TESTING-US

VALIDÁ TU APRENDIZAJE

then
await
this

CUADERNILLO DE ESTUDIO · NIVEL AVANZADO

JavaScript

asincronía y closures

Closures, this, el bucle de eventos, promesas y async/await: los temas que más se preguntan en las entrevistas técnicas.

1 Alcance y closures

2 this, call, apply y bind

3 El bucle de eventos

4 Promesas

5 async y await

\+ Ejercicios con respuestas

Testing-US · Validá tu aprendizaje
testing-us.com

Edición octubre 2026

Antes de empezar

Esta evaluación es la continuación de "JavaScript: fundamentos". Cubre lo que separa a quien escribe JavaScript de quien entiende por qué funciona como funciona: closures, this, el bucle de eventos y el código asincrónico con promesas y async/await. Son temas fijos en las entrevistas técnicas para puestos de desarrollo web.

CÓMO APROVECHARLO

- Probá cada ejemplo en la consola del navegador (F12) y cambiá los valores para ver qué pasa.
- Hacé los ejercicios del final de cada tema antes de mirar las respuestas.
- Cuando termines, rendí la evaluación **JavaScript: asincronía y closures** en testing-us.com.

TIP Antes de ejecutar un ejemplo, tratá de adivinar qué va a mostrar. Así se entrena justo lo que pide la evaluación.

Contenido

1. **Alcance y closures**
   Alcance léxico, closures, datos privados, IIFE y la trampa de var.
2. **this, call, apply y bind**
   Métodos, funciones sueltas, new, flechas, clases y campos privados.
3. **El bucle de eventos**
   Un solo hilo, tareas y microtareas, orden de ejecución, temporizadores.
4. **Promesas**
   Estados, then, catch, finally, all, allSettled, race y any.
5. **async y await**
   Errores con try/catch, fetch, cancelación, serie y paralelo.

TEMA 1

1 Alcance y closures

LO QUE VAS A APRENDER

- Entender el alcance léxico.
- Saber qué es un closure y qué recuerda.
- Usar closures para datos privados y fábricas de funciones.
- Evitar la trampa de var en los bucles.

Alcance léxico

- Cada función crea un **alcance** nuevo, con sus propias variables locales, cada vez que se llama. `let` y `const` además crean alcance en cada bloque `{}`.
- El alcance es **léxico**: una función ve las variables del lugar donde fue **escrita**, no del lugar desde donde se la llama. Si `f` usa `x` y fue escrita donde `x` vale 1, devuelve 1 aunque se la llame desde otra función que tiene su propia `x`.
- Una variable interna con el mismo nombre que una externa la **oculta** (*shadowing*) dentro de su bloque, sin cambiar la de afuera.
- En modo estricto, asignar a una variable que nunca se declaró (`total = 5;`) da `ReferenceError`. Sin modo estricto, crearía una variable global por accidente.
- En un **módulo** (`<script type="module">` o archivos con `import` / `export`), las variables del nivel superior existen solo dentro del módulo, salvo que se exporten. Los módulos siempre están en modo estricto.

Qué es un closure

- Un **closure** es una función junto con las variables del alcance donde fue creada. La función **recuerda** esas variables aunque ese alcance ya haya terminado.
- Ejemplo de contador: `function crear() { let c = 0; return () => ++c; }`. Con `const f = crear();`, cada llamada a `f()` da 1, 2, 3...: `c` sigue viva dentro del closure.
- Cada llamada a `crear()` crea un `c` **nuevo**: dos contadores hechos con `crear` son independientes.
- El closure guarda una **referencia** a la variable, no una copia de su valor: si la variable cambia después, la función ve el valor nuevo.
- Las variables que usa un closure siguen en memoria mientras el closure exista.
- Las funciones flecha también forman closures.

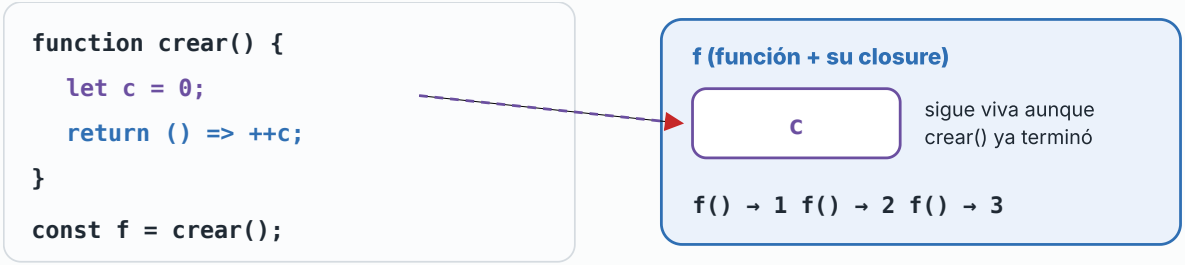


Figura 1.1 La función devuelta recuerda la variable c.

Usos habituales

- **Datos privados:** las variables del closure no se pueden leer desde afuera. `function cuenta() { let saldo = 0; return { depositar(m) { saldo += m; return saldo; } }; }` : `cuenta().saldo` da `undefined`; solo `depositar` puede tocar `saldo`.

- **Fábricas de funciones:** `const sumar = a => b => a + b;` hace que `sumar(2)` sea una función que suma 2: `sumar(2)(3)` da `5`.
- **Ejecutar una sola vez:** una función que guarda en el closure si ya se ejecutó y, si es así, no vuelve a hacerlo.
- **IIFE** (*expresión de función invocada inmediatamente*): `(function () { ... })()` ejecuta el código en el momento y crea un alcance privado. Antes de los módulos era la forma de no llenar el alcance global.
- **Manejadores de eventos:** `botones.forEach((b, i) => b.addEventListener("click", () => console.log(i)))` hace que cada botón recuerde su propio `i`.

La trampa de var en los bucles

- `for (var i = 0; i < 3; i++) { setTimeout(() => console.log(i)); }` muestra **3, 3, 3**: hay **una sola** variable `i` (var no tiene alcance de bloque) y cuando se ejecutan los `setTimeout` el bucle ya terminó.
- Con `let` muestra **0, 1, 2**: en un `for`, `let` crea una variable **nueva en cada vuelta**, y cada función recuerda la suya.

TIP En entrevistas es muy común la pregunta del for con var y setTimeout. Explicá por qué da 3, 3, 3 y cómo lo arregla let.

Para practicar

1. ¿Qué es un closure? Dá un ejemplo de uso.
2. Con el contador del cuadernillo, ¿qué dan a(), a() y b() si a y b se crearon por separado?
3. ¿Por qué el for con var y setTimeout muestra 3, 3, 3?
4. ¿Cómo harías una variable privada sin usar clases?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 2

2 this, call, apply y bind

LO QUE VAS A APRENDER

- Saber qué vale this según cómo se llama la función.
- Entender por qué se pierde this al pasar un método.
- Usar call, apply y bind.
- Elegir entre función común y flecha.

this depende de cómo se llama la función

- En una función común, `this` **no** depende de dónde se escribió, sino de **cómo se la llama**.
- **Como método** (`obj.metodo()`): `this` es el objeto que está antes del punto.
- **Sola** (`f()`): en modo estricto `this` es `undefined` . Por eso, si se guarda un método en una variable (`const f = o.saludar; f();`), se "pierde" el objeto y leer `this.n` da `TypeError` .
- **Con new** (`new Persona("Sol")`): `this` es el objeto nuevo que se está creando.
- **Como listener** con una función común (`boton.addEventListener("click", function () { … })`): `this` es el elemento que tiene el listener.
- Pasar un método como callback (`setTimeout(o.saludar, 100)`) también pierde el `this` (en el navegador pasa a ser `window`).

Cómo se llama	Qué vale this
`o.metodo()`	el objeto o (lo que está antes del punto)
`f() (función sola)`	undefined en modo estricto
`new F()`	el objeto nuevo
`f.call(x) · f.apply(x) · f.bind(x)`	x (lo elegís vos)
`() => { … } (flecha)`	el this de afuera, donde fue escrita

Figura 2.1 El valor de this según la forma de llamar.

Funciones flecha

- Las funciones flecha **no tienen** `this` **propio**: usan el `this` del alcance donde fueron escritas. En el nivel superior de un módulo, ese `this` es `undefined` .
- Por eso son útiles dentro de un método: en `f() { return [1, 2].map(x => x * this.n); }` , la flecha usa el `this` del método.
- Y por eso **no conviene** usarlas como métodos de un objeto ni como listeners cuando se necesita `this` : tomarían el `this` de afuera.
- Una función flecha no se puede usar con `new` : da `TypeError` .

call, apply y bind

- `f.call(obj, a, b)` llama a `f` con `this = obj` y los argumentos **separados**.

- `f.apply(obj, [a, b])` hace lo mismo pero recibe los argumentos en un **arreglo**. Ejemplo: `Math.max.apply(null, [1, 5, 3])` da `5` (hoy también se escribe `Math.max(...[1, 5, 3])`).
- `f.bind(obj)` **no ejecuta** la función: devuelve una **nueva función** con `this` fijado en `obj`. Sirve para callbacks: `setTimeout(o.saludar.bind(o), 100)`.
- Una función creada con `bind` queda fijada para siempre: volver a aplicarle `bind` no cambia su `this` (solo `new` lo ignora).

Clases

- `class Contador { #c = 0; inc() { return ++this.#c; } }`: el `#` indica un **campo privado**, que solo se puede usar dentro de la clase. Escribir `cont.#c` afuera es un error de sintaxis.
- El código dentro de una clase siempre corre en modo estricto.
- Un método que termina con `return this` permite **encadenar** llamadas: `c.sumar(2).sumar(3)`.

Para practicar

1. ¿Qué vale this en o.metodo() y en f() en modo estricto?
2. ¿Qué diferencia hay entre call, apply y bind?
3. ¿Por qué setTimeout(o.saludar, 100) pierde el this? Dá dos soluciones.
4. ¿Cuándo conviene usar una función flecha dentro de un método?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 3

3 El bucle de eventos

LO QUE VAS A APRENDER

- Entender por qué JavaScript tiene un solo hilo.
- Diferenciar tareas y microtareas.
- Predecir el orden de ejecución.
- Elegir entre setTimeout, requestAnimationFrame y Web Workers.

Un solo hilo

- JavaScript ejecuta el código en **un solo hilo**: hay una **pila de llamadas** (*call stack*) y se ejecuta una cosa a la vez.
- Si una función tarda mucho (por ejemplo, un bucle que espera varios segundos), **bloquea** la página: no se dibuja nada ni se atienden clics hasta que termina.
- Lo que tarda (temporizadores, pedidos a la red, eventos) lo maneja el **entorno** (el navegador o Node), fuera de la pila. Cuando termina, deja una función para ejecutar en una **cola**.

Tareas y microtareas

- **Tareas** (*macrotasks*): las callbacks de `setTimeout`, `setInterval`, eventos del usuario, mensajes.
- **Microtareas**: las callbacks de las promesas (`then`, `catch`, `finally`), lo que sigue a un `await` y `queueMicrotask`.
- El **bucle de eventos** repite: ejecutar una tarea; cuando la pila queda vacía, ejecutar **todas** las microtareas pendientes (incluidas las que se agreguen mientras tanto), en el orden en que llegaron; después, si hace falta, redibujar la pantalla; y pasar a la tarea siguiente.
- Consecuencia: las callbacks de promesas que ya están en la cola de microtareas **se ejecutan antes** que cualquier `setTimeout` pendiente, aunque sea de 0 ms.

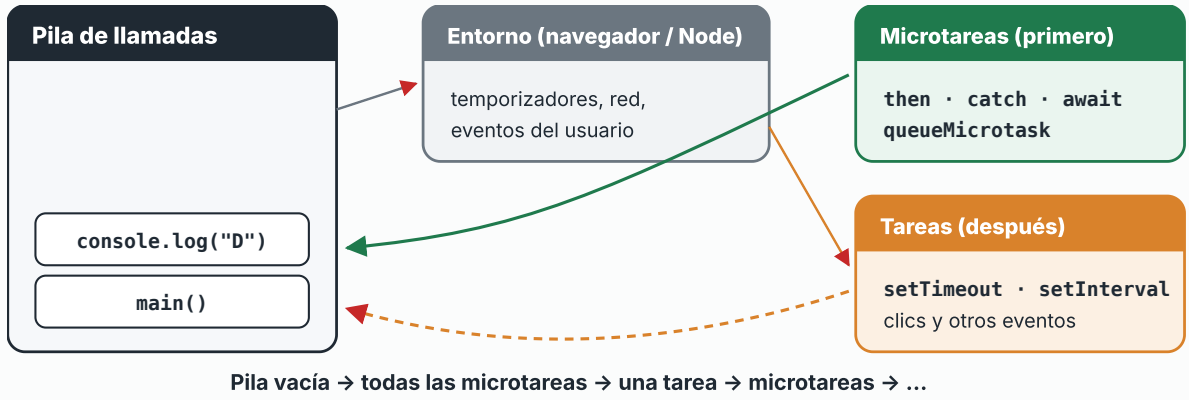


Figura 3.1 Lo sincrónico, después las microtareas y después las tareas.

Leer el orden de ejecución

1. Primero se ejecuta todo el código **sincrónico**, de arriba abajo.
2. Después, las **microtareas** (promesas).
3. Después, las **tareas** (temporizadores), por orden de vencimiento; y después de cada una, otra vez sus microtareas.

Ejemplo:
`console.log("A"); setTimeout(() => console.log("B"), 0); Promise.resolve().then(() => console.log("C")); console.log("D");`

muestra **A, D, C, B**.

- La función que se le pasa a `new Promise(...)` (el *ejecutor*) corre **en el momento**, de forma sincrónica. Lo asincrónico son los `then`.
- Un `then` sobre una promesa **ya cumplida** igual se ejecuta después del código sincrónico.
- Una función `async` corre de forma sincrónica **hasta el primer** `await`; lo que sigue queda como microtarea.

OJO Las promesas y async/await no crean hilos: el código igual corre en el hilo principal. Un cálculo pesado bloquea la página aunque esté dentro de una función async.

Temporizadores y otras herramientas

- `setTimeout(fn, 1000)` garantiza que `fn` se ejecute **no antes** de 1 segundo, pero puede ser más tarde si la pila o la cola están ocupadas. `setTimeout(fn, 0)` no es "ya": es "apenas se pueda, después de lo que está corriendo".
- `setInterval(fn, ms)` repite `fn` hasta que se llame a `clearInterval`.
- `requestAnimationFrame(fn)` ejecuta `fn` justo antes del próximo redibujado: es lo indicado para animaciones.
- Un **Web Worker** ejecuta código en **otro hilo**, sin bloquear la página; se comunica con mensajes.

Para practicar

1. ¿Qué muestra: console.log(1); setTimeout(() ⇒ console.log(2)); Promise.resolve().then(() ⇒ console.log(3)); console.log(4);?
2. ¿Por qué setTimeout(fn, 0) no se ejecuta enseguida?
3. ¿Qué parte de una función async corre de forma sincrónica?
4. ¿Qué usarías para no congelar la página con un cálculo muy largo?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 4

4 Promesas

LO QUE VAS A APRENDER

- Conocer los estados de una promesa.
- Encadenar con then y manejar errores con catch.
- Usar finally.
- Combinar promesas con all, allSettled, race y any.

Qué es una promesa

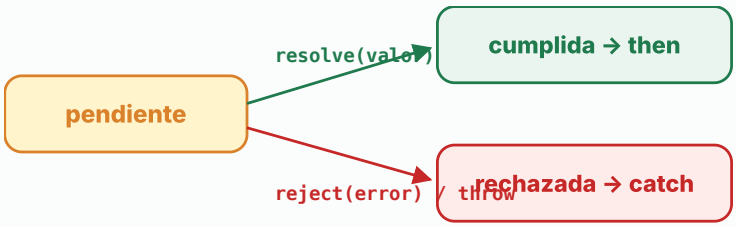
- Una **promesa** representa un valor que va a estar disponible más adelante (o un error).
- Tiene tres estados: **pendiente**, **cumplida** (*fulfilled*) o **rechazada** (*rejected*). Cuando se cumple o se rechaza queda así **para siempre** (se dice que está *asentada*): llamar otra vez a `resolve` o `reject` no hace nada.
- Se crea con `new Promise((resolve, reject) => { ... })`. Ejemplo: `const esperar = ms => new Promise(r => setTimeout(r, ms));` devuelve una promesa que se cumple a los `ms` milisegundos.
- `Promise.resolve(v)` (con un `v` que no sea promesa) y `Promise.reject(e)` crean promesas ya cumplidas o ya rechazadas.

then, catch y finally

- `p.then(fn)` ejecuta `fn` con el valor cuando la promesa se cumple, y **devuelve una promesa nueva** con lo que devuelva `fn`. Así se **encadenan**: `Promise.resolve(2).then(x => x * 3).then(x => x + 1)` termina en `7`.
- Si `fn` devuelve **otra promesa**, la cadena espera a que esa se resuelva y sigue con su valor.
- Si `fn` no tiene `return` (por ejemplo, con llaves y sin return), el siguiente `then` recibe `undefined`.
- `p.catch(fn)` atrapa un rechazo **o un error lanzado** (`throw`) en cualquier paso anterior de la cadena. Después de un `catch`, la cadena **sigue** con lo que devolvió el `catch`.
- `p.finally(fn)` se ejecuta siempre, se cumpla o se rechace. `fn` **no recibe** el valor, y la cadena sigue con el valor original (salvo que `fn` lance un error).
- Una promesa rechazada sin ningún `catch` genera un aviso de **rechazo no manejado** (*unhandled rejection*); en Node puede terminar el programa.

Combinar promesas

- `Promise.all([...])`: espera a **todas** y da un arreglo con los resultados **en el orden en que se pasaron** (no en el que terminaron). Si **una** se rechaza, se rechaza enseguida con ese error.
- `Promise.allSettled([...])`: espera a todas y da, para cada una, `{ status: "fulfilled", value }` o `{ status: "rejected", reason }`. Nunca se rechaza.
- `Promise.race([...])`: se resuelve igual que la **primera que termine**, se cumpla o se rechace. Sirve para poner un límite de tiempo.
- `Promise.any([...])`: se cumple con la **primera que se cumpla**, ignorando las rechazadas; solo se rechaza si fallan todas.



`Promise.all`	todas se cumplen → arreglo en orden	una se rechaza → se rechaza
`Promise.allSettled`	espera a todas	nunca se rechaza: status de cada una
`Promise.race`	la primera que termine	aunque sea un rechazo
`Promise.any`	la primera que se cumpla	se rechaza solo si fallan todas

Figura 4.1 Estados de una promesa y combinadores.

Para practicar

1. ¿Qué pasa si se llama a resolve y después a reject en la misma promesa?
2. ¿Qué recibe el then que sigue a un catch?
3. ¿Qué diferencia hay entre Promise.all y Promise.allSettled?
4. Escribí una función esperar(ms) que devuelva una promesa.

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 5

5 async y await

LO QUE VAS A APRENDER

- Saber qué devuelve una función async.
- Manejar errores con try/catch.
- Usar fetch correctamente (ok, json, cancelación).
- Elegir entre ejecución en serie y en paralelo.

Funciones async

- Una función `async` **siempre devuelve una promesa**: `async function f() { return 5; }` hace que `f()` sea una promesa que se cumple con `5`.
- Si dentro se lanza un error (`throw`), la promesa que devuelve queda **rechazada**.
- `await` espera a que una promesa se resuelva y da su valor. Si se usa con un valor que no es promesa (`await 7`), lo devuelve igual (pero igual cede el turno).
- `await` **pausa solo esa función**: el resto del programa sigue funcionando. No bloquea el hilo.
- `await` solo se puede usar dentro de funciones `async` o en el **nivel superior de un módulo**. En una función común da error de sintaxis.

Errores

- Con `await` se usan `try` / `catch` normales: si la promesa se rechaza, el error salta al `catch`.
- Ojo: `return promesa` dentro de un `try` **sin** `await` no atrapa el rechazo: la promesa se devuelve tal cual, sin lanzar nada dentro del `try`. Para atraparlo, `return await promesa`.
- `fetch` **no rechaza** la promesa por errores HTTP como 404 o 500: solo por fallas de red (o si se cancela). Hay que revisar `respuesta.ok` (o `respuesta.status`).
- `respuesta.json()` también devuelve una **promesa**: hay que esperarla (`await respuesta.json()`).
- Para **cancelar** un `fetch` en curso se usa un `AbortController` y se pasa su `signal`.

TIP Patrón seguro: `const r = await fetch(url); if (!r.ok) throw new Error(r.status); const datos = await r.json();` dentro de un `try`.

En serie o en paralelo

- `await esperar(1000); await esperar(1000);` tarda unos **2 segundos**: la segunda espera empieza cuando termina la primera.
- `await Promise.all([esperar(1000), esperar(1000)]);` tarda **1 segundo**: las dos corren a la vez.
- `const datos = await Promise.all(ids.map(id => buscar(id)));` busca todo en paralelo y da los resultados en el orden de `ids`.
- `for (const u of urls) { await fetch(u); }` hace los pedidos **uno después del otro**. Sirve cuando el orden importa o para no saturar un servidor.
- `forEach` **no espera** funciones `async`: ignora las promesas que devuelven. Para esperar en orden se usa `for...of`; para paralelo, `Promise.all` con `map`.

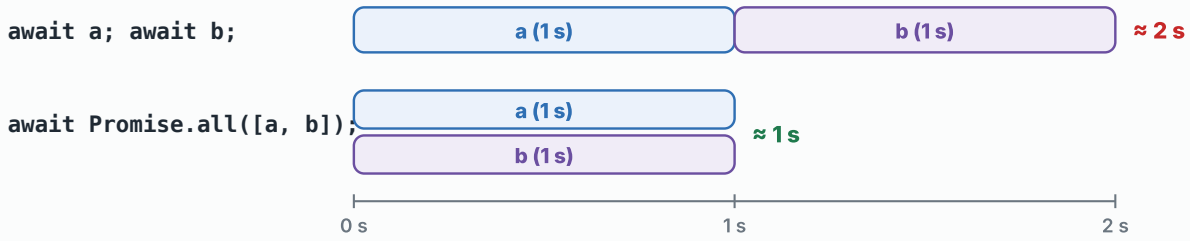


Figura 5.1 Dos esperas de 1 segundo, en serie y en paralelo.

Para practicar

1. ¿Qué devuelve async function f() { return 5; }?
2. ¿Por qué fetch no avisa con un error si la página no existe (404)?
3. Reescribí con Promise.all: for (const id of ids) { res.push(await buscar(id)); }
4. ¿Por qué forEach no sirve para esperar funciones async?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

PARA CONTROLAR

Respuestas de los ejercicios

Tema 1

1. Una función que recuerda las variables del alcance donde se creó. Por ejemplo, un contador o una función que se ejecuta una sola vez.
2. 1, 2 y 1: cada contador tiene su propio c.
3. Porque var crea una sola i para todo el bucle, y los setTimeout se ejecutan cuando el bucle ya terminó (i vale 3).
4. Con un closure: una función que declara la variable y devuelve funciones que la usan.

Tema 2

1. En o.metodo(), el objeto o. En f() en modo estricto, undefined.
2. call y apply ejecutan la función con el this indicado (apply recibe los argumentos en un arreglo); bind devuelve una función nueva sin ejecutarla.
3. Porque setTimeout llama a la función sola, sin el objeto. Soluciones: o.saludar.bind(o) o () ⇒ o.saludar().
4. Cuando dentro del método hay un callback (map, setTimeout) que necesita el this del método.

Tema 3

1. 1, 4, 3, 2.
2. Porque va a la cola de tareas: se ejecuta después del código sincrónico y de las microtareas.
3. Todo lo que está antes del primer await.
4. Un Web Worker (o partir el trabajo en pedazos).

Tema 4

1. Nada: la promesa ya quedó cumplida y el reject se ignora.
2. Lo que devolvió el catch (si no devuelve nada, undefined).
3. all se rechaza si una falla; allSettled espera a todas y nunca se rechaza.
4. const esperar = ms ⇒ new Promise(r ⇒ setTimeout(r, ms));

Tema 5

1. Una promesa que se cumple con 5.
2. Porque fetch solo se rechaza por fallas de red; hay que revisar respuesta.ok.
3. const res = await Promise.all(ids.map(id ⇒ buscar(id)));
4. Porque forEach no espera las promesas que devuelve la función.



Validá lo que aprendiste

Rendí la evaluación **JavaScript: asincronía y closures** en **testing-us.com**. Es gratis: en cada intento te tocan 10 preguntas al azar, dos de cada tema de este cuadernillo. Tenés 20 minutos y aprobás con 70%.

Si aprobás, recibís un certificado con código de verificación que podés sumar a tu currículum y compartir con quien te contrate.

MIS NOTAS

© 2026 Testing-US. Material de estudio propio. Se permite imprimirlo y compartirlo sin modificaciones y sin fines comerciales.

Referencias: especificación ECMAScript (Ecma International, ECMA-262), estándar HTML de WHATWG (bucle de eventos) y documentación de JavaScript de MDN Web Docs (developer.mozilla.org). Los textos y dibujos de este cuadernillo son propios; ninguna de estas fuentes respalda este material. JavaScript es una marca registrada de Oracle Corporation en los Estados Unidos y otros países; se menciona solo para identificar el lenguaje.

El certificado de Testing-US acredita la aprobación de una evaluación de conocimientos. No es un título oficial ni habilita para el ejercicio profesional.