# **Quiz de JavaScript**

**Pregunta 1: Async/Await y Promises**

const promisify = (item, delay) => new Promise((resolve) =>

setTimeout(() => resolve(item), delay)

);

const a = () => promisify('a', 100); // Tarda 100ms

const b = () => promisify('b', 5000); // Tarda 5000ms (5 segundos)

const c = () => promisify('c', 3000); // Tarda 3000ms (3 segundos)

async function one() {

const promises = [a(), b(), c()]; // ¡Se ejecutan TODAS al mismo tiempo!

const [output1, output2, output3] = await Promise.all(promises);

return `one is done: ${output1} ${output2} ${output3}`;

}

async function two() {

const promises = [a(), b(), c()]; // ¡Se ejecutan TODAS al mismo tiempo!

const output1 = await Promise.race(promises);

return `two is done: ${output1}`;

}

async function three() {

const output1 = await a(); // Espera 100ms

const output2 = await b(); // Luego espera 5000ms más

const output3 = await c(); // Luego espera 3000ms más

return `three is done ${output1} ${output2} ${output3}`;

}  
  
**Respuesta Correcta:**  
two is done: a one is done: a b c three is done a b c

**Conceptos Fundamentales a Dominar:**

1. **Promesas (Promises):** Representan un valor que puede estar disponible ahora, en el futuro o nunca. Son la base de la asincronía moderna en JavaScript.
2. **async/await:** Una sintaxis más limpia para trabajar con promesas, que permite escribir código asíncrono que parece síncrono.
3. **Ejecución Paralela vs. Secuencial:** Entender cuándo las tareas se inician al mismo tiempo y cuándo una espera a que la anterior termine.

**Análisis Detallado de la Lógica del Código:**  
El orden de las salidas en la consola depende del tiempo total que tarda cada función en completarse.

* **function two() (La más rápida - 100ms):**
  + Inicia las promesas a, b y c en paralelo.
  + Promise.race espera solo a que la **primera** promesa se resuelva. La más rápida es a() con 100ms.
  + Por lo tanto, esta función termina primero.
* **function one() (Intermedia - 5000ms):**
  + Inicia las promesas a, b y c en paralelo.
  + Promise.all espera a que **todas** las promesas se resuelvan. El tiempo total está determinado por la promesa más lenta, que es b() con 5000ms.
* **function three() (La más lenta - 8100ms):**
  + Usa await de forma secuencial. No inicia la siguiente promesa hasta que la anterior haya terminado.
  + El tiempo total es la **suma** de todas las esperas: 100ms (a) + 5000ms (b) + 3000ms (c) = 8100ms.

**Explicación de Métodos y Sintaxis Clave:**

* **Promise.all(iterable):** Ideal para cuando tienes múltiples tareas asíncronas que pueden ejecutarse en paralelo y necesitas esperar a que todas terminen para continuar. Devuelve un array con los resultados en el mismo orden en que se pasaron las promesas.
* **Promise.race(iterable):** Útil cuando tienes múltiples fuentes para obtener un mismo dato y solo te importa la que responda primero. Devuelve el valor de la primera promesa que se resuelva o rechace.

**Pregunta 2: Acceso al Objeto Global**

**La Pregunta:**

// ¿Cuál es el objeto global en el navegador?  
  
**Respuesta Correcta:** window

**Conceptos Fundamentales a Dominar:**

1. **Ámbito Global (Global Scope):** El nivel más alto del ámbito en un programa JavaScript. Las variables declaradas aquí son accesibles desde cualquier parte.
2. **Entornos de Ejecución (Runtimes):** JavaScript se ejecuta en diferentes entornos (navegador, servidor), y cada uno puede tener un nombre diferente para el objeto global.

**Análisis Detallado de la Lógica del Código:**  
La pregunta especifica "en un navegador típico". En este entorno, el objeto global ha sido históricamente window.

* window no solo contiene las variables y funciones que declaras globalmente, sino también una gran cantidad de APIs proporcionadas por el navegador, como document, location, navigator, setTimeout, fetch, etc.

**Explicación de Métodos y Sintaxis Clave:**

* **window:** El objeto global en el entorno del navegador.
* **global:** El objeto global en el entorno de Node.js.
* **self:** A menudo se usa en Web Workers, donde window no está disponible.
* **globalThis:** El estándar moderno (ES2020) para acceder al objeto global de manera universal, sin importar el entorno. Es la forma recomendada hoy en día para escribir código portable.

**Descarte de Opciones Incorrectas:**

* **Object:** Es el constructor base para todos los objetos de JavaScript, no el objeto global.
* **there is no global object:** Es falso. Todo entorno de JavaScript tiene un objeto global.
* **$:** Es un identificador de variable válido, popularizado por librerías como jQuery, pero no es una parte nativa del lenguaje para acceder al objeto global.

**Pregunta 3: JavaScript Engine vs. Runtime**

**La Pregunta:**

// ¿Cuál es la diferencia entre JavaScript Engine y Runtime?

// ¿V8 es un Engine o Runtime?

// ¿Node.js es un Engine o Runtime?  
  
**Respuestas Correctas:**

* Engine compiles or interprets the code, Runtime gives access to environment APIs
* Node.js is a Runtime, V8 is an Engine

**Conceptos Fundamentales a Dominar:**

1. **Arquitectura de JavaScript:** Comprender que el lenguaje que escribimos es ejecutado por un sistema compuesto por varias partes.

**Análisis Detallado de la Lógica del Código:**  
La pregunta busca la relación correcta entre dos componentes clave.

* El **Engine** es el "motor de un coche". Su única función es tomar el combustible (código JS) y hacerlo funcionar. No sabe de carreteras, señales de tráfico o GPS (APIs).
* El **Runtime** es el "coche completo". Incluye el motor (Engine), pero también el chasis, las ruedas, el volante, el GPS (APIs del entorno), etc., que permiten que el motor interactúe con el mundo real.

**Explicación de Métodos y Sintaxis Clave:**

* **Engine (Motor):** Programa que ejecuta código JS. *Ejemplo:* V8 (de Google), SpiderMonkey (de Mozilla).
* **Runtime (Entorno de Ejecución):** El ecosistema completo donde se ejecuta el código. *Ejemplo:* Un navegador (contiene el Engine + APIs del DOM, fetch, etc.) o Node.js (contiene el Engine + APIs del sistema de archivos, red, etc.).

**Descarte de Opciones Incorrectas:**

* **Web Browser is a Runtime, Chrome Browser is the Engine:** Incorrecto. Chrome es el Runtime (el navegador completo). Su Engine es V8.
* **They refer to the same thing:** Incorrecto. El Engine es una parte fundamental del Runtime.

**Pregunta 4: Event Loop y Microtasks**

**La Pregunta:**

setTimeout(() => {console.log('a')}, 100); // Macrotask - 100ms

setTimeout(() => {console.log('b')}, 0); // Macrotask - 0ms

Promise.resolve('c').then(console.log); // Microtask

console.log('d'); // Síncrono  
  
**Respuesta Correcta:** d, c, b, a

**Conceptos Fundamentales a Dominar:**

1. **Modelo de Concurrencia de JavaScript:** Entender que JavaScript es de un solo hilo (single-threaded) pero puede manejar operaciones asíncronas gracias al Event Loop.
2. **Call Stack, Microtask Queue, Macrotask Queue:** Conocer la jerarquía y el orden en que se procesan estas tres estructuras.

**Análisis Detallado de la Lógica del Código:**  
El intérprete de JavaScript procesa el código siguiendo un orden de prioridad estricto:

1. **Ejecución Síncrona (Call Stack):** Se ejecuta todo el código principal de arriba a abajo.
   * console.log('d') se ejecuta inmediatamente. **Salida: d**.
2. **Procesamiento de Microtareas (Microtask Queue):** Una vez que la pila de llamadas está vacía, el motor revisa si hay tareas en la cola de microtareas. Si las hay, las ejecuta **todas** hasta que la cola esté vacía.
   * Promise.resolve('c').then(console.log) añade una tarea a esta cola.
   * Se ejecuta a continuación. **Salida: c**.
3. **Procesamiento de Macrotareas (Macrotask Queue):** Solo cuando la cola de microtareas está vacía, el Event Loop toma **una sola tarea** de la cola de macrotareas para ejecutarla.
   * setTimeout(..., 0) añade una tarea a esta cola. Se ejecuta ahora. **Salida: b**.
   * El Event Loop vuelve a empezar su ciclo (revisa microtareas, etc.). Luego, 100ms más tarde, el timer de setTimeout(..., 100) termina y añade su tarea a la cola. En un ciclo posterior, se ejecuta. **Salida: a**.

**Explicación de Métodos y Sintaxis Clave:**

* **console.log():** Código síncrono.
* **.then() de una Promesa:** Su callback siempre se agenda como una **microtarea**.
* **setTimeout():** Su callback siempre se agenda como una **macrotarea**. Un delay de 0 no significa "inmediato", significa "agéndalo en la cola de macrotareas tan pronto como sea posible".

**Pregunta 5: call, apply y bind**

**La Pregunta:**

const wizard = {

name: 'Merlin',

health: 100,

heal(num1, num2) {

this.health += num1 + num2;

}

}

const archer = {

name: 'Robin Hood',

health: 50

}

// ¿Cómo hacer que archer.health sea 100?  
  
**Respuesta Correcta:** wizard.heal.call(archer, 30, 20)

**Conceptos Fundamentales a Dominar:**

1. **Contexto de Ejecución (this):** Entender que el valor de this en JavaScript es dinámico y depende de *cómo* se llama a una función.
2. **Reutilización de Métodos:** Técnicas para "prestar" un método de un objeto para que sea usado por otro.

**Análisis Detallado de la Lógica del Código:**  
El objetivo es ejecutar el método heal, que es propiedad de wizard, pero de tal manera que this.health dentro del método se refiera a archer.health. Para esto, necesitamos establecer explícitamente el contexto (this).

**Explicación de Métodos y Sintaxis Clave:**

* **function.prototype.call(thisArg, arg1, arg2, ...):**
  + **Qué hace:** Invoca una función **inmediatamente**.
  + **thisArg:** El valor que se asignará a this durante la ejecución de la función. En nuestro caso, archer.
  + **arg1, arg2, ...:** Una lista de argumentos que se pasarán a la función, separados por comas. En nuestro caso, 30 y 20.
  + *Uso:* Perfecto para cuando quieres llamar a una función con un this específico y tienes los argumentos listos de forma individual.
* **function.prototype.apply(thisArg, [argsArray]):**
  + **Qué hace:** Invoca una función **inmediatamente**, igual que call.
  + **thisArg:** El valor para this.
  + **[argsArray]:** La principal diferencia. Los argumentos se pasan como un **único array**.
  + *Uso:* Muy útil cuando los argumentos que quieres pasar ya los tienes en un array o cuando el número de argumentos es dinámico.
* **function.prototype.bind(thisArg, arg1, arg2, ...):**
  + **Qué hace:** **No invoca** la función. En su lugar, **crea y devuelve una nueva función**. Esta nueva función tiene su this permanentemente "atado" (bound) a thisArg.
  + *Uso:* Ideal para cuando necesitas pasar una función como callback o asignarla a un evento, asegurándote de que siempre se ejecute con el contexto correcto, sin importar cómo se llame más tarde.
  + *Ejemplo:* const healTheArcher = wizard.heal.bind(archer); healTheArcher(30, 20);

**Descarte de Opciones Incorrectas:**

* **wizard.heal.apply(archer, 30, 20):** Sintaxis incorrecta. apply requiere un array para los argumentos: apply(archer, [30, 20]).
* **archer.heal(30, 20):** Error. El objeto archer no tiene un método heal.
* **archer.heal += 50:** Lógica incorrecta. Esto es una operación matemática, no una llamada a una función.

**Pregunta 6: Prototype Override**

**La Pregunta:**

Array.prototype.map = function() {

arr = [];

for (let i = 0; i < this.length; i++) {

arr.push((this[i] + '!'));

}

return arr;

}

answer = [1,2,3].map(x => x\*2);  
  
**Respuesta Correcta:** ["1!", "2!", "3!"]

**Conceptos Fundamentales a Dominar:**

1. **Cadena de Prototipos (Prototype Chain):** El mecanismo por el cual los objetos en JavaScript heredan características unos de otros.
2. **Mutabilidad de los Prototipos:** Entender que los prototipos de los objetos nativos (Array, Object, String) son modificables, aunque es una práctica generalmente desaconsejada en producción.

**Análisis Detallado de la Lógica del Código:**

1. **Array.prototype.map = function() { ... }:** Esta línea accede al "manual de instrucciones" de todos los arrays y reemplaza la definición del método map por una nueva.
2. **La Nueva Implementación:** Nuestra función map personalizada ignora por completo cualquier argumento que se le pase (la función x => x \* 2 es desechada). Su única lógica es:
   * Crear un array vacío (arr).
   * Recorrer el array sobre el que se llama (this).
   * Para cada elemento, concatenarle un ! y meterlo en el nuevo array.
   * Devolver el nuevo array.
3. **La Llamada:** Cuando se ejecuta [1, 2, 3].map(...), JavaScript busca el método map en el prototipo de Array y encuentra nuestra versión modificada, la cual ejecuta, dando como resultado ["1!", "2!", "3!"].

**Pregunta 7: Array Mutation**

**La Pregunta:**

const basket = [1, 2, 3];

// ¿Cuáles opciones MUTAN el array original?  
  
**Respuestas Correctas:**

basket[3] = 4

const grow = basket.push.bind(basket); grow(4);

**Conceptos Fundamentales a Dominar:**

1. **Mutabilidad de los Arrays:** A diferencia de los tipos primitivos, los arrays son objetos mutables, lo que significa que su contenido puede cambiar.
2. **Métodos de Array Mutadores:** Identificar qué métodos modifican el array original (push, pop, splice) y cuáles devuelven uno nuevo (map, filter, slice).

**Análisis Detallado de la Lógica del Código:**  
El objetivo es añadir el número 4 al final del array [1, 2, 3].

* **basket[3] = 4:** Los arrays en JavaScript son "dispersos". Al asignar un valor a un índice que está justo después del último elemento existente, el array simplemente se extiende para incluirlo. Esto es una forma directa de mutar el array.
* **basket.push.bind(basket):**
  + basket.push es un método que añade elementos al final del array.
  + .bind(basket) crea una nueva función (grow) que, cuando sea llamada, ejecutará push con this fijado permanentemente a basket.
  + Por lo tanto, grow(4) es funcionalmente idéntico a basket.push(4), y modifica el array.

**Descarte de Opciones Incorrectas:**

* **basket[3] = [4]:** Esto añadiría un array como cuarto elemento: [1, 2, 3, [4]].
* **const grow2 = ...; grow(2):** La función grow2 es correcta, pero la llamada grow(2) añadiría un 2 al array, no un 4.

**Pregunta 8: Hoisting**

**La Pregunta:**

a();

function a() { console.log("I'm Spartacus"); }

function a() { console.log("No! I'm Spartacus"); }

var a = () => console.log("You're all wrong. I'm Spartacus");  
  
**Respuesta Correcta:** "No! I'm Spartacus"

**Conceptos Fundamentales a Dominar:**

1. **Hoisting (Elevación):** El proceso del motor de JavaScript de mover las declaraciones al principio de su ámbito antes de la ejecución.
2. **Precedencia de Declaraciones:** Las declaraciones de función (function name(){}) tienen mayor prioridad y sobrescriben a las declaraciones de variable (var name;) durante el hoisting.

**Análisis Detallado de la Lógica del Código:**  
El motor de JavaScript procesa el código en dos fases:

1. **Fase de Creación (Hoisting):**
   * function a() { console.log("I'm Spartacus") } es elevada.
   * function a() { console.log("No! I'm Spartacus") } es elevada y, como tiene el mismo nombre, **sobrescribe a la anterior**.
   * var a se eleva como var a;, pero como ya existe una propiedad a con mayor precedencia (la función), esta declaración es ignorada en la práctica.
   * Al final de esta fase, en memoria, a es la función que imprime "No! I'm Spartacus".
2. **Fase de Ejecución:**
   * a();: Se ejecuta la primera línea. Llama a la versión de a que existe en memoria en ese momento. **Salida: "No! I'm Spartacus"**.
   * var a = () => ...: Mucho después, se llega a esta línea y la variable a es **reasignada** a la nueva función flecha, pero esto ocurre después de que la llamada original ya se ha hecho.

**Pregunta 9: Shallow vs. Deep Copy**

**La Pregunta:**

let obj = {

a: 'a',

b: 'b',

c: { deep: 'try and copy me' }

};

let clone = Object.assign({}, obj); // Shallow copy

let clone2 = {...obj}; // Shallow copy

let clone3 = JSON.parse(JSON.stringify(obj)); // Deep copy

obj.c.deep = 'hahaha';  
  
**Respuesta Correcta:** La primera opción.

Generated code

{a: "a", b: "b", c: {deep: "hahaha"}}

{a: "a", b: "b", c: {deep: "hahaha"}}

{a: "a", b: "b", c: {deep: "hahaha"}}

{a: "a", b: "b", c: {deep: "try and copy me"}}

**Conceptos Fundamentales a Dominar:**

1. **Copia por Valor vs. Copia por Referencia:** Los tipos primitivos se copian por valor (se crea un duplicado). Los objetos se copian por referencia (se crea un nuevo "puntero" al mismo objeto en memoria).
2. **Estructuras de Datos Anidadas:** Entender cómo las operaciones de copia afectan a los objetos que están dentro de otros objetos.

**Análisis Detallado de la Lógica del Código:**  
El punto clave es el objeto anidado c.

* **clone y clone2 (Shallow Copy):** Object.assign y la sintaxis de spread (...) crean una **copia superficial**. Crean un nuevo objeto de nivel superior, pero si una de sus propiedades es un objeto (como c), simplemente copian la **referencia** a ese objeto. Esto significa que obj.c, clone.c y clone2.c son tres nombres diferentes para el **mismo objeto** en memoria.
* **clone3 (Deep Copy):** El truco JSON.parse(JSON.stringify(obj)) crea una **copia profunda**. Primero convierte todo el objeto, incluyendo las estructuras anidadas, en una cadena de texto (rompiendo todas las referencias). Luego, reconstruye un objeto completamente nuevo a partir de esa cadena. clone3 y su sub-objeto c son totalmente independientes del original.
* **La Modificación:** Cuando se ejecuta obj.c.deep = 'hahaha', se está modificando el objeto anidado original. Como clone y clone2 comparten la referencia a ese mismo objeto, el cambio se refleja en ellos también. clone3, al ser independiente, no se ve afectado.

**Pregunta 10: try-catch y var Scope**

**La Pregunta:**  
(function () {

try {

throw new Error(6);

} catch (err) {

var err = 5;

var boo = 10;

console.log(err); // ?

}

console.log(err); // ?

console.log(boo); // ?

})();  
  
**Respuesta Correcta:** None of the above (La salida real es 5, 5, 10).

**Conceptos Fundamentales a Dominar:**

1. **Ámbito de var (Function Scope):** Las variables declaradas con var son accesibles en toda la función, sin importar en qué bloque (if, for, catch) se declaren.
2. **Shadowing de Variables:** Cuando una variable en un ámbito interno tiene el mismo nombre que una en un ámbito externo.

**Análisis Detallado de la Lógica del Código:**  
Este es un caso peculiar de "shadowing" y hoisting.

1. **Hoisting:** Antes de ejecutar, var err y var boo se elevan al principio de la IIFE con valor undefined.
2. **Bloque try:** throw new Error(6) envía la ejecución al bloque catch.
3. **Bloque catch (err):**
   * El err en catch(err) es una variable especial con **ámbito de bloque**. En este momento, contiene el objeto Error(6).
   * **var err = 5;:** ¡Aquí está la clave! var no respeta el ámbito de bloque. En lugar de crear una nueva variable, busca una variable err en el ámbito de la función (que existe gracias al hoisting) y le **asigna el valor 5**. Esta variable de función ahora "ensombrece" (shadows) al parámetro err del catch.
   * console.log(err);: Lee la variable err más cercana, que ahora es la de la función con valor 5. **Salida: 5**.
4. **Fuera del catch:**
   * console.log(err);: Se lee la variable err de ámbito de función, que sigue teniendo el valor 5. **Salida: 5**.
   * console.log(boo);: Se lee la variable boo (también elevada), que fue asignada a 10. **Salida: 10**.

**Pregunta 11: Array.reduce()**

**La Pregunta:**

const userArray = [

{name: 'Tom', type: 'cat'},

{name: 'Jerry', type: 'mouse'}

];

function convertReducer(acc, person) {

return {...acc, [person.name]: person};

}

console.log(userArray.reduce(convertReducer, {}));  
  
**Respuesta Correcta:** { Tom: {name: "Tom", type: "cat"}, Jerry: {name: "Jerry", type: "mouse"} }

**Conceptos Fundamentales a Dominar:**

1. **Programación Funcional:** Paradigma que favorece el uso de funciones para transformar datos.
2. **Transformación de Datos:** Convertir una estructura de datos (un array) en otra (un objeto).

**Análisis Detallado de la Lógica del Código:**  
El objetivo es convertir un array de objetos en un solo objeto, donde cada name del usuario se convierte en una clave.

* **Array.prototype.reduce(callback, valorInicial):** Itera sobre un array para "reducirlo" a un único valor.
  + **valorInicial:** En este caso, es un objeto vacío {}. Este es el valor inicial de nuestro acumulador.
  + **callback(acc, person):** La función que se ejecuta en cada iteración.
    - acc: El objeto que estamos construyendo.
    - person: El elemento actual del array ({name: 'Tom', ...}).
  + **return {...acc, [person.name]: person}:** Esta línea es el corazón de la transformación.
    - {...acc}: Copia todas las propiedades del objeto acumulador actual.
    - [person.name]: **Computed Property Name**. Usa el valor de person.name (ej: "Tom") como la nueva clave del objeto.
    - : person: Asigna el objeto person completo como el valor de esa nueva clave.

**Pregunta 12: let y Block Scope**

**La Pregunta:**

function loop() {

for(let i = 0; i < 5; i++) {

// código

}

console.log(i); // ?

}  
  
**Respuesta Correcta:** ReferenceError: i is not defined

**Conceptos Fundamentales a Dominar:**

1. **Ámbito de Bloque (Block Scope):** El concepto de que las variables declaradas con let y const solo existen dentro del par de llaves {} más cercano.

**Análisis Detallado de la Lógica del Código:**

* **for(let i = 0; ...):** La declaración let i crea una variable i cuyo ámbito está **estrictamente limitado al bloque del bucle for**.
* El bucle se ejecuta 5 veces. En cada iteración, i existe y tiene un valor (0, 1, 2, 3, 4).
* Una vez que la condición i < 5 se vuelve falsa, el bucle termina. En ese instante, la variable i es destruida y deja de existir.
* **console.log(i):** Esta línea está **fuera** del bloque del bucle. Intenta acceder a una variable que ya no está en el ámbito, lo que provoca un ReferenceError.

**Pregunta 13: Function Composition (Pipe)**

**La Pregunta:**

const pipe = (f, g, h) => (a) => h(g(f(a)));

const less5 = (input) => input - 5;

const times2 = (input) => input \* 2;

const less1 = (input) => input - 1;

// ¿Cuál llamada da como resultado -3?  
  
**Respuesta Correcta:** pipe(less5, times2, less1)(4)

**Conceptos Fundamentales a Dominar:**

1. **Composición de Funciones:** Crear funciones complejas combinando funciones más simples. Es un pilar de la programación funcional que promueve la reutilización y la legibilidad.
2. **Funciones de Orden Superior:** Funciones que toman otras funciones como argumentos o las devuelven como resultado. pipe es una función de orden superior.

**Análisis Detallado de la Lógica del Código:**  
La función pipe está definida para ejecutar las funciones en el orden en que se pasan (f, luego g, luego h).

* const pipe = (f, g, h) => (a) => h(g(f(a)))  
  El objetivo es obtener -3 a partir de 4. Sigamos el flujo de datos:

1. **f(a):** La primera función (f es less5) se aplica al argumento (a es 4). less5(4) -> -1.
2. **g(f(a)):** La segunda función (g es times2) se aplica al resultado anterior. times2(-1) -> -2.
3. **h(g(f(a))):** La tercera función (h es less1) se aplica al resultado anterior. less1(-2) -> -3.  
   La única opción que sigue este orden y sintaxis es la segunda.

**Pregunta 14: Currying**

**La Pregunta:**

const multiplyByTwo = multiplyBy(null)(2);

multiplyByTwo(4); // 8

// ¿Cómo debe ser la función multiplyBy?  
  
**Respuesta Correcta:** La tercera opción (o su equivalente con la palabra clave function).  
const multiplyBy = (x) => (num1) => (num2) => num1 \* num2;

**Conceptos Fundamentales a Dominar:**

1. **Currying:** La técnica de transformar una función que toma múltiples argumentos en una secuencia de funciones anidadas, cada una tomando un solo argumento.
2. **Closures:** La función interna "recuerda" los argumentos de las funciones externas.

**Análisis Detallado de la Lógica del Código:**  
La llamada multiplyBy(null)(2) debe devolver una función.

* **multiplyBy(null):** Llama a la función más externa. El argumento x (null) es ignorado. La función devuelve la siguiente función en la cadena: (num1) => (num2) => num1 \* num2.
* **(2):** Llama a la función devuelta, pasándole 2 como num1. Esta llamada, a su vez, devuelve la función más interna: (num2) => 2 \* num2. Esta función se almacena en multiplyByTwo.
* **multiplyByTwo(4):** Ejecuta la función almacenada, pasándole 4 como num2. Se ejecuta 2 \* 4, devolviendo 8.

**Pregunta 15: Closures y Memory**

**La Pregunta:**

const runCalculation = someFunc();

runCalculation(699);

runCalculation(701);

runCalculation(900);

function someFunc() {

const bigArray = new Array(7000).fill('😄');

return function(item) {

return bigArray[item];

}

}

// ¿Cuántas veces se crea bigArray?  
  
**Respuesta Correcta:** Once

**Conceptos Fundamentales a Dominar:**

1. **Closures (Clausuras):** La capacidad de una función para "recordar" y acceder a las variables de su ámbito de creación, incluso después de que ese ámbito haya desaparecido.
2. **Ciclo de Vida de las Funciones:** Entender la diferencia entre declarar una función y ejecutarla.

**Análisis Detallado de la Lógica del Código:**

1. **const runCalculation = someFunc();:** La función someFunc se **ejecuta una sola vez** en esta línea.
2. Dentro de esa única ejecución, const bigArray = new Array(...) se ejecuta, creando el array en memoria **una sola vez**.
3. someFunc retorna una nueva función. Esta función retornada **forma un closure**, manteniendo una referencia viva al bigArray.
4. Las llamadas posteriores (runCalculation(699), etc.) ejecutan la función retornada. Esta función accede al bigArray que "recuerda" gracias al closure. El array no se vuelve a crear.

**Pregunta 16: Pure Functions**

**La Pregunta:**

function one(input) {

console.log(input); // ❌ Efecto secundario

}

function two(input = 5) {

return 5 \* input; // ✅ Determinista, sin efectos secundarios

}

function three(input) {

return two(); // ✅ Siempre retorna 25

}

function four(input) {

return one(new Date()); // ❌ new Date() no es determinista

}

// ¿Cuáles son Pure Functions?  
  
**Respuestas Correctas:** two y three.

**Conceptos Fundamentales a Dominar:**

1. **Funciones Puras:** Son la base de la programación funcional. Una función es pura si cumple dos reglas:
   * **Determinista:** Para la misma entrada, siempre produce la misma salida.
   * **Sin Efectos Secundarios (Side Effects):** No modifica ningún estado fuera de su propio ámbito (no cambia variables globales, no escribe en la consola, no hace llamadas a la red).

**Análisis Detallado de la Lógica del Código:**

* **one:** Impura. console.log es un efecto secundario (modifica un estado externo: la consola).
* **two:** Pura. Es determinista (5 \* input siempre dará lo mismo) y no tiene efectos secundarios.
* **three:** Pura. Siempre llama a two() sin argumentos, por lo que siempre devuelve 25. Es determinista y no tiene efectos secundarios.
* **four:** Impura. new Date() no es determinista (su resultado depende del momento en que se ejecuta). Además, llama a one, que es impura.

**Pregunta 17: setTimeout y var vs. let**

**La Pregunta:**

const array = [1,2,3,4];

// Loop con var

for(var i=0; i < array.length; i++) {

setTimeout(function(){

console.log(i);

}, 3000);

}

// Loop con let

for(let i=0; i < array.length; i++) {

setTimeout(function(){

console.log(i);

}, 400);

}  
  
**Respuesta Correcta:** 0, 1, 2, 3, 4, 4, 4, 4

**Conceptos Fundamentales a Dominar:**

1. **Ámbito de var vs. let:** Repaso de la diferencia entre ámbito de función y ámbito de bloque.
2. **Closures en Bucles:** Cómo los callbacks asíncronos "capturan" el valor de las variables del bucle.

**Análisis Detallado de la Lógica del Código:**

* **Bucle con let (400ms):**
  + let tiene ámbito de bloque. En cada iteración, se crea una **nueva variable i independiente**.
  + El callback de cada setTimeout forma un closure sobre su propia copia de i.
  + Por eso, cuando se ejecutan, imprimen los valores que capturaron: 0, 1, 2, 3.
* **Bucle con var (3000ms):**
  + var tiene ámbito de función. Hay **una única variable i** para todo el bucle.
  + El bucle se completa muy rápido, y el valor final de i es 4.
  + Cuando los callbacks se ejecutan 3 segundos después, todos acceden a la misma y única variable i, que en ese momento vale 4.
  + Por eso, todos imprimen 4.
* **Orden de Salida:** Los callbacks de 400ms se ejecutan mucho antes que los de 3000ms.

**Pregunta 18: Array Deduplication**

**La Pregunta:**

// ¿Cuáles funciones FUNCIONAN para eliminar duplicados?

function removeDuplicate(arr) {

const cache = {};

return arr.filter(item => {

if (cache[item]) {

return false;

}

cache[item] = true;

return true;

});

}

function removeDuplicate2(arr) {

return [...new Set(arr)];

}

function removeDuplicate3(arr) {

return arr.filter((item, index) => arr.indexOf(item) === index);

}

function removeDuplicate4(arr) {

arr.forEach(item => {

// código

});

}  
  
**Respuestas Correctas:** removeDuplicate, removeDuplicate2, removeDuplicate3.

**Conceptos Fundamentales a Dominar:**

1. **Métodos de Array:** Conocer el propósito y el valor de retorno de métodos como filter y forEach.
2. **Estructuras de Datos:** Entender los casos de uso de Object como hash map y de Set para almacenar valores únicos.

**Análisis Detallado de la Lógica del Código:**

* **removeDuplicate3 (Filter + indexOf):** Funciona porque arr.indexOf(item) siempre devuelve el índice de la **primera** aparición de item. La condición arr.indexOf(item) === index solo será verdadera para la primera vez que se encuentra cada elemento, filtrando así los duplicados posteriores.
* **removeDuplicate2 (Set):** Funciona porque la estructura de datos Set por definición solo puede contener valores únicos. Es la forma más moderna y legible de eliminar duplicados.
* **removeDuplicate (Cache):** Funciona usando un objeto como un hash map para llevar un registro de los elementos ya vistos. Es una implementación manual eficiente de la lógica de un conjunto.
* **removeDuplicate4 (forEach):** **No funciona** porque el método forEach **siempre devuelve undefined**. Se usa para ejecutar una acción por cada elemento, no para construir un nuevo array.

**Pregunta 19: Function Chaining**

**La Pregunta:**

function sayMySecret() {

const a = 'a';

return function findSecret() {

const b = 'b';

return function printSecret() {

const c = 'c';

return 'The Secret';

}

}

}

// ¿Cómo obtener "The Secret"?  
  
**Respuesta Correcta:** sayMySecret()()()

**Conceptos Fundamentales a Dominar:**

1. **Funciones de Orden Superior:** Funciones que pueden retornar otras funciones.
2. **Sintaxis de Invocación de Funciones:** Entender que el operador () es lo que ejecuta una función.

**Análisis Detallado de la Lógica del Código:**  
La estructura es una serie de funciones anidadas, donde cada función externa retorna la función interna.

1. **sayMySecret():** Ejecuta la función más externa. Su return es la función findSecret.
2. **() (segundo par):** Ejecuta la función que se acaba de retornar (findSecret). Su return es la función printSecret.
3. **() (tercer par):** Ejecuta la función que se acaba de retornar (printSecret). Su return es el valor final, la cadena 'The Secret'.  
   Cada () invoca el resultado de la expresión anterior.

**Pregunta 20: Time Complexity**

**La Pregunta:**

function findFirstRecurringCharacter1(input) {

for (let i = 0; i < input.length; i++) { // n iteraciones

for (let j = i + 1; j < input.length; j++) { // n iteraciones por cada i

if(input[i] === input[j]) {

return input[i];

}

}

}

return undefined;

}

function findFirstRecurringCharacter2(input) {

let map = {};

for (let i = 0; i < input.length; i++) { // n iteraciones

if (map[input[i]] !== undefined) { // O(1) lookup

return input[i];

} else {

map[input[i]] = i; // O(1) assignment

}

}

return undefined;

}

// ¿Cuál es más eficiente?  
  
**Respuesta Correcta:** findFirstRecurringCharacter2

**Conceptos Fundamentales a Dominar:**

1. **Notación Big O:** Una forma de describir la eficiencia de un algoritmo en términos de cómo escala su tiempo de ejecución o uso de memoria a medida que aumenta el tamaño de la entrada (n).
2. **Eficiencia de Estructuras de Datos:** Saber que la búsqueda de claves en un objeto/hash map es una operación muy rápida (O(1) en promedio).

**Análisis Detallado de la Lógica del Código:**

* **findFirstRecurringCharacter1 (Complejidad Cuadrática - O(n²)):**
  + Utiliza dos bucles anidados. Por cada elemento (i), recorre el resto del array (j).
  + El número de operaciones crece exponencialmente con el tamaño de la entrada. Si n es 1000, hará aproximadamente 1,000,000 de operaciones. Es ineficiente para entradas grandes.
* **findFirstRecurringCharacter2 (Complejidad Lineal - O(n)):**
  + Utiliza un objeto (map) como un hash map.
  + Recorre el array **una sola vez**.
  + En cada iteración, la comprobación if (map[input[i]] !== undefined) es una búsqueda en el hash map, que es una operación de tiempo constante (O(1)) en promedio.
  + El número total de operaciones es directamente proporcional al tamaño de la entrada. Si n es 1000, hará aproximadamente 1000 operaciones. Es mucho más eficiente.