunas de ellas:*

- ❑ **readFileSync**: lectura de un archivo en forma sincrónica
- ❑ **writeFileSync**: escritura de un archivo en forma sincrónica
- ❑ **appendFileSync**: actualización de un archivo en forma sincrónica
- ❑ **unlinkSync**: borrado de un archivo en forma sincrónica
- ❑ **mkdirSync**: creación de una carpeta

# ***Acerca de las rutas...***

- Si la ruta comienza con un `"` o `'./'` se trata de una *ruta relativa*.
  - Supongamos que el programa se está ejecutando en la carpeta **`'/user/documents/workspace/proyecto/'`**
  - Si llamamos a alguna función con la ruta: **`'./mi-archivo.txt'`** o **`'mi-archivo.txt'`**, estaremos en realidad leyendo la ruta: **`'/user/documents/workspace/proyecto/mi-archivo.txt'`**.
- Si la ruta, en cambio, comienza con  `'/'`, estaremos leyendo exactamente esa ruta.

# *Leer un archivo*



❑ `fs.readFileSync(path, encoding)`

```
const data = fs.readFileSync('./test-input-sync.txt', 'utf-8')
console.log(data)
```

- El **primer parámetro** es un **string** con la **ruta** del **archivo** que queremos **leer**
- El **segundo parámetro** indica el **formato** de **codificación** de **caracteres** con que fue escrito el dato que estamos leyendo
- El formato que utilizaremos con más frecuencia será **'utf-8'** (inglés: 8-bit Unicode Transformation Format, español: Formato de Codificación de caracteres Unicode).

***Vamos al IDE...*** 



# ***Sobreescribir un archivo***



❑ `fs.writeFileSync(ruta, datos) //sobreescribe archivo`

```
fs.writeFileSync('./test-output-sync.txt', 'ESTO ES UNA PRUEBA\n')
```

- El **primer parámetro** es un **string** con la **ruta** del **archivo** en el que queremos **escribir**
- El **segundo parámetro** indica **lo que queremos escribir**.
- La función admite un **tercer parámetro opcional** para **indicar** el **formato** de codificación de caracteres con que queremos escribir el texto: por defecto `'utf-8'`.
- Si la **ruta** provista fuera **válida**, pero el nombre de **archivo no existiera**, la función creará un **nuevo archivo** con el nombre provisto. **USE**

***Veamos un ejemplo rápido...*** 

# ***Agregar contenidos a un archivo***



❏ `fs.appendFileSync(ruta, datos)` //agregar contenido a archivo

```
fs.appendFileSync('./test-output-sync.txt', 'ESTO ES UN AGREGADO\n')
```

- El **primer parámetro** es un **string** con la **ruta** del **archivo** al que le queremos **agregar contenidos**
- El **segundo parámetro** indica **lo que queremos agregar**.
- La función admite un **tercer parámetro opcional** para **indicar** el **formato** de codificación de caracteres con que queremos escribir el texto: por defecto `'utf-8'`.
- Si la **ruta** provista fuera **válida**, pero el nombre de **archivo no existiera**, la función creará un **nuevo archivo** con el nombre provisto

***Vamos al IDE...*** 

# ***Borrar un archivo***



❏ `fs.unlinkSync(ruta)`

```
fs.unlinkSync('./test-output-sync.txt')
```

El **único parámetro** es un **string** con la **ruta** del **archivo** que queremos borrar.



***Veamos un ejemplo rápido...*** 

# *Manejo de errores*

```
try {  
  const data = fs.readFileSync('/ruta/que/no/existe')  
} catch (err) {  
  console.log(err)  
}
```

Ante una situación de error, las excepciones se lanzan inmediatamente y se pueden manejar usando **try... catch**. Esta forma de capturar errores se puede utilizar en todas las funciones sincrónicas de acceso al sistema de archivos.

***Vamos al IDE...*** 





***¿Alguna pregunta hasta  
ahora?***



## ***Fecha y hora***

Vamos a practicar lo aprendido hasta ahora



Realizar un programa que:

- A) Guarde en un archivo llamado *fyh.txt* la fecha y hora actual.
- B) Lea nuestro propio archivo de programa y lo muestre por consola.
- C) Incluya el manejo de errores con try catch (progresando las excepciones con throw new Error).

**Aclaración:** utilizar las funciones sincrónicas de lectura y escritura de archivos del módulo fs de node.js

*Tiempo: 5/10 minutos*

# ***FS: modo asincrónico vía Callbacks***

# ***Introducción: fs con Callbacks***



- Las funciones asincrónicas tiene el **mismo nombre** que sus versiones sincrónicas, pero **sin** la palabra “**Sync**” al final
- Son operaciones **no bloqueantes**
- **Reciben** un **nuevo** último **parámetro**: un **callback**.
- Los callbacks pueden recibir un primer parámetro destinado al error (si lo hubiere) para saber cómo manejarlo y un segundo parámetro, en caso de que la función en cuestión devuelva algún resultado, para indicar qué hacer con el mismo.
- Para **manejar los errores** que pueden surgir de su ejecución, **no será necesario** ejecutarlas utilizando **try / catch**.

# ***Operaciones Asíncronas***



*Podemos listar algunas de ellas:*

- ❑ **readFile**: lectura de un archivo en forma asíncronica
- ❑ **writeFile**: escritura de un archivo en forma asíncronica
- ❑ **appendFile**: actualización de un archivo en forma asíncronica
- ❑ **unlink**: borrado de un archivo en forma asíncronica
- ❑ **mkdir**: creación de una carpeta

# *Leer un archivo*



❑ *fs.readFile(ruta, encoding, callback)*

```
fs.readFile('/ruta/al/archivo', 'utf-8', (error, contenido) => {  
  if (error) {  
    // hubo un error, no pude leerlo, hacer algo!  
  } else {  
    // en este punto del código, puedo acceder a todo el contenido  
    // del archivo a través de la variable "contenido".  
    console.log(contenido)  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, **más** el **callback**. La **función** se encarga **internamente** de **abrir y cerrar** el **archivo** una vez finalizado su uso.

***Veamos un ejemplo rápido...*** 



# ***Sobreescribir un archivo***



❏ `fs.writeFile(ruta, datos, callback)` //sobreescribe archivo

```
fs.writeFile('/ruta/al/archivo', 'TEXTO DE PRUEBA\n', error => {  
  if (error) {  
    // hubo un error, no pude sobreescribirlo, hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('guardado!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**. La función se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

***Vamos al IDE...*** 

# ***Agregar contenidos a un archivo***



❏ `fs.appendFile(ruta, datos, callback)` //agregar contenido a archivo

```
fs.appendFile('/ruta/al/archivo', 'TEXTO A AGREGAR\n', error => {  
  if (error) {  
    // hubo un error, no pude agregarlo, hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('guardado!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**. La función se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

***Veamos un ejemplo rápido...*** 

# Borrar un archivo



❏ `fs.unlink(ruta, callback)`

```
fs.unlink(ruta, error => {  
  if (error) {  
    // hubo un error, no pude borrarlo, hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('borrado!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**. La función se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

***Vamos al IDE...*** 

***Otras funciones útiles***

# Crear una carpeta



❏ `fs.mkdir(ruta, callback)`

```
fs.mkdir(ruta, error => {  
  if (error) {  
    // hubo un error, no pude crear la carpeta! hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('carpeta creada!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**.

***Esta función también se encuentra en su versión sincrónica (mkdirSync).***



# *Leer el contenido de una carpeta*



❏ `fs.readdir(ruta, callback)`

```
fs.readdir(ruta, (error, nombres) => {  
  if (error) {  
    // hubo un error, no pude leer la carpeta! hacer algo!  
  } else {  
    // hacer algo con los nombres!  
    console.log(nombres)  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**.

***Esta función también se encuentra en su versión sincrónica (readdirSync).***

***Vamos al IDE...*** 



***¿Alguna pregunta hasta  
ahora?***



# ***Lectura y escritura de archivos***

Práctica del modo asincrónico

*Tiempo: 10/15 minutos*



Escribir un programa ejecutable bajo node.js que realice las siguientes acciones:

**A)** Abra una terminal en el directorio del archivo y ejecute la instrucción: *npm init -y*.

Esto creará un archivo especial (lo veremos más adelante) de nombre *package.json*

**B)** Lea el archivo *package.json* y declare un objeto con el siguiente formato y datos:

```
const info = {  
  contenidoStr: (contenido del archivo leído en formato string),  
  contenidoObj: (contenido del archivo leído en formato objeto),  
  size: (tamaño en bytes del archivo)  
}
```

**C)** Muestre por consola el objeto info luego de leer el archivo

**D)** Guardar el objeto info en un archivo llamado info.txt dentro de la misma carpeta de *package.json*

**E)** Incluir el manejo de errores (con `throw new Error`)



## Aclaraciones:

- Utilizar la lectura y escritura de archivos en modo asincrónico con callbacks.
- Consigna B): Para deserializar un string con contenido JSON utilizar `JSON.parse` (convierte string en object).
- Consigna C): Para serializar un objeto (convertirlo a string) y guardarlo en un archivo utilizar `JSON.stringify`.

## Ayuda:

Para el Punto 3 considerar usar `JSON.stringify(info, null, 2)` para preservar el formato de representación del objeto en el archivo (2 representa en este caso la cantidad de espacios de indentación usadas al representar el objeto como string).



***BREAK***

**¡10 MINUTOS Y VOLVEMOS!**

# ***FS: modo asincrónico vía Promesas***



# ***Introducción: fs con Promesas***



- El módulo **fs** nos permite operar tanto de forma **sincrónica** como **asincrónica**.
- **fs** inicialmente ofrecía funciones que reciben un callback para manejar el asincronismo.
- En una **actualización** de este módulo se agregaron versiones de **funciones asincrónicas** que en lugar de recibir callbacks, **operan mediante promesas** con then/catch.
- Posteriormente se incluyó una **sintaxis simplificada** utilizando las nuevas palabras reservadas **“async”** y **“await”**.

# Leer un archivo



*fs.promises.readFile(ruta, encoding)*



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

//Leo el archivo usando sintaxis then/catch
function leerTC() {
  fs.promises.readFile('/ruta/al/archivo', 'utf-8')
    .then( contenido => {
      console.log(contenido)
    })
    .catch( err => {
      // hubo un error, no pude leerlo, hacer algo!
      console.log('Error de lectura!',err)
    })
}
leerTC()

//Leo el archivo usando sintaxis async/await
async function leerAA() {
  try {
    const contenido = await fs.promises.readFile('/ruta/al/archivo', 'utf-8')
    console.log(contenido)
  }
  catch (err) {
    // hubo un error, no pude leerlo, hacer algo!
    console.log('Error de lectura!',err)
  }
}
leerAA()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

# ***¡Aclaraciones!***

- En el caso de querer hacer algo con la variable **fuera** del **bloque try/catch**, la **declaración** debería hacerse **fuera** del mismo.
- Recordar que debemos anteponer la palabra “**await**” al llamado a la función para que ésta se comporte de manera bloqueante.
- Si se omitiera la palabra “**await**” la instrucción `console.log(contenido)` se ejecutaría ANTES de que a la variable contenido se le asigne el resultado de la operación de lectura del archivo.
- Recordar también que la palabra “**await**” puede usarse ÚNICAMENTE dentro de una función de tipo “async” Dado que estas funciones ya no poseen un parámetro que nos permite elegir cómo manejar los errores que pueden surgir de su ejecución, vuelve a ser necesario ejecutarlas utilizando **try / catch**

# ***Sobreescribir un archivo***



❏ *fs.promises.writeFile(ruta, datos)*

```
async function escribir() {  
  try {  
    await fs.promises.writeFile('/ruta/al/archivo', 'TEXTO DE PRUEBA\n')  
    console.log('guardado!')  
  }  
  catch (err) {  
    // hubo un error, no pude escribirlo, hacer algo!  
  }  
}  
escribir()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

# Agregar contenidos a un archivo



❏ `fs.promises.appendFile(ruta, datos)`

```
async function agregar() {  
  try {  
    await fs.promises.appendFile('/ruta/al/archivo', 'TEXTO DE PRUEBA\n')  
    console.log('agregado!')  
  }  
  catch (err) {  
    // hubo un error, no pude escribirlo, hacer algo!  
  }  
}  
agregar()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

# ***Renombrar un archivo***



❏ *fs.promises.rename(rutaVieja, rutaNueva)*

```
async function renombrar(rutaVieja, rutaNueva) {  
  try {  
    await fs.promises.rename(rutaVieja, rutaNueva)  
    console.log('renombrado!')  
  }  
  catch (err) {  
    // hubo un error, no pude escribirlo, hacer algo!  
  }  
}  
renombrar()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

***Vamos al IDE...*** 



***¿Alguna pregunta hasta  
ahora?***





# ***Lectura y escritura con promises***

*Tiempo: 5/10 minutos*



Realizar un programa que ejecute las siguientes tareas:

- A)** Lea el archivo `info.txt` generado en el desafío anterior deserializándolo en un objeto llamado `info`.
- B)** Mostrar este objeto `info` en la consola.
- C)** Modifique el `author` a "Coderhouse" y guarde el objeto serializado en otro archivo llamado `package.json.coder`
- D)** Mostrar los errores por consola.



## Aclaraciones:

Trabajar con fs.promises (then/catch).

## Ayuda:

Para el punto 3 considerar usar `JSON.stringify(info.contenidoObj, null, 2)` para preservar el formato de representación del objeto en el archivo.



# ***MANEJO DE ARCHIVOS***

# Manejo de archivos

**Formato:** carpeta comprimida con el proyecto.

Desafío  
entregable



**>> Consigna:** Implementar programa que contenga una clase llamada Contenedor que reciba el nombre del archivo con el que va a trabajar e implemente los siguientes métodos:

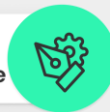
- `save(Object): Number` - Recibe un objeto, lo guarda en el archivo, devuelve el id asignado.
- `getById(Number): Object` - Recibe un id y devuelve el objeto con ese id, o null si no está.
- `getAll(): Object[]` - Devuelve un array con los objetos presentes en el archivo.
- `deleteById(Number): void` - Elimina del archivo el objeto con el id buscado.
- `deleteAll(): void` - Elimina todos los objetos presentes en el archivo.

# Manejo de archivos

**Formato:** carpeta comprimida con el proyecto.

**Sugerencia:** usar un archivo para la clase y otro de test, que la importe

Desafío  
entregable



## >> Aspectos a incluir en el entregable:

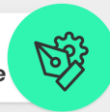
- El método `save` incorporará al producto un id numérico, que deberá ser siempre uno más que el id del último objeto agregado (o id 1 si es el primer objeto que se agrega) y no puede estar repetido.
- Tomar en consideración el contenido previo del archivo, en caso de utilizar uno existente.
- Implementar el manejo de archivos con el módulo `fs` de `node.js`, utilizando promesas con `async/await` y manejo de errores.
- Probar el módulo creando un contenedor de *productos*, que se guarde en el archivo: "productos.txt"
- Incluir un llamado de prueba a cada método, y mostrando por pantalla según corresponda para verificar el correcto funcionamiento del módulo construido.
- El formato de cada producto será :

```
{  
  title: (nombre del producto),  
  price: (precio),  
  thumbnail: (url de la foto del producto)  
}
```

# Manejo de archivos

**Formato:** carpeta comprimida con el proyecto.

Desafío  
entregable



## >> Ejemplo:

Contenido de "productos.txt" con 3 productos almacenados

```
[
  {
    title: 'Escuadra',
    price: 123.45,
    thumbnail: 'https://cdn3.iconfinder.com/data/icons/education-209/64/ruler-triangle-stationary-school-256.png',
    id: 1
  },
  {
    title: 'Calculadora',
    price: 234.56,
    thumbnail: 'https://cdn3.iconfinder.com/data/icons/education-209/64/calculator-math-tool-school-256.png',
    id: 2
  },
  {
    title: 'Globo Terráqueo',
    price: 345.67,
    thumbnail: 'https://cdn3.iconfinder.com/data/icons/education-209/64/globe-earth-geograhpy-planet-school-256.png',
    id: 3
  }
]
```

**CODER HOUSE**

***¿PREGUNTAS?***

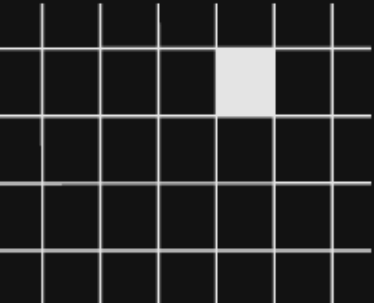






# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Funciones
  - Callbacks
  - Promesas
  - Ejecución sincrónica/asincrónica
  - Manejo de archivos en Node.js
- 



***OPINA Y VALORA ESTA CLASE***

***#DEMOCRATIZANDOLAEDUCACIÓN***