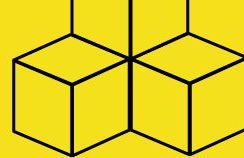
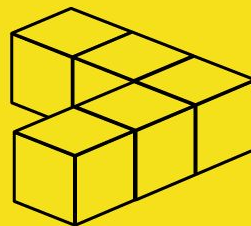
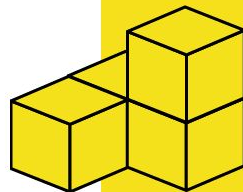


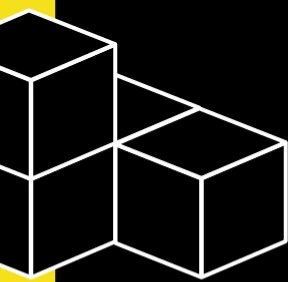
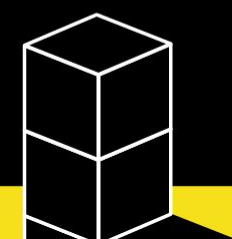
Escuela de **JavaScript**





Agenda de Actividades:



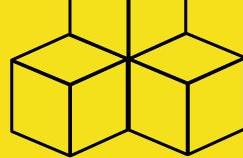
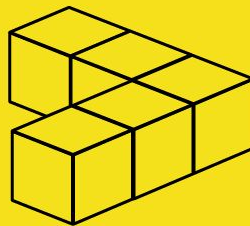
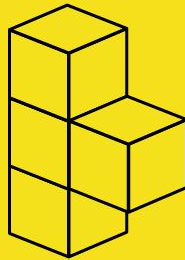
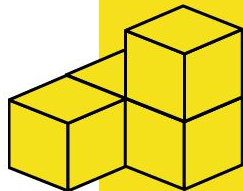
- 09:00 - 09:15 Recapitulación Fundamentos de JavaScript
 - 09:15 - 09:30 Q&A
 - 09:30 - 10:00 Clase de Practica
 - 10:00 - 10:30 Q&A
 - 10:30 - 11:00 Resolver Retos
 - 11:00 - 12:15 Mentoría sobre proyectos
- 
- 

Código de Conducta





Recapitulación Profesional de JavaScript



Scope:

El **Scope** o ámbito es lo que define el tiempo de vida de una variable, en que partes de nuestro código pueden ser usadas.

Global Scope

Variables disponibles de forma global se usa la palabra *var*, son accesibles por todos los scripts que se cargan en la página. Aquí hay mucho riesgo de sobreescritura.

Function Scope

Variables declaradas dentro de una función sólo visibles dentro de ella misma (incluyendo los argumentos que se pasan a la función).

Block Scope

Variables definidas dentro de un bloque, por ejemplo variables declaradas dentro un loop *while* o *for*. Se usa *let* y *const* para declarar este tipo de variables.

Module Scope

Cuando se denota un script de tipo module con el atributo `type="module"` las variables son limitadas al archivo en el que están declaradas.

Scope:

El **Scope** o ámbito es lo que define el tiempo de vida de una variable, en que partes de nuestro código pueden ser usadas.

Global Scope

Variables disponibles de forma global se usa la palabra *var*, son accesibles por todos los scripts que se cargan en la página. Aquí hay mucho riesgo de sobreescritura.

Function Scope

Variables declaradas dentro de una función sólo visibles dentro de ella misma (incluyendo los argumentos que se pasan a la función).

Block Scope

Variables definidas dentro de un bloque, por ejemplo variables declaradas dentro un loop *while* o *for*. Se usa *let* y *const* para declarar este tipo de variables.

Module Scope

Cuando se denota un script de tipo module con el atributo `type="module"` las variables son limitadas al archivo en el que están declaradas.

Closures

Son funciones que regresan una función o un objeto con funciones que mantienen las variables que fueron declaradas fuera de su scope.

Los **closures** nos sirven para tener algo parecido a variables privadas, característica que no tiene JavaScript por *default*. Es decir encapsulan variables que no pueden ser modificadas directamente por otros objetos, sólo por funciones pertenecientes al mismo.

This

this se refiere a un objeto, ese objeto es el que actualmente está ejecutando un pedazo de código.

No se puede asignar un valor a *this* directamente y este depende de en que scope nos encontramos:

Cuando llamamos a *this* en el **Global Scope** o **Function Scope**, se hace referencia al objeto *window*. A excepción de cuando estamos en **strict mode** que nos regresará *undefined*.

Cuando llamamos a *this* desde **una función** que está contenida en un objeto, *this* se hace referencia a ese objeto.

Cuando llamamos a *this* desde una “**clase**”, se hace referencia a la instancia generada por el constructor.

Métodos call, apply y bind

Estas funciones nos sirven para establecer el valor de *this*, es decir cambiar el contexto que se va usar cuando la función sea llamada.

Las funciones **call**, **apply** y **bind** son parte del prototipo Function. Toda función usa este prototipo y por lo tanto tiene estas tres funciones.

functionName.call(). Ejecuta la función recibiendo como primer argumento el *this* y los siguientes son los argumentos que recibe la función que llamó a call.

functionName.apply(). Ejecuta la función recibiendo como primer argumento el *this* y como segundo un arreglo con los argumentos que recibe la función que llamó a apply.

functionName.bind(). Recibe como primer y único argumento el *this*. No ejecuta la función, sólo regresa otra función con el nuevo this integrado.

Prototype

En Javascript todo son objetos, no tenemos clases, no tenemos ese plano para crear objetos.

Todos los objetos “heredan” de un prototipo que a su vez hereda de otro prototipo y así sucesivamente creando lo que se llama la **prototype chain**.

La keyword *new* crea un nuevo objeto que “hereda” todas las propiedades del prototype de otro objeto. No confundir prototype con **proto** que es sólo una propiedad en cada instancia que apunta al prototipo del que hereda.

Herencia Prototipal

Por default los objetos en JavaScript tienen cómo prototipo a **Object** que es el punto de partida de todos los objetos, es el prototipo padre. Object es la raíz de todo, por lo tanto tiene un prototipo padre *undefined*.

Cuando se llama a una función o variable que no se encuentra en el mismo objeto que la llamó, se busca en toda la prototype chain hasta encontrarla o regresar *undefined*.

La función **hasOwnProperty** sirve para verificar si una propiedad es parte del objeto o si viene heredada desde su prototype chain.

Parsers y el Abstract Syntax Tree

El JS Engine recibe el código fuente y lo procesa de la siguiente manera:

El **parser** descompone y crea tokens que integran el **AST**.

Se compila a **bytecode** y se ejecuta.

Lo que se pueda se **optimiza a machine code** y se reemplaza el código base.

Un **SyntaxError** es lanzado cuando el motor JavaScript encuentra partes que no forman parte de la sintaxis del lenguaje y esto lo logra gracias a que se tiene un AST generado por el parser.

El *parser* es del 15% al 20% del proceso de ejecución por lo que hay que usar parser del código justo en el momento que lo necesitamos y no antes de saber si se va a usar o no.

Cómo funciona el JavaScript Engine

Una vez tenemos el **AST** ahora hay que convertirlo a Bytecode.

Bytecode es como el código assembler pero en lugar de operar en el procesador opera en la máquina virtual **V8** del navegador.

Machine code es el más bajo nivel, es código binario que va directo al procesador.

El **profiler** se sitúa en medio del bytecode y el optimizador

Cada máquina virtual tiene sus particularidades, por ejemplo V8 tiene algo llamado **Hot Functions**.

Cuando una sentencia función es ejecutada muy frecuentemente, V8 la denomina como una *hot function* y hace una optimización que consiste en convertirla a *machine code* para no tener que interpretarla de nuevo y agilizar su ejecución.

Cada navegador tiene su implementación de JavaScript Engine:

SpiderMonkey = Firefox - Chackra = Edge - JavaScriptCore = Safari - V8 = Chrome

Event Loop

El **Event Loop** hace que Javascript parezca ser multihilo a pesar de que corre en un solo proceso.

Javascript se organiza usando las siguientes estructuras de datos:

Stack. Va apilando de forma organizada las diferentes instrucciones que se llaman. Lleva así un rastro de dónde está el programa, en que punto de ejecución nos encontramos.

Memory Heap. De forma desorganizada se guarda información de las variables y del scope.

Schedule Tasks. Aquí se agregan a la cola, las tareas programadas para su ejecución.

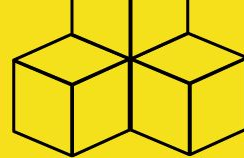
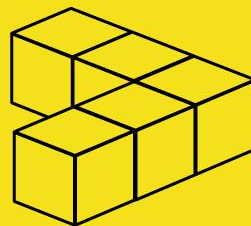
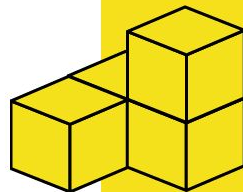
Task Queue. Aquí se agregan las tareas que ya están listas para pasar al stack y ser ejecutadas. El stack debe estar vacío para que esto suceda.

MicroTask Queue. Aquí se agregan las promesas. Esta Queue es la que tiene mayor prioridad.

El Event Loop es un loop que está ejecutando todo el tiempo y pasa periódicamente revisando las queues y el stack moviendo tareas entre estas dos estructuras.



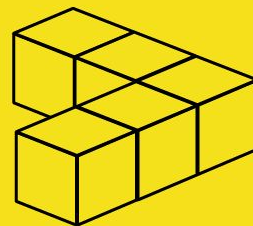
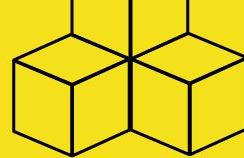
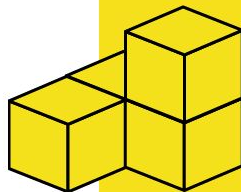
Clase Practica



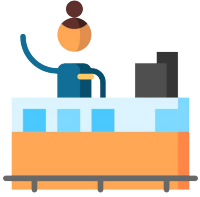


Introducción

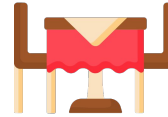
Reto 04



Fonditaloop



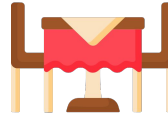
Mesa 2



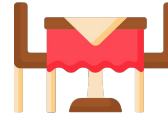
Mesa 4



Mesa 1



Mesa 3



Mesa 5



Fonditaloop



Mesa 2



Mesa 4



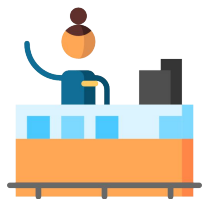
Mesa 1



Mesa 3



Mesa 5



Fonditaloop



Waiter 3



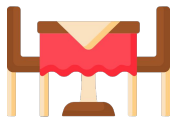
Mesa 2



Mesa 4



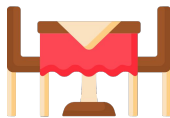
Waiter 1



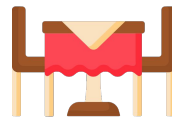
Mesa 1



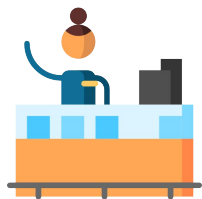
Waiter 2



Mesa 3



Mesa 5



Fonditaloop



Waiter 3



Mesa 2



Mesa 4



Waiter 1



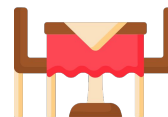
Mesa 1



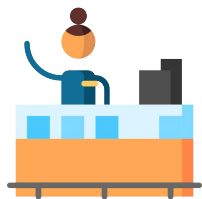
Waiter 2



Mesa 3



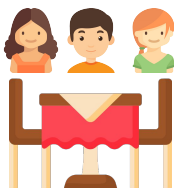
Mesa 5



Fonditaloop



Waiter 3



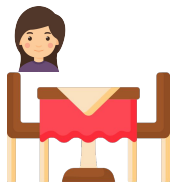
Mesa 2



Mesa 4



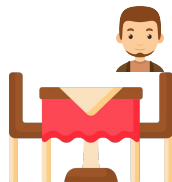
Waiter 1



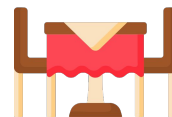
Mesa 1



Waiter 2



Mesa 3



Mesa 5



Fonditaloop



Waiter 3



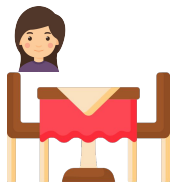
Mesa 2



Mesa 4



Waiter 1



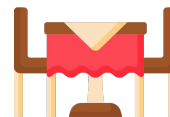
Mesa 1



Waiter 2



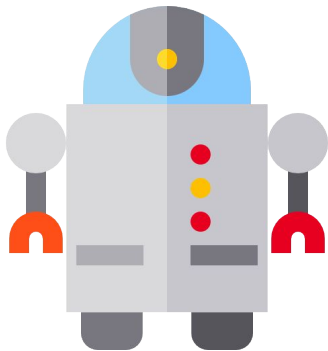
Mesa 3



Mesa 5

¡Vamos al código!

Fonditaloop



Reto 1: Implementa una función `randomTime` que regrese un valor entre 1000 y 8000 ms (Investiga)

Reto 2: Crea una función llamada `'waiter2'` que se encargue de recoger los pedidos de las mesas 1 y 3 (usando promesas y `randomTime`)

Reto 3: Crea una función llamada `'waiter3'` que se encargue de recoger el pedido de la mesa 2. Los platillos solo pueden ser servidos cuando la orden esté completa. (usando `async/await`, `randomTime` y manejo de errores)

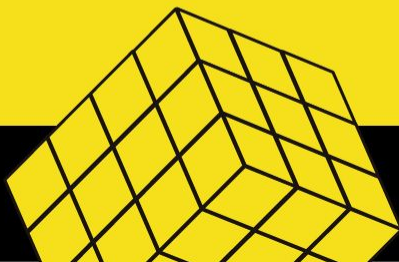
Reto 4: (Opcional) Crea una función llamada `"fetchOrders"` que realice un llamado a la API de ordenes y una función llamada `"waiter4"` que se encargue de solicitar 4 pedidos que deban de ser entregados hasta que estén todos listos.

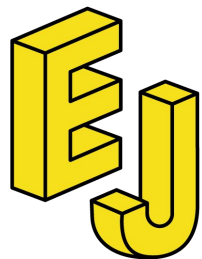
¿Dudas?



Reto:

Fork: <https://github.com/platzi/escuelajs-reto-04>





Escuela de **JavaScript**

