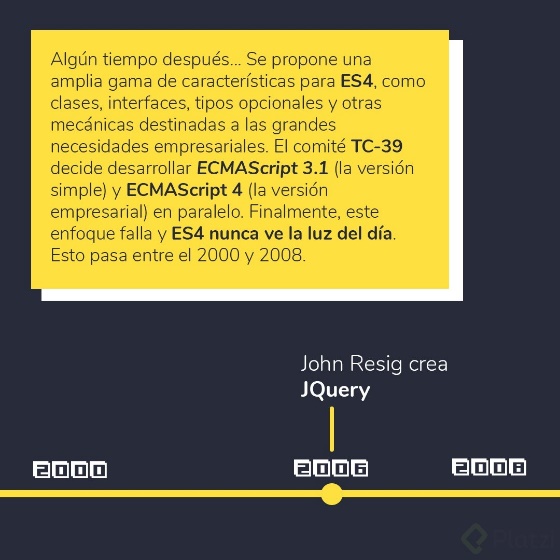
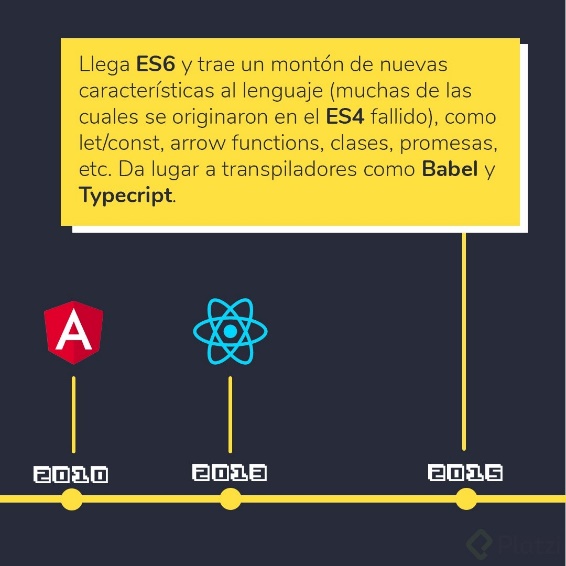
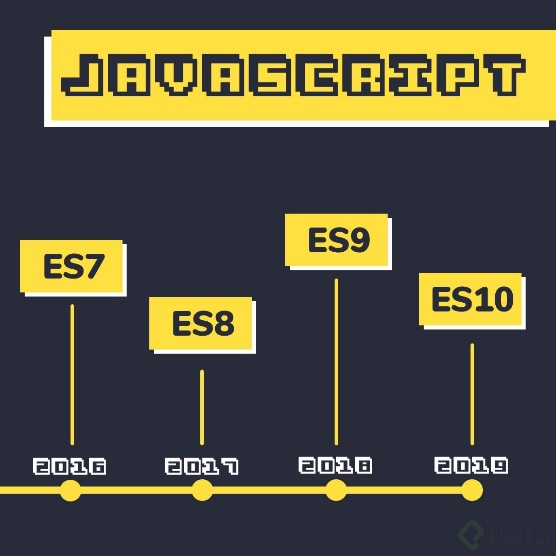
Curso de JavaScript Engine (V8) y el Navegador

# Historia de JavaScript

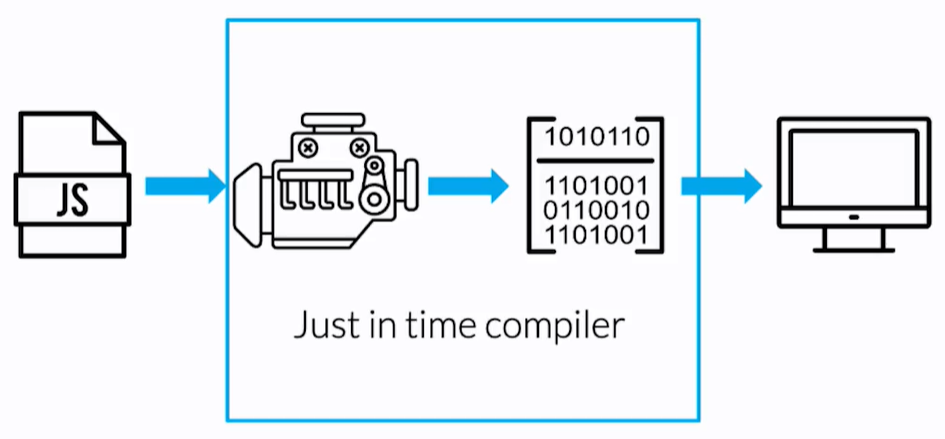
 

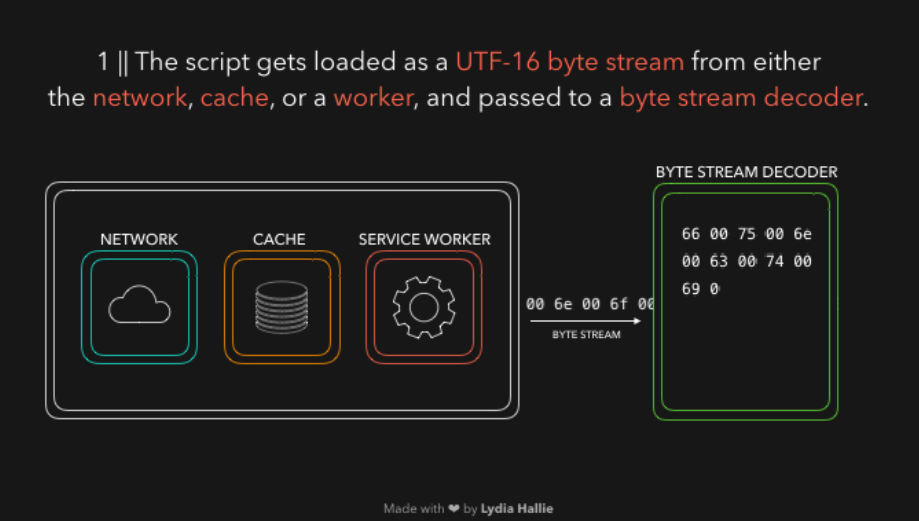
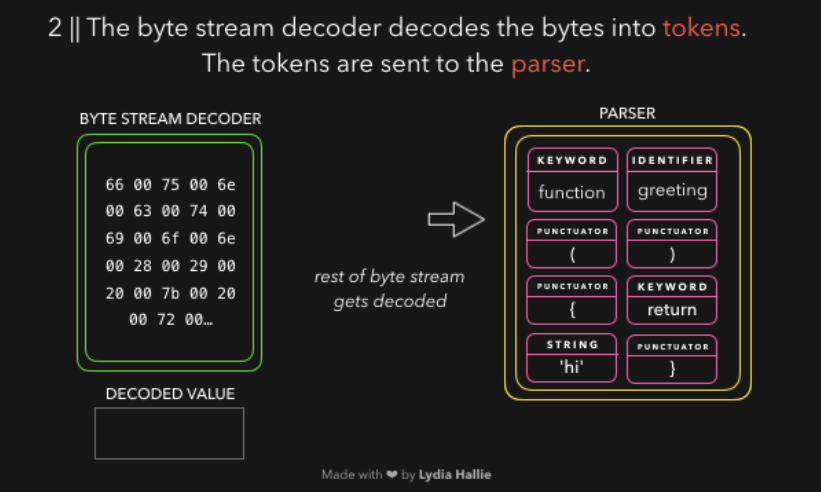
 

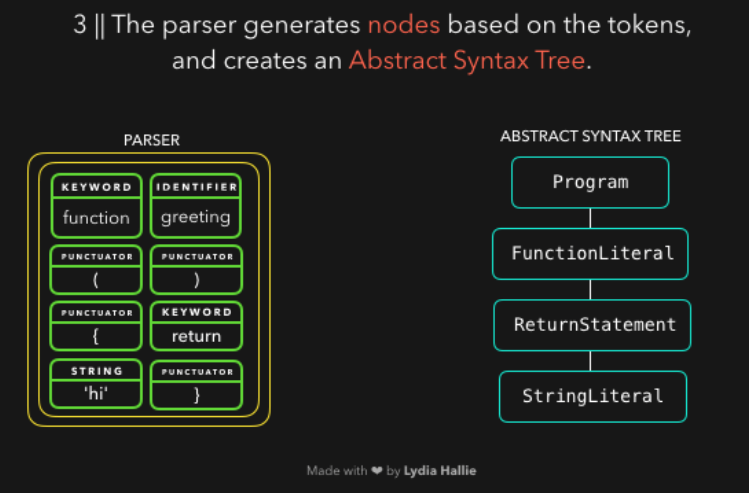
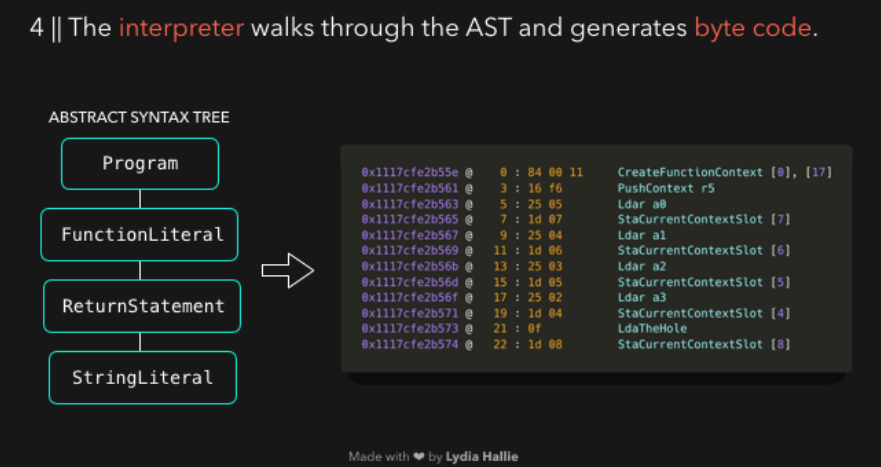
# ¿Cómo funciona el JavaScript Engine?

JavaScript Engine: Es el motor de JS, siempre corre en el navegador.  
Su función es interpretar el código JavaScript y convertirlo a Machine Code para que la máquina pueda entenderlo.

* Just in time compiler: Es la compilación en tiempo real que sucede en el proceso del Engine.



# V8, el JavaScript Engine de Chrome

V8 es un motor open-source escrito en C++ para compilar JavaScript y WebAssembly en código máquina. Esto quiere decir que traduce JavasScript a un código puramente digital capaz de ser interpretado por la CPU donde se ejecuta.

Este motor fue desarrollado por Google para Google Chrome y su primera versión vio la luz en 2008 junto con la primera versión del navegador.

Cada navegador tiene un motor diferente, pero como V8 es open source otros navegadores están migrando a él.

* V8 ayuda a que JS corra de manera más rápida, con esto podemos crear aplicaciones más robustas y rápidas.  
  -V8 nació para que Google Maps funcione de manera óptima.

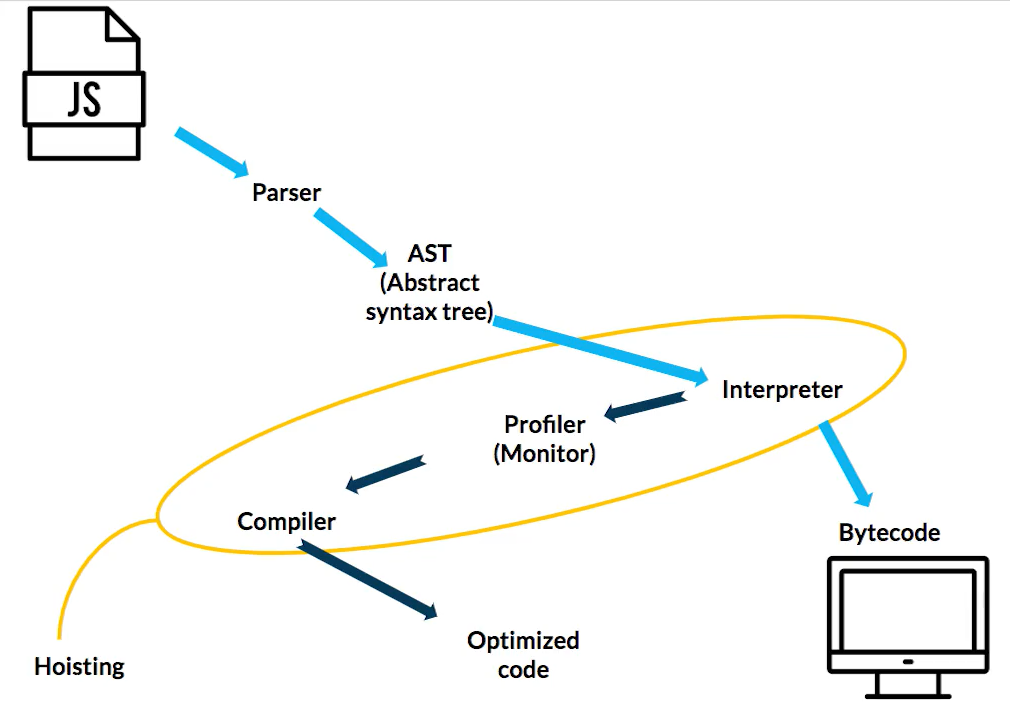
*Tres preguntas de exmen*

El motor de JavaScript V8 es lanzado en el 2008, a traves del navegador de Google Crhome, pero especialemte nacio para que la aplcacion Google Maps corriera mas rapido.

V8 ayuda a que JS corra de manera más rápida, con esto podemos crear aplicaciones más robustas y rápidas.

# Profundizando en el Engine

* Una vez el motor recibe un archivo JS genera un Global enviroment
* En el Global enviroment se genera un Global object que es igual a window
* También se genera una variable this
* this depende del contexto, en el entorno global this es igual al objeto global
* Ya después de generado el entorno global, comienza a correr nuestro código en el contexto de ejecución, a través de un stack de tareas apiladas
* Una vez que el motor comienza a interactuar con el navegador, genera un parseo del documento completo, para encontrar las keywords y las pasa al AST (abstract syntax tree) [AST Explorer](https://astexplorer.net/)
* Una vez se tiene el AST, el motor lo pasa al interprete para retornar bytecode puede ser entendido por la máquina
* Si en la interpretación del AST ve mucho código que se repite y que se puede optimizar entra el profiler o monitor, y optimiza el código y lo regresa compilado como bytecode para que la máquina lo pueda comprender. Es justo en este paso donde aparece el hoisting, donde el motor en su función de optimización reinterpreta el código de una mejor forma y se pueden presentar errores de programación.
* El hoisting solo sucede con variables y funciones que se mandan a llamar.



# Ejemplo de Objeto global y hoisting

Hoisting es un término que no encontrará utilizado en ninguna especificación previa a ECMAScript® 2015 Language Specification. El concepto de Hoisting fue pensado como una manera general de referirse a cómo funcionan los contextos de ejecución en JavaScript (específicamente las fases de creación y ejecución). Sin embargo, el concepto puede ser un poco confuso al principio.  
  
Conceptualmente, por ejemplo, una estricta definición de hoisting sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código, pero esto no es lo que ocurre en realidad. Lo que sucede es que las declaraciones de variables y funciones son asignadas en memoria durante la fase de compilación, pero quedan exactamente en dónde las has escrito en el código.

**Hoisting**

* Si llamamos una variable antes de ser declarada, el compiler crea la variable en la memory heap y la inicializa como undefined
* En el caso de las funciones es distinto, primero mandamos a llamar a las funciones antes de ejecutarlas.
* El hoisting a veces funciona pero no tenemos control de las variables que se van a cambiar
* Si llamamos a una constante (const) antes de inicializar retorna un error de tipo: Uncaught ReferenceError, que corresponde a variables que son referenciadas pero no pudieron ser capturadas

*Una preguntica de examen*

Una estricta definición de hoisting sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código,asignandolas en memoria dentro de un contexto de ejecucion.

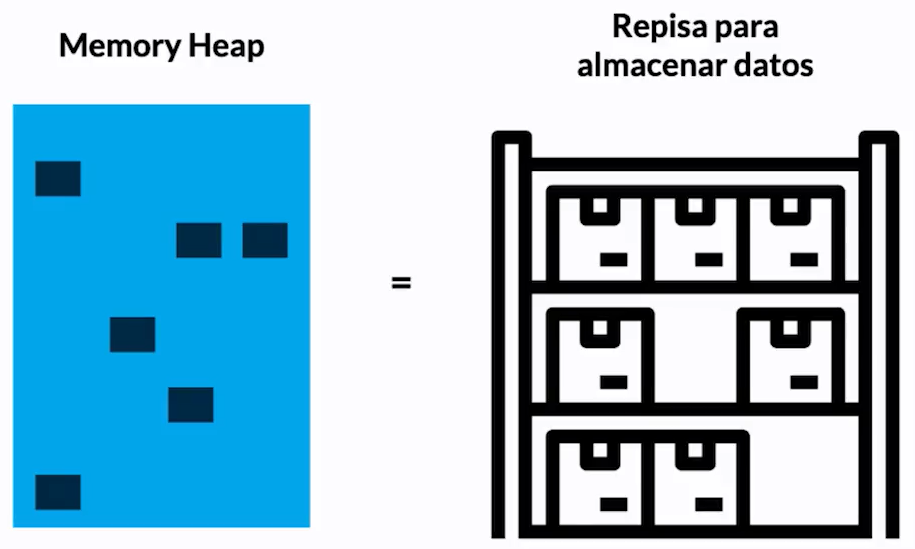
**Código de Ejecución**

# Memory Heap

Los objetos en JS (objetos, arrays, funciones y básicamente todo lo que no sea un valor primitivo) se almacenan en la parte de memoria que de llama Memory Heap. Los valores primitivos son almacenados en el Call Stack, dentro del Scope (Contexto de Ejecución de la función que tenga acceso a esa variable). Acceder al Call Stack es mucho más rápido que al Heap. Además, en el Call Stack también se guardan las referencias, “como si fueran valores primitivos”. Cuando se asigna una variable a otra y esta apunta a un objeto, se copia la referencia, como si fuera un valor primitivo. Si el objeto tiene atributos como un número por ejemplo, este se guarda en la posición de memoria reservada para ese objeto. Los objetos también pueden tener más objetos dentro. En ese caso, dentro de la posición de memoria de ese objeto se va a guardar una referencia a otra posición de memoria.

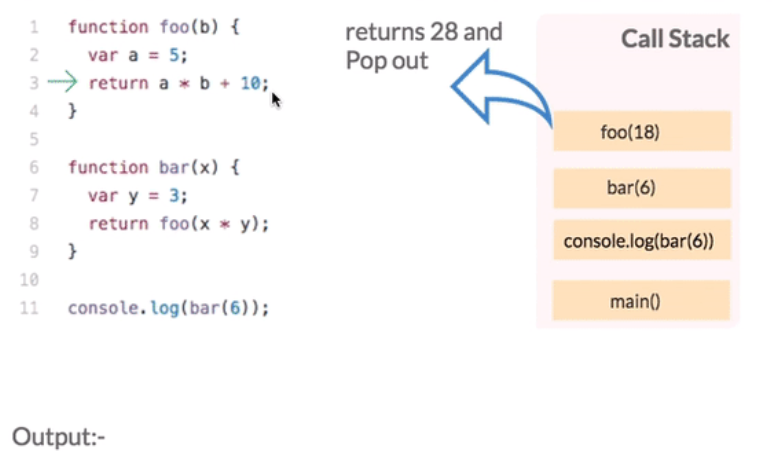
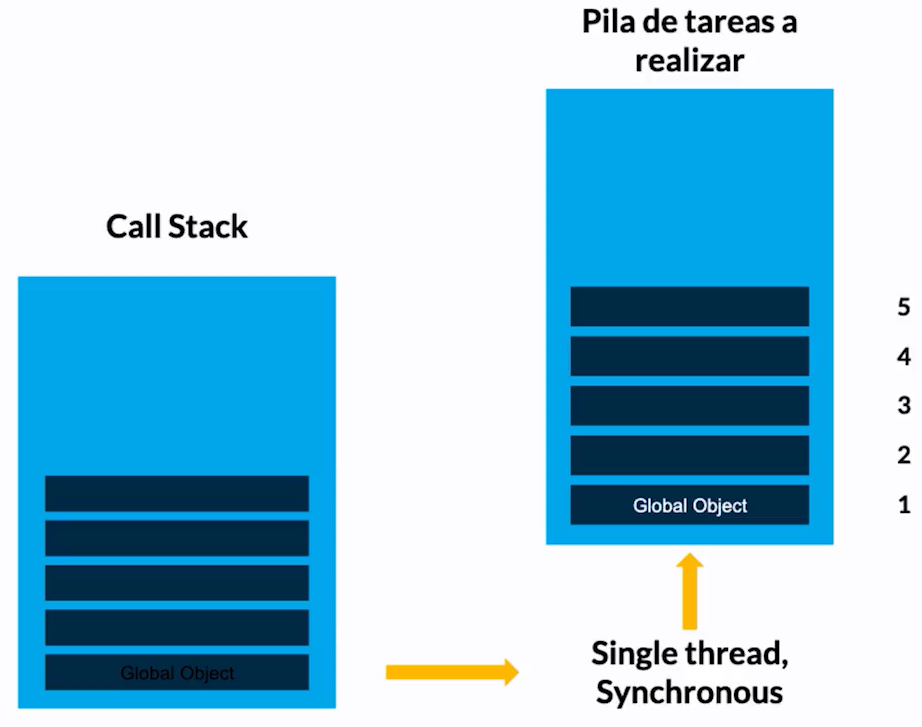
Memory Heap:

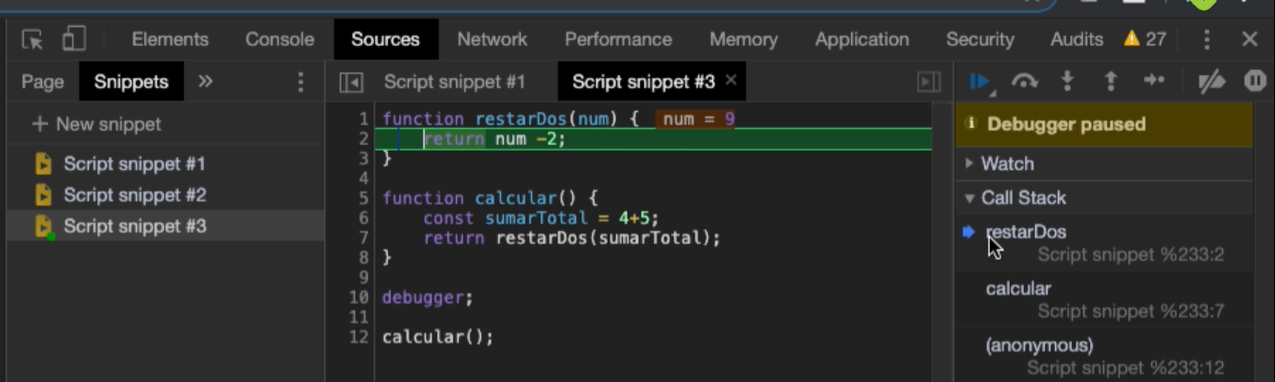
* Donde se almacena los valores de las variables y las funciones
* Se destina un espacio en memoria para las variables.
* La información en el memory heap, No se guarda de manera lineal



# Call Stack

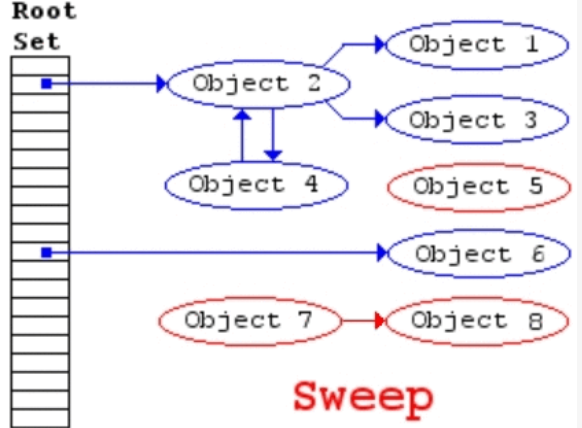
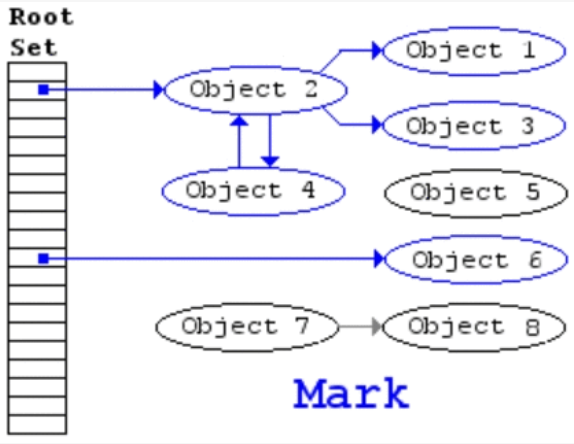
* Como se mandan a llamar las variables y las funciones
* Las tareas en el callstack se apilan de abajo hacia arriba.
* Se llaman de la última que mandamos a llamar hacia abajo
* En la base de la pila reposa el Global Object
* Si una función llama a otra, la pone encima de la pila.
* Se ejecuta una tarea a la vez (sincronía)
* Una vez que se van ejecutando las tareas se van retirando de la pila
* Al ejecutar todas las tareas se retira el Global object.



# Garbage Collection

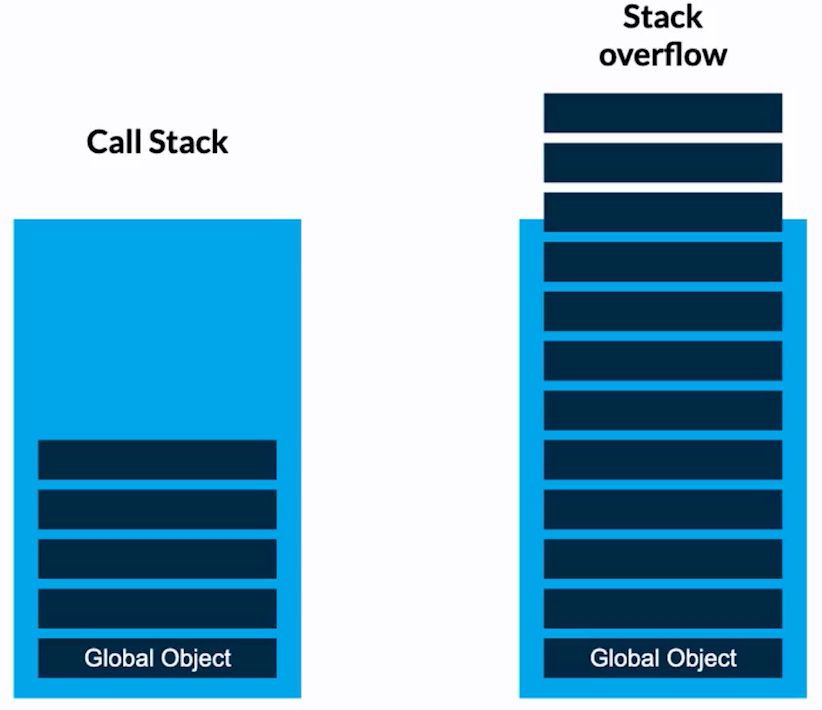
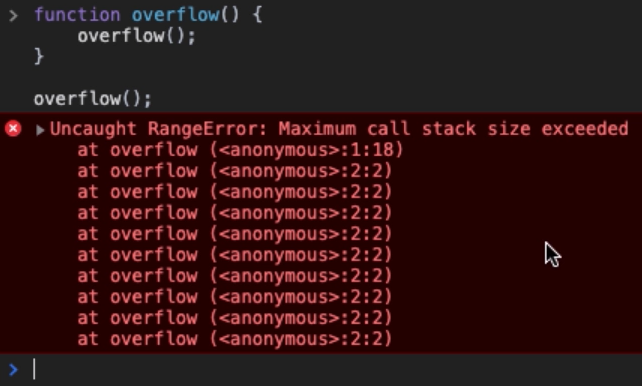
Limpía la memoria de los datos no utilizados para no sobrecargarla y seguir trabajando sin problemas.



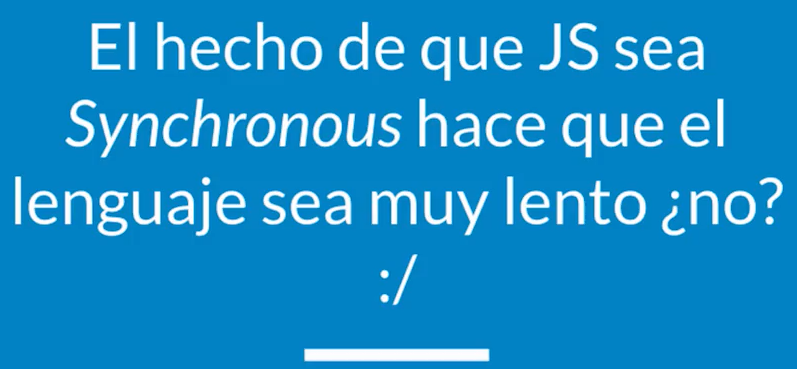
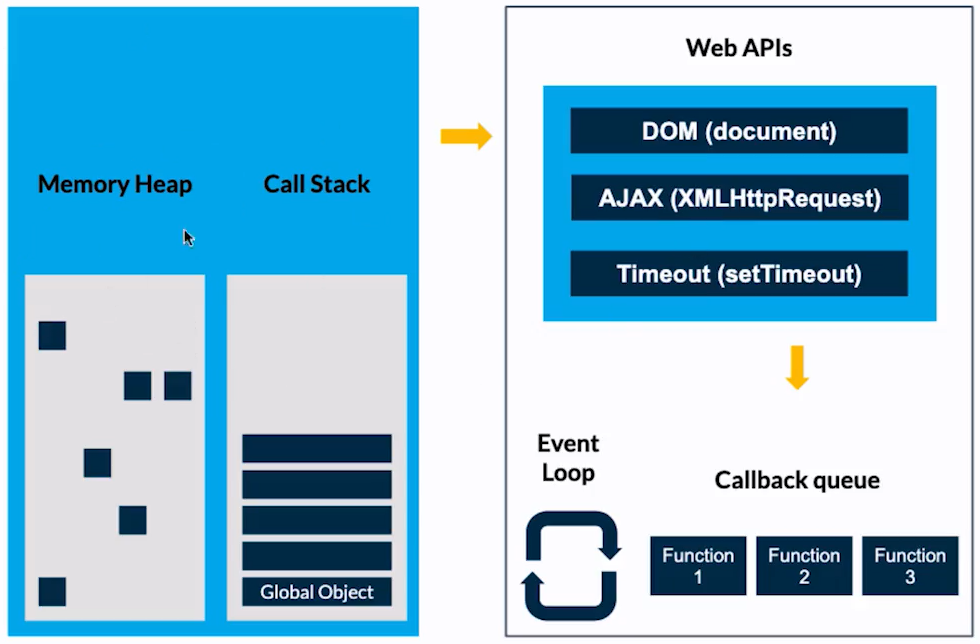
El algoritmo **Mark & Sweep** hace referencia a cuando un tipo de dato **se vuelve inalcanzable** para el programa.  
Se toma una raíz la cual es el Objeto Global y a medida que el programa avanza, los objetos van creando o borrando referencias a sus raíces.  
Cuando un objeto se queda con 0 referencias, se dice que el objeto es inalcanzable y es aquí cuando el Garbage Collector entra a hacer su trabajo (Liberar el espacio que usaban esas variables u objetos)  
Desde 2012 los navegadores usan un Garbage Collector basado en este algoritmo, que ha ido recibiendo mejoras en su implementación constantemente.

# Stack overflow

El **stack overflow** se genera cuando el **call stack** se llena completamente (**pila de tareas**) .Esto pasa cuando se genera o se trabaja con **bucles infinitos, recursividad y funciones**. Entonces este entra en stack overflow , tenemos que tener cuidado de no ocasionar estos stack!!!

# JavaScript Runtime



# Asincronía

Por default corre una tarea a la vez – Sincronismo  
Ahora veremos que es el asincronismo  
Memory Heap: Espacio donde se guardan funciones y variables  
Call Stack: Donde se apilan todas las tareas que tenemos que hacer con Javascript  
Web API´s (Ofrecidas por el navegador para manipular lo siguiente)

* Dom(document)
* AJAX(XMLHttpRequest)
* Timeout(setTimeout)  
  Call Back Queue: El orden en que se van a ejecutar a funciones  
  Al momento de usar asincronismo sacamos funciones del Call Back Queue que no serán ejecutadas por javascript y serán ejecutadas por el navegador despues  
  Ejemplo



# Recapitulación

