

JavaScript

```
const fetchData = async (id) => {  
  const fetchData = async (id) => {  
    const res = await API.get(</users({id}>);  
    return res.data.map(user => ({  
      const res = await API.get(</users({id}>);  
      ..user, isActive: true  
    })));  
  }  
}
```



Curso Completo de JavaScript

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

lucasegaacademy.com

Curso Completo de JavaScript

Autor: Esteban Gutiérrez
lucasegaacademy.com

*Guía completa con explicaciones extensas por unidad,
prácticas y examen final.*

ÍNDICE

PRÓLOGO	4
UNIDAD 1: Primeros pasos con JavaScript	7
UNIDAD 2: Datos y decisiones	42
UNIDAD 3: Comparaciones y lógica	79
UNIDAD 4: Repetición y funciones	114
UNIDAD 5: Funciones modernas y listas	154
UNIDAD 6: Recorrer datos y objetos	184
UNIDAD 7: Textos dinámicos y el DOM	218
UNIDAD 8: Interactividad en la página	255
UNIDAD 9: Datos del servidor	290
UNIDAD 10: Código moderno y tu primer proyecto	328
EXAMEN FINAL	370

PRÓLOGO

Bienvenido al Curso Completo de JavaScript de Lucas EGA Academy.

Si estás leyendo esto, es porque ya tomaste una decisión importante: aprender el lenguaje que le da vida a la web. HTML es el esqueleto de una página, CSS es su maquillaje, pero JavaScript es su cerebro. Sin JavaScript, los botones no responden, los menús no se abren, los formularios no validan nada y las páginas son carteles estáticos. Con JavaScript, todo eso cambia: tus páginas escuchan, reaccionan, calculan, piden datos a servidores y se actualizan solas. Esa es la diferencia entre una página que se ve y una página que vive.

Este libro nació del curso en video de 30 clases de lucasegaacademy.com, pero no es una simple transcripción. Aquí cada tema se explica a fondo, con calma, con ejemplos que puedes copiar y correr, con ejercicios para que practiques de verdad y con un examen final para que compruebes lo que aprendiste. Si el video es la clase en el pizarrón, este libro es sentarte después con el cuaderno y entenderlo todo, sin prisa.

Cómo está organizado este libro

El libro está dividido en 10 unidades, y cada unidad agrupa 3 clases del curso original. Cada unidad sigue siempre la misma estructura, para que sepas qué esperar:

Primero viene una introducción extensa donde te cuento qué vas a aprender en la unidad, por qué importa y cómo se conecta con lo que ya viste. Después, la explicación detallada: el corazón de la unidad, donde cada concepto se desmenuza con analogías, ejemplos de la vida real, errores comunes que cometen todos los principiantes y buenas prácticas que te van a ahorrar dolores de cabeza. Luego vienen los ejemplos paso a paso, código comentado que puedes copiar, pegar y modificar. Después, la práctica: ejercicios con su solución explicada paso a paso, para que no te quedes con la duda. Y al final, un resumen de la unidad con los puntos clave.

Al cerrar el libro encontrarás el examen final: 50 preguntas de opción múltiple, verdadero o falso y preguntas abiertas con código, más todas las respuestas explicadas. Te recomiendo que lo hagas de verdad, con lápiz y papel o en tu editor, antes de ver las respuestas. El examen no es un castigo: es la forma más rápida de descubrir qué partes dominas y cuáles necesitas repasar.

Cómo estudiar este libro para que funcione

Te voy a ser directo: leer sin practicar no sirve. JavaScript es un lenguaje que se aprende escribiendo código, rompiéndolo y arreglándolo. Así que aquí van las reglas del juego:

Uno: ten tu editor abierto mientras lees. Cada vez que veas un bloque de código, cópialo, córralo, cámbiale cosas y observa qué pasa. La curiosidad es tu mejor maestra. Si el libro dice que un ciclo se repite cinco veces, cámbialo a diez y compruébalo. Si dice que un operador hace algo, pruébalo con valores raros.

Dos: haz los ejercicios antes de ver las soluciones. Es tentador saltar directo a la respuesta, pero el aprendizaje real ocurre en el intento. Date al menos quince minutos por ejercicio antes de rendirte. Si te atorras, relee la explicación de la unidad. Si sigues atorado, mira la solución, pero luego ciérrala y resuélvelo tú solo desde cero.

Tres: no te brinques unidades. Cada unidad se construye sobre la anterior. Las funciones de la unidad 4 usan los condicionales de la unidad 2. El DOM de la unidad 7 necesita las funciones y los objetos de las unidades anteriores. Si algo no te queda claro, regresa y repásalo antes de seguir.

Cuatro: acepta que te vas a equivocar. Todos los programadores, hasta los que llevan veinte años, escriben código con errores todos los días. La diferencia entre un principiante y un profesional no es que el profesional no se equivoque: es que el profesional sabe leer el error, buscar la causa y corregirlo. Este libro te enseña eso también.

Cinco: termina el libro y luego construye algo tuyo. El proyecto del contador de la unidad 10 es solo el inicio. Una lista de tareas, un juego de adivinanzas, una calculadora, una página que consuma datos de una API pública. Los proyectos propios son lo que convierte el conocimiento en habilidad.

Lo que necesitas para empezar

No necesitas nada caro ni complicado. Solo tres cosas: una computadora (cualquiera que corra un navegador moderno), un navegador (Chrome, Edge o Firefox, el que prefieras) y un editor de texto (Visual Studio Code es gratis y es el estándar). Todo el código de este libro corre en el navegador, así que no tienes que instalar nada más. Si sabes abrir un archivo HTML en tu navegador, ya tienes todo lo necesario.

Una última cosa antes de empezar

Aprender a programar es como aprender a tocar un instrumento: al principio todo se siente torpe y lento, y es normal. La unidad 1 te va a parecer fácil, la unidad 5 te va a retar, y en algún punto vas a pensar que esto no es para ti. Ese momento le llega a todo el mundo, y la única diferencia entre los que aprenden y los que no es que los primeros siguen adelante un día más. Confía en el proceso, practica todos los días aunque sea media hora, y cuando llegues al examen final vas a mirar atrás y no vas a creer lo lejos que llegaste.

Nos vemos en la unidad 1. ¡A programar!

UNIDAD 1: Primeros pasos con JavaScript

Introducción

Bienvenido a JavaScript. Si este es tu primer lenguaje de programación, acabas de tomar una de las mejores decisiones posibles para entrar al mundo del desarrollo. Y si ya programas en otro lenguaje, estás a punto de conocer el que más terreno ocupa en el planeta: el único lenguaje que corre de forma nativa en todos los navegadores del mundo, el lenguaje que hace que la web se mueva, que los botones respondan, que los formularios validen datos antes de enviarse, que los mapas se deslicen bajo tus dedos y que las aplicaciones que usas todos los días en tu teléfono funcionen.

En esta primera unidad vamos a poner los cimientos. No vamos a construir nada espectacular todavía, pero lo que aprendas aquí va a sostener todo lo demás. Piénsalo como aprender a sostener bien el martillo antes de intentar clavar: puede sonar aburrido, pero la gente que se salta esta parte es la misma que meses después sigue cometiendo los mismos errores básicos y no entiende por qué su código no funciona.

Empecemos por la pregunta más básica: ¿qué es JavaScript? JavaScript es un lenguaje de programación de alto nivel, interpretado, que originalmente nació para darle vida a las páginas web. Hoy en día hace muchísimo más que eso: con JavaScript se construyen servidores completos, aplicaciones de escritorio, aplicaciones móviles y hasta se programan robots y dispositivos. Pero su hogar original, el lugar donde nació y donde sigue siendo rey absoluto, es el navegador.

Aquí hay un punto que vale la pena aclarar desde ya, porque confunde a todo el mundo al principio: JavaScript no tiene casi nada que ver con Java. Los nombres se parecen por una decisión de mercadotecnia de los años noventa, cuando Java estaba de moda y a alguien se le ocurrió que pegarle el nombre Java a este nuevo lenguaje le daría publicidad gratis. Fue una jugada comercial y funcionó, pero tres décadas después seguimos pagando el precio de esa confusión. Java y JavaScript son lenguajes distintos, con filosofías distintas, sintaxis distinta y propósitos distintos. Si alguien te pregunta, ya sabes la respuesta: el parecido es solo el nombre.

¿Para qué sirve JavaScript en la práctica? Imagina una página web como un cuerpo humano. HTML es el esqueleto: define la estructura, qué va dónde, los huesos de la página. CSS es la ropa y la apariencia: los colores, las formas, cómo se ve todo. JavaScript es el sistema

nervioso y los músculos: es lo que hace que las cosas se muevan, reaccionen y respondan. Sin JavaScript, la web sería un cartel estático. Con JavaScript, la web respira.

Cuando haces clic en "me gusta" y el corazón se pone rojo sin que la página se recargue, eso es JavaScript. Cuando escribes en un buscador y te aparecen sugerencias mientras escribes, eso es JavaScript. Cuando un carrito de compras suma los precios automáticamente, cuando un formulario te dice "ese correo no es válido" antes de enviarlo, cuando ves un carrusel de imágenes que se mueve solo, cuando juegas algo en el navegador: todo eso es JavaScript trabajando por debajo.

Y lo más importante para ti como estudiante: JavaScript es el lenguaje con la barrera de entrada más baja que existe. No necesitas instalar nada especial para empezar. Ya tienes todo lo que necesitas: un navegador y un editor de texto. Eso significa que puedes escribir tu primera línea de código funcional hoy, en este momento, sin configuraciones complicadas, sin compiladores, sin entornos raros. Esa inmediatez es oro para aprender: escribes algo, lo pruebas, ves el resultado al instante, corriges y sigues. Ese ciclo rápido de prueba y error es exactamente como se aprende a programar.

En esta unidad vamos a cubrir tres grandes temas que corresponden a las primeras tres clases del curso. Primero, qué es JavaScript y para qué sirve: su historia breve, dónde vive, qué puede y qué no puede hacer. Segundo, cómo incluir JavaScript en una página HTML: la etiqueta script, dónde ponerla, por qué el orden importa, cómo usar archivos externos y qué significa ese atributo defer que vas a ver en todos los proyectos profesionales. Tercero, las variables: cómo guardar información en la memoria usando let y const, por qué var quedó en el pasado, cómo nombrar tus variables para que tu código se entienda solo y qué es eso del scope o alcance que determina dónde vive cada variable y quién puede verla.

Cada uno de estos temas parece pequeño por separado, pero juntos forman la base sobre la que se construye todo lo demás. Las variables, por ejemplo, son el concepto más usado en toda la programación: absolutamente todo programa que escribas en tu vida va a guardar datos en variables. Entenderlas bien desde el día uno te ahorra cientos de horas de confusión después.

Un consejo antes de empezar: no te limites a leer. La programación no se aprende leyendo, se aprende haciendo. Cada vez que veas un ejemplo de código en este libro, ábrelo en tu editor, ejecútalo, rómpelo a propósito y arréglalo. Cambia los valores, cambia los nombres, quita una línea y observa qué se rompe. Esa curiosidad de "¿qué pasa si...?" es la habilidad más valiosa que puedes desarrollar como programador. Los mejores desarrolladores que conozco no son los que memorizaron más sintaxis, son los que no tienen miedo de experimentar.

Otra cosa: vas a cometer errores. Muchos. Todos los días. Eso no significa que seas malo programando, significa que estás programando. El error es el maestro más honesto que existe: te dice exactamente qué está mal, en qué línea y muchas veces por qué. Aprende a leer los mensajes de error en lugar de tenerles miedo y habrás dado un salto gigante.

Vamos a ello. Tu viaje con JavaScript empieza aquí, con los primeros pasos. Camínalos con calma, entiéndelos bien, y cuando llegues a la unidad 30 estarás construyendo proyectos completos sin darte cuenta de lo lejos que llegaste.

Explicación detallada

Qué es JavaScript y dónde vive

Para entender JavaScript de verdad, ayuda conocer un poco de su historia, porque explica muchas de las cosas raras que vas a encontrar en el lenguaje. JavaScript nació en 1995, creado por Brendan Eich en Netscape en aproximadamente diez días. Sí, leíste bien: diez días. El lenguaje que hoy mueve buena parte de la economía digital mundial se escribió en poco más de una semana, bajo presión, para un navegador que ya ni existe. Esa historia de origen explica por qué JavaScript tiene algunas decisiones de diseño extrañas y algunos comportamientos que parecen caprichos: fue un lenguaje hecho a toda prisa que después tuvo que crecer sin romper la web.

Y crecer, creció. De ser un juguetito para hacer que los botones cambiaran de color, JavaScript se convirtió en un lenguaje serio, estandarizado por una organización llamada ECMA bajo el nombre oficial de ECMAScript. Cuando escuches hablar de ES6 o ECMAScript 2015, se refieren a una versión del estándar que trajo cambios enormes al lenguaje: ahí nacieron `let`, `const`, las funciones flecha, las clases, los módulos y muchas cosas más. Todo el JavaScript moderno que vas a aprender en este libro es JavaScript post-ES6. Es importante saberlo porque en internet hay tutoriales viejísimos que enseñan JavaScript de 2005 como si fuera actual, y te van a confundir. Si un tutorial te enseña `var` como la forma normal de declarar variables, cierra esa pestaña: es contenido desactualizado.

¿Dónde vive JavaScript? Su casa original y principal es el navegador. Cada navegador moderno (Chrome, Firefox, Safari, Edge) trae un motor de JavaScript integrado: V8 en Chrome y Edge, SpiderMonkey en Firefox, JavaScriptCore en Safari. Estos motores son programas increíblemente optimizados que leen tu código JavaScript y lo ejecutan a velocidades absurdas. Cuando abres una página web y esta hace algo interactivo, es el motor de JavaScript de tu navegador ejecutando el código que el desarrollador escribió.

Pero JavaScript escapó del navegador hace años. En 2009 nació Node.js, que permite ejecutar JavaScript fuera del navegador, en servidores y en tu computadora. Gracias a Node.js, hoy puedes construir el backend completo de una aplicación (la parte del servidor) usando el mismo lenguaje que usas en el frontend (la parte visual). Esto es una ventaja enorme: aprendes un solo lenguaje y puedes trabajar en ambos lados. Por eso tantos desarrolladores eligen JavaScript como primer lenguaje: te abre las puertas del desarrollo web completo.

También existen frameworks y herramientas construidas sobre JavaScript que dominan la industria: React, Vue, Angular para interfaces; Express para servidores; React Native para aplicaciones móviles; Electron para aplicaciones de escritorio como Visual Studio Code o Discord. Todo eso es JavaScript por debajo. Pero ojo: todos esos frameworks asumen que ya dominas el JavaScript base, el que vamos a aprender en este libro. Mucha gente quiere saltar directo a React sin saber JavaScript bien, y eso es como querer correr un maratón sin saber caminar. Primero los fundamentos, después las herramientas.

¿Qué puede hacer JavaScript en el navegador? Prácticamente todo lo que ves interactivo: modificar el contenido de la página en tiempo real, responder a clics, movimientos del mouse y teclas presionadas, validar formularios, hacer animaciones, comunicarse con servidores para traer datos sin recargar la página, guardar información en el navegador del usuario, dibujar gráficos, reproducir audio y video de forma programada, acceder a la geolocalización, y mucho más. En unidades posteriores vamos a tocar casi todos estos temas.

¿Qué no puede hacer JavaScript en el navegador? Por seguridad, el JavaScript que corre en una página web vive en una cajita de arena, un entorno restringido. No puede leer archivos arbitrarios de tu computadora, no puede acceder a otros programas, no puede hacer lo que quiera con tu sistema. Esto es a propósito: imagina si cualquier página web que visitaras pudiera borrar tus archivos. El navegador pone límites estrictos, y eso está bien. Cuando necesites hacer cosas más potentes, usas Node.js fuera del navegador, donde tú controlas los permisos.

Cómo incluir JavaScript en una página HTML

Ahora viene la parte práctica: ¿cómo le dices al navegador "oye, aquí hay código JavaScript, ejecútalo"? La respuesta es la etiqueta script. Es el puente entre tu HTML y tu JavaScript, y entenderla bien te evita una categoría entera de errores de principiante.

La forma más directa es escribir el JavaScript directamente dentro del HTML, entre una etiqueta de apertura script y una de cierre /script:

```
<script>
  alert("Hola desde JavaScript");
</script>
```

Esto se llama script en línea o interno. Funciona, y para experimentos rápidos y ejemplos pequeños está bien. Pero en proyectos reales casi nunca vas a hacer esto, por razones que veremos en un momento.

¿Dónde pones la etiqueta script dentro del HTML? Tienes dos opciones comunes: en el head o al final del body, justo antes de la etiqueta de cierre /body. Y esta decisión sí importa, por cómo el navegador lee tu página.

El navegador lee el HTML de arriba hacia abajo, en orden, como tú lees un libro. Cuando encuentra una etiqueta script, por defecto hace una pausa: deja de leer el HTML, ejecuta el JavaScript, y cuando termina sigue con el resto. Esto se llama bloqueo: el script bloquea la lectura del resto de la página mientras se ejecuta.

Entonces, si pones tu script en el head y ese script intenta modificar un elemento que está más abajo en el body, va a fallar, porque cuando el script se ejecuta, ese elemento todavía no existe: el navegador aún no ha llegado a leerlo. Es como intentar saludar a alguien que todavía no ha entrado a la fiesta. Este es uno de los errores más comunes de los principiantes y causa mensajes de error del tipo "cannot read properties of null", que veremos más adelante.

La solución clásica y más simple: pon tus scripts al final del body, justo antes de /body. Así te aseguras de que todo el HTML ya fue leído cuando tu JavaScript se ejecuta. Todos los elementos existen, todo está en su lugar, y tu código puede trabajar con la página completa sin problemas.

```

<!DOCTYPE html>
<html>
<head>
  <title>Mi primera página con JS</title>
</head>
<body>
  <h1>Hola mundo</h1>
  <p id="mensaje">Este texto va a cambiar.</p>

  <script>
    document.getElementById("mensaje").textContent = "¡JavaScript sí funciona!";
  </script>
</body>
</html>

```

En este ejemplo, cuando el navegador llega al script, el párrafo con id="mensaje" ya existe, así que el código puede encontrarlo y cambiar su texto sin problema. Si ese mismo script estuviera en el head, fallaría.

Ahora, la segunda forma de incluir JavaScript, y la que vas a usar en proyectos reales: el archivo externo. En lugar de escribir el código dentro del HTML, lo escribes en un archivo separado con extensión.js, por ejemplo app.js, y lo conectas con el atributo src:

```

<script src="app.js"></script>

```

El archivo app.js contiene solo JavaScript, sin etiquetas script alrededor:

```

// app.js
document.getElementById("mensaje").textContent = "¡JavaScript sí funciona!";

```

¿Por qué es mejor el archivo externo? Por varias razones de peso. Primera: separación de responsabilidades. El HTML se encarga de la estructura, el CSS de la apariencia y el JavaScript del comportamiento, cada uno en su archivo. Esto hace tu proyecto más ordenado y más fácil de mantener. Segunda: reutilización. Un mismo archivo.js lo puedes conectar a varias páginas HTML distintas sin copiar y pegar código. Tercera: el navegador puede guardar en caché el archivo.js, así que si el usuario visita varias páginas de tu sitio, el JavaScript se descarga una sola vez y las siguientes páginas cargan más rápido. Cuarta: los editores de código te dan mejor ayuda (autocompletado, resaltado, detección de errores) cuando el JavaScript está en su propio archivo.

Un error clásico de principiante: poner código dentro de una etiqueta script que ya tiene src. No funciona. Cuando una etiqueta script tiene atributo src, el navegador ignora por completo cualquier código escrito entre la apertura y el cierre. Es una u otra: o src, o código interno, nunca ambos.

```
<!-- ESTO ESTÁ MAL: el código interno se ignora -->
<script src="app.js">
  alert("Esto nunca se va a ejecutar");
</script>
```

Otro detalle: el atributo src puede apuntar a una ruta relativa (app.js, js/app.js, ../js/app.js) o a una URL completa (https://ejemplo.com/libreria.js). Las rutas relativas se resuelven respecto a la ubicación del archivo HTML, igual que con las imágenes. Si tu estructura de proyecto tiene el HTML en la raíz y el JavaScript en una carpeta js, escribirías src="js/app.js".

Ahora hablemos de defer, porque lo vas a ver en todos los proyectos profesionales y conviene entenderlo desde ahora. Recuerda el problema: los scripts bloquean la lectura del HTML. Con archivos externos grandes, esto puede hacer que la página se vea en blanco un rato mientras se descarga y ejecuta el JavaScript. El atributo defer le dice al navegador: "descarga este archivo en segundo plano mientras sigues leyendo el HTML, y ejecútalo hasta que termines de leer toda la página".

```
<script src="app.js" defer></script>
```

Con defer, puedes poner tu script en el head sin miedo, porque aunque el navegador lo descargue temprano, no lo va a ejecutar hasta que el documento esté completamente leído. Es lo mejor de los dos mundos: el archivo se empieza a descargar pronto (rápido) y se ejecuta tarde (seguro). Por eso es el estándar en proyectos modernos.

Hay un primo de defer llamado async, que también descarga en segundo plano pero ejecuta el script en cuanto termina de descargarse, sin esperar a que el HTML esté completo. Suena parecido pero el matiz es importante: con async no tienes garantía del orden de ejecución si tienes varios scripts, y el script puede ejecutarse antes de que el HTML esté listo. async sirve para scripts independientes que no tocan el HTML ni dependen de otros scripts, como herramientas de analítica o publicidad. Para tu código normal, que sí trabaja con la página, defer es la opción correcta. Si solo vas a recordar una cosa: usa defer para tus archivos, ponlos en el head o al final del body, y olvídate del problema del orden.

Resumiendo las reglas prácticas: para aprender y hacer pruebas rápidas, el script interno al final del body está perfecto. Para cualquier proyecto que vaya a crecer, archivo externo con defer. Nunca mezcles src con código interno. Y si tu JavaScript "no funciona" y no sabes por qué, lo primero que debes revisar es dónde está tu etiqueta script y si el HTML que intentas manipular ya existe cuando el código se ejecuta.

Variables: let y const

Llegamos al concepto más importante de esta unidad y uno de los más importantes de toda la programación: las variables. Una variable es un espacio en la memoria donde guardas un dato para usarlo después. Es como una cajita con una etiqueta: le pones nombre a la cajita, metes algo dentro, y cuando lo necesitas, lo pides por su nombre.

Piensa en una receta de cocina. En lugar de escribir "250 gramos de harina" cincuenta veces en la receta, defines al inicio: harina = 250 gramos. Y cada vez que la receta dice "agrega la harina", sabes de qué habla. Las variables hacen exactamente eso en el código: le dan nombre a los datos para que puedas referirte a ellos, cambiarlos y usarlos en operaciones.

En JavaScript moderno declaras variables con let o const. Declarar significa crear la variable, presentarla formalmente ante el lenguaje:

```
let nombre = "Esteban";  
const PI = 3.1416;
```

La palabra let o const va primero, luego el nombre que tú eliges, luego un signo igual y el valor inicial. El signo igual aquí no significa "es igual a" en el sentido matemático, significa "guarda este valor en esta variable". Los programadores le dicen asignación: estás asignando el valor "Esteban" a la variable nombre.

¿Cuál es la diferencia entre let y const? Es simple y es una de las decisiones más importantes que tomarás línea por línea: let declara una variable cuyo valor puede cambiar después; const declara una constante cuyo valor no puede ser reasignado. Mira:

```
let edad = 25;  
edad = 26; // Perfecto, let permite cambiar el valor  
  
const pais = "México";  
pais = "España"; // ERROR: no puedes reasignar una const
```


La segunda línea del ejemplo con `const` lanza un error del tipo "Assignment to constant variable". El lenguaje te protege de cambiar algo que declaraste como inmutable. Y aquí viene la buena práctica que te va a acompañar toda tu carrera: usa `const` por defecto, y `let` solo cuando sepas que el valor va a cambiar. ¿Por qué? Porque la mayoría de los valores en un programa no cambian, y declarar todo como `const` hace tu código más predecible y más fácil de razonar: cuando ves `const` sabes con certeza que ese valor es el mismo de principio a fin. Además te protege de errores tontos, como reasignar por accidente una variable que no debías tocar.

Hay un matiz importante con `const` que confunde mucho al principio: `const` impide la reasignación, pero no congela el contenido si el valor es un objeto o un array. Es decir, no puedes hacer que la variable apunte a otra cosa, pero sí puedes modificar lo que hay dentro. Lo veremos a fondo en la unidad de objetos y arrays, pero que no te sorprenda desde ahora:

```
const colores = ["rojo", "azul"];
colores.push("verde"); // Permitido: no reasignas, modificas el contenido
colores = ["amarillo"]; // ERROR: esto sí es reasignación
```

Piensa en `const` como un candado en la etiqueta de la cajita, no en el contenido: la etiqueta siempre apunta a la misma cajita, pero puedes meter y sacar cosas de la cajita si esta es un objeto o un array.

Otro detalle: tanto `let` como `const` exigen que declares la variable antes de usarla, y `const` además exige que le des un valor inicial en la misma declaración. Esto tiene sentido: si es constante y no la puedes reasignar después, ¿de qué serviría declararla vacía?

```
let puntos; // Válido: vale undefined por ahora
puntos = 10; // Válido después

const impuesto; // ERROR: a const hay que darle valor al declararla
```

El fantasma de `var`: por qué ya no se usa

Si buscas tutoriales viejos o lees código antiguo, vas a ver `var` por todas partes. `var` era la única forma de declarar variables en JavaScript antes de 2015. Hoy en día no debes usarla en código nuevo, y necesitas saber por qué para entender el código viejo que inevitablemente te vas a encontrar.

El problema de var es su scope, su alcance. Sin entrar todavía en todos los detalles del scope (que vienen en la siguiente sección), la idea es esta: let y const respetan los bloques, es decir, una variable declarada con let dentro de un par de llaves {} solo existe dentro de esas llaves. var no: var ignora los bloques y su alcance es toda la función donde se declara, o global si está fuera de funciones. Esto causa comportamientos sorprendentes y errores difíciles de rastrear.

Mira este ejemplo clásico:

```
if (true) {  
  var x = 10;  
  let y = 20;  
}  
console.log(x); // 10: ¡var se escapó del bloque!  
console.log(y); // ERROR: y no existe aquí fuera
```

La variable x declarada con var dentro del if sigue viva fuera del if, como si el bloque no existiera. La variable y declarada con let muere al cerrarse la llave, que es el comportamiento sensato y esperado. Con var es facilísimo pisar variables sin querer, declarar dos veces la misma variable sin que nadie te avise, y terminar con bugs que te hacen arrancarte el pelo.

Además var permite redeclarar la misma variable en el mismo alcance sin protestar:

```
var nombre = "Ana";  
var nombre = "Luis"; // var no se queja, silenciosamente pisa el valor
```

Con let, redeclarar en el mismo alcance es un error inmediato y claro. Ese error inmediato es un regalo: te avisa en el momento en que cometes el descuido, en lugar de dejar que el bug se esconda y aparezca tres horas después en otra parte del programa.

En resumen: var es historia. Existe solo por compatibilidad con el pasado, porque hay millones de páginas viejas que lo usan y la web no puede romperlas. Tú, en tu código nuevo, usa let y const siempre. Cuando veas var en código ajeno, tradúcelo mentalmente a let pero con cuidado, porque su alcance es más amplio y traicionero.

Cómo nombrar tus variables

Elegir buenos nombres para tus variables es una de las habilidades más subestimadas de la programación. Un buen nombre hace que el código se lea solo; un mal nombre convierte el código en un acertijo. Y como vas a leer tu propio código muchas más veces de las que lo escribes, invertir dos segundos en un buen nombre te ahorra minutos de confusión después.

Primero las reglas duras, las que el lenguaje te impone: el nombre puede contener letras, números, guion bajo `_` y signo de dólar `$`. No puede empezar con un número. No puede contener espacios ni guiones medios. No puede ser una palabra reservada del lenguaje como `let`, `const`, `function` o `return`. JavaScript distingue mayúsculas de minúsculas: `nombre`, `Nombre` y `NOMBRE` son tres variables distintas.

```
// Válidos
let edad;
let _privado;
let $precio;
let nombre2;

// Inválidos
let 2nombre; // no puede empezar con número
let mi-nombre; // el guion se interpreta como resta
let mi nombre; // los espacios no están permitidos
let let; // palabra reservada
```

Ahora las convenciones, que no son obligatorias pero que todo el mundo sigue: en JavaScript se usa `camelCase` para variables y funciones: la primera palabra en minúscula y cada palabra siguiente con mayúscula inicial, todo pegado. `nombreDeUsuario`, `totalDeCompra`, `estaEsMiVariable`. Las constantes que representan valores fijos universales se suelen escribir en `MAYÚSCULAS_CON_GUIONES_BAJOS`: `PI`, `IVA`, `DIAS_POR_SEMANA`. Los nombres de clases (que verás mucho después) usan `PascalCase`: `MiClase`.

Pero más importante que la convención es el significado. Compara:

```
// Código malo: ¿qué es d? ¿qué es x?  
let d = 30;  
let x = d * 0.16;  
console.log(x);  
  
// Código bueno: se lee solo  
let diasTrabajados = 30;  
let sueldoPorDia = 500;  
let sueldoTotal = diasTrabajados * sueldoPorDia;  
console.log(sueldoTotal);
```

El segundo ejemplo no necesita comentarios porque los nombres ya cuentan la historia. Esa es la meta: código que se explica solo. Evita las abreviaturas crípticas (usr, tmp, aux, data2) salvo que sean convenciones muy establecidas en tu equipo. Evita los nombres de una sola letra excepto en contextos diminutos y obvios, como el contador `i` de un ciclo `for`, que es una convención universal.

Un truco que usan los buenos programadores: nombra las variables booleanas (las que guardan verdadero o falso) como preguntas: `esMayorDeEdad`, `tieneDescuento`, `estaActivo`. Cuando lees `if (esMayorDeEdad)` suena a frase natural: "si es mayor de edad". Los arrays suelen nombrarse en plural: `nombres`, `precios`, `usuarios`. Y las funciones con verbos: `calcularTotal`, `obtenerNombre`, `validarCorreo`. Ya hablaremos de funciones en su unidad, pero el hábito del buen nombre empieza aquí, con las variables.

¿Sabías que hay estudios que muestran que los programadores pasan más tiempo leyendo código que escribiéndolo? Por eso los nombres importan tanto: no escribes código para la computadora, la computadora no necesita nombres bonitos; escribes código para las personas que lo van a leer después, empezando por tu yo del futuro dentro de tres meses, que ya no se va a acordar qué significaba esa `x`.

Scope: dónde vive cada variable

El scope, o alcance, es el territorio donde una variable existe y puede ser usada. Es uno de esos conceptos que al principio suena abstracto pero que en la práctica determina si tu programa funciona o explota con un "is not defined".

Hay tres niveles principales. El scope global: las variables declaradas fuera de cualquier bloque o función viven en todas partes, todo el código puede verlas. El scope de bloque: las variables declaradas con `let` o `const` dentro de un par de llaves `{}` solo existen dentro de esas llaves. El scope de función: las variables declaradas dentro de una función solo existen dentro

de esa función (esto lo veremos a fondo en la unidad de funciones, pero la idea es la misma que el bloque).

Piensa en el scope como habitaciones de una casa con paredes de vidrio polarizado: desde adentro puedes ver hacia afuera, pero desde afuera no puedes ver hacia adentro. El código dentro de un bloque puede usar las variables globales y las de los bloques que lo contienen, pero el código de afuera no puede usar las variables declaradas dentro del bloque.

```
let ciudad = "Guadalajara"; // scope global: todos la ven

if (true) {
  let calle = "Av. Juárez"; // scope de bloque: solo dentro del if
  console.log(ciudad); // OK: desde adentro se ve afuera
}

console.log(calle); // ERROR: desde afuera no se ve adentro
```

Esta regla es tu amiga aunque al principio te frustre. Limitar dónde vive cada variable evita que partes distantes del programa se pisen entre sí. Imagina un programa grande con cientos de variables globales: cualquier función podría cambiar cualquiera, y encontrar quién rompió qué sería una pesadilla. El scope de bloque te permite reutilizar nombres cortos como `i`, nombre o total en bloques distintos sin que choquen entre ellos.

Un error típico de principiante es declarar una variable dentro de un `if` o un ciclo y tratar de usarla fuera:

```
let edad = 20;
if (edad >= 18) {
  let mensaje = "Eres mayor de edad";
}
console.log(mensaje); // ERROR: mensaje murió al cerrarse el if
```

La solución es declarar la variable en el scope donde la necesitas, o sea, fuera del bloque si la vas a usar fuera:

```
let edad = 20;
let mensaje; // declarada fuera, vive fuera
if (edad >= 18) {
  mensaje = "Eres mayor de edad";
}
console.log(mensaje); // OK
```

Fíjate en el patrón: declaras con `let` fuera (sin valor inicial, o con un valor por defecto) y asignas dentro. Es un patrón que vas a usar miles de veces.

Cuidado con crear variables globales por accidente: si asignas un valor a un nombre sin declararlo con `let`, `const` o `var`, JavaScript en modo no estricto crea una variable global silenciosamente. Es uno de los peores comportamientos heredados del lenguaje:

```
function saludar() {  
  nombre = "Pedro"; // ¡Sin let! Se crea una global por accidente  
}
```

Esa variable `nombre` ahora vive en todas partes y puede chocar con cualquier otra cosa. Por eso en código moderno se usa el modo estricto ("use strict"), que convierte este descuido en un error claro en lugar de un bug silencioso. No te preocupes demasiado por el modo estricto ahora; solo recuerda la regla de oro: declara siempre tus variables. Toda variable que uses debe nacer con `let` o `const`.

Para cerrar esta sección, una analogía que me gusta: las variables son como los contactos de tu teléfono. `let` es un contacto normal: puedes cambiarle el número cuando quieras. `const` es un contacto bloqueado contra edición: el número que guardaste es el que queda, aunque si el contacto es una empresa (un objeto) sus datos internos sí pueden actualizarse. `var` es ese contacto viejo duplicado que aparece tres veces con números distintos y nunca sabes cuál es el bueno: por eso lo jubilamos. Y el scope es tu lista de contactos compartida o privada: algunas variables las ve todo el programa, otras solo el bloque donde nacieron.

Preguntas frecuentes de la unidad

¿Puedo declarar una variable sin darle valor? Con `let` sí: vale `undefined` hasta que le asignes algo. Con `const` no: es obligatorio inicializarla, porque después ya no podrás asignarle nada.

¿`let` y `const` son más lentos que `var`? No, para nada. Los motores modernos optimizan los tres por igual. La diferencia es de semántica y seguridad, no de velocidad.

¿Debo escribir los nombres en español o en inglés? En la industria se acostumbra el inglés, porque el código se comparte con gente de todo el mundo y porque las palabras clave del lenguaje ya están en inglés. Pero para aprender, usar español está perfectamente bien y te ayuda a pensar con claridad. Lo importante es ser consistente: no mezcles `nombreDeUsuario` con `userName` en el mismo proyecto.

¿Qué pasa si intento usar una variable antes de declararla con `let`? Obtienes un `ReferenceError`. Existe un concepto llamado "zona muerta temporal" que explica por qué: entre el inicio del bloque y la línea de declaración, la variable existe pero está en una zona donde tocarla es error. Es el lenguaje protegiéndote de usar algo que aún no está listo. Con `var` pasaba algo distinto y peor: valía `undefined` sin avisar, y los bugs se escondían.

¿El punto y coma es obligatorio? Técnicamente JavaScript tiene inserción automática de punto y coma y mucho código funciona sin ellos. Pero hay casos borde donde omitirlos rompe el código de formas muy raras. La recomendación profesional es usarlos siempre al final de cada instrucción. Es un hábito barato que te evita una categoría entera de dolores de cabeza.

Ejemplos paso a paso

Vamos a bajar todo esto a tierra con ejemplos completos que puedes copiar, pegar y ejecutar. Te recomiendo abrir la consola del navegador (F12 en la mayoría de los navegadores, pestaña Consola) o un archivo HTML de prueba, y probar cada ejemplo con tus propias modificaciones. La consola es el laboratorio del programador JavaScript: escribes una línea, presionas Enter y ves el resultado al instante.

Ejemplo 1: tu primer script interno. Crea un archivo `hola.html` con este contenido y ábrelo en el navegador. Verás una ventana emergente con el saludo, y si abres la consola verás el mensaje registrado ahí también.

```

<!DOCTYPE html>
<html>
<head>
  <title>Mi primer script</title>
</head>
<body>
  <h1>Abre la consola con F12</h1>

  <script>
    // alert muestra una ventana emergente
    alert("¡Hola! Este es mi primer JavaScript");

    // console.log escribe en la consola del navegador
    // Es la herramienta número uno para depurar
    console.log("El script se ejecutó correctamente");
  </script>
</body>
</html>

```

Fíjate en los comentarios: en JavaScript, // marca un comentario de una línea, y el navegador ignora todo lo que sigue en esa línea. Los comentarios no afectan el programa; existen para los humanos. Úsalos para explicar el porqué de algo raro, no para describir lo obvio. Un comentario que dice // suma uno más uno encima de `let x = 1 + 1` no aporta nada; un comentario que explica por qué usaste cierto truco sí vale oro.

Ejemplo 2: archivo externo con defer. Esta es la estructura profesional mínima. Crea dos archivos en la misma carpeta. El HTML:

```

<!DOCTYPE html>
<html>
<head>
  <title>Proyecto con archivo externo</title>
  <!-- defer: descarga en segundo plano, ejecuta al terminar el HTML -->
  <script src="app.js" defer></script>
</head>
<body>
  <h1 id="titulo">Título original</h1>
  <p>Esta página carga su JavaScript desde un archivo externo.</p>
</body>
</html>

```

Y el archivo app.js, solo con JavaScript, sin etiquetas:


```
// app.js
// Gracias a defer, este código corre cuando el HTML ya está completo,
// así que el elemento con id="titulo" garantizado que existe.

const titulo = document.getElementById("titulo");

// Cambiamos el texto del h1 usando la variable
titulo.textContent = "Título cambiado con JavaScript";

// También podemos cambiar estilos desde JS
titulo.style.color = "darkgreen";

console.log("app.js cargado con defer: todo el HTML ya existe");
```

Ejemplo 3: jugando con let y const en la consola. Abre la consola del navegador (F12) y escribe estas líneas una por una, presionando Enter después de cada una. Observa qué responde la consola y qué errores aparecen a propósito.

```
let nombre = "Mariana";
console.log(nombre); // Mariana

nombre = "Mariana Torres"; // reasignar con let: permitido
console.log(nombre); // Mariana Torres

const ciudad = "Monterrey";
console.log(ciudad); // Monterrey

ciudad = "Tijuana"; // TypeError: Assignment to constant variable
// La consola te muestra el error en rojo. ¡Eso es bueno!
// Significa que const está haciendo su trabajo: proteger el valor.

const iva = 0.16;
let precio = 100;
let total = precio + precio * iva;
console.log(total); // 116

precio = 200; // cambiamos el precio...
total = precio + precio * iva; // ...y hay que recalcular el total
console.log(total); // 232
```

Nota algo importante en ese último fragmento: cambiar precio no actualiza total automáticamente. Las variables guardan valores, no fórmulas. total guardó el número 116, no

la operación que lo produjo. Si quieres que total se actualice, tienes que volver a ejecutar la asignación. Este malentendido es muy común al principio: la computadora no recuerda cómo calculaste algo, solo recuerda el resultado. Cada vez que necesites el valor actualizado, repite la operación o, mejor aún, usa una función (tema de una unidad posterior).

Ejemplo 4: el scope en acción. Este ejemplo demuestra visualmente dónde vive cada variable. Cópialo en un archivo o en la consola y predice el resultado de cada `console.log` antes de ejecutarlo. Acostúmbrate a predecir: es el ejercicio mental que más te hace aprender.

```
let animal = "perro"; // global: visible en todas partes

if (true) {
  let comida = "croquetas"; // vive solo dentro de este bloque
  console.log(animal); // "perro" - desde adentro se ve afuera
  console.log(comida); // "croquetas" - estamos dentro de su casa

  if (true) {
    let juguete = "pelota"; // bloque anidado, aún más adentro
    console.log(animal); // "perro" - se ve desde dos niveles adentro
    console.log(comida); // "croquetas" - se ve desde un nivel adentro
    console.log(juguete); // "pelota"
  }

  // console.log(juguete); // ERROR si lo descomentas: juguete murió al cerrar su
  // bloque
}

// console.log(comida); // ERROR si lo descomentas: comida murió al cerrar el if
console.log(animal); // "perro" - la global sigue viva
```

La regla de oro del scope en una frase: el código puede ver hacia afuera, nunca hacia adentro. Si dibujas los bloques como cajas anidadas, cada caja ve su propio interior y todo lo que la contiene, pero nada de lo que está en cajas hermanas o hijas.

Ejemplo 5: `var` contra `let`, la demostración del problema. Este ejemplo muestra por qué la industria abandonó `var`. Ejecútalo y observa la diferencia de comportamiento.

```
// Con var: la variable se fuga del bloque
for (var i = 0; i < 3; i++) {
  // hacemos algo tres veces
}
console.log("Con var, i vale:", i); // 3 - ¡i sobrevivió al ciclo!

// Con let: la variable muere donde debe morir
for (let j = 0; j < 3; j++) {
  // hacemos algo tres veces
}
// console.log(j); // ERROR: j no existe aquí fuera. Correcto y esperado.
console.log("Con let, j no existe fuera del ciclo. Así debe ser.");
```

Esa *i* que sobrevive con *var* es una fuente clásica de bugs: imagina un programa grande donde más adelante, sin darte cuenta, usas otra variable llamada *i* pensando que es nueva, pero en realidad estás reutilizando la fugada del ciclo anterior con el valor 3 guardado. Con *let* eso es imposible: cada bloque tiene sus propias variables y al cerrarse se las lleva.

Ejemplo 6: buenos nombres contra malos nombres. Dos versiones del mismo programa: una tienda que calcula el cambio. La primera funciona pero es un jeroglífico; la segunda se explica sola. Acostumbra tus ojos a preferir la segunda.

```
// VERSIÓN MALA: funciona, pero nadie sabe qué hace sin estudiarla
let a = 500;
let b = 0.16;
let c = a * b;
let d = a + c;
let e = 1000;
let f = e - d;
console.log(f);

// VERSIÓN BUENA: el mismo programa, legible al instante
const precioBase = 500;
const tasaIva = 0.16;
const montoIva = precioBase * tasaIva;
const precioTotal = precioBase + montoIva;
const pagoRecibido = 1000;
const cambio = pagoRecibido - precioTotal;
console.log("Su cambio es: $" + cambio);
```

Fíjate también en el uso de `const` en la versión buena: ninguno de esos valores cambia después de calcularse, así que `const` es la declaración correcta para todos. Solo usarías `let` si, por ejemplo, el cliente agregara más productos al ticket y tuvieras que ir acumulando el total.

Ejemplo 7: el error del script en el lugar equivocado, y cómo diagnosticarlo. Este es el error número uno de los principiantes con la etiqueta `script`, así que vamos a provocarlo a propósito para aprender a reconocerlo. Crea este HTML con el script en el head SIN `defer`:

```
<!DOCTYPE html>
<html>
<head>
  <title>Error clásico</title>
  <script>
    // Este código corre ANTES de que el navegador lea el body.
    // El párrafo todavía no existe.
    const parrafo = document.getElementById("saludo");
    console.log(parrafo); // null: no lo encontró
    parrafo.textContent = "Hola"; // TypeError: no se puede leer textContent de
null
  </script>
</head>
<body>
  <p id="saludo">Texto original</p>
</body>
</html>
```

Abre la consola y verás el error en rojo: "Cannot read properties of null (reading 'textContent')". Traducido: intentaste usar `textContent` sobre algo que vale `null`, o sea, sobre nada. `getElementById` devolvió `null` porque el elemento aún no existía. Ahora aplica cualquiera de las dos soluciones que aprendiste: mueve el script al final del body, o agrega `defer` (si fuera archivo externo). El error desaparece. Cuando en el futuro veas "cannot read properties of null", tu primer sospechoso debe ser este: estás tocando un elemento que todavía no existe cuando tu código se ejecuta.

Ejemplo 8: mini-reto guiado. Vamos a juntar todo: variables, `const`, `let`, buenos nombres y `console.log`. El reto: calcular el salario semanal de un empleado con horas extra. Lee el enunciado, intenta resolverlo tú antes de ver la solución, y luego compara.

Enunciado del reto: un empleado trabaja 40 horas normales a \$120 por hora, más 5 horas extra que se pagan al doble. Calcula el pago normal, el pago de horas extra y el total. Usa const para todo lo que no cambie y let solo si lo necesitas. Muestra los tres resultados en consola con mensajes claros.

```
// Solución del mini-reto
const horasNormales = 40;
const horasExtra = 5;
const pagoPorHora = 120;
const factorHoraExtra = 2; // las horas extra se pagan al doble

const pagoNormal = horasNormales * pagoPorHora;
const pagoExtra = horasExtra * pagoPorHora * factorHoraExtra;
const salarioTotal = pagoNormal + pagoExtra;

console.log("Pago normal: $" + pagoNormal); // 4800
console.log("Pago horas extra: $" + pagoExtra); // 1200
console.log("Salario total: $" + salarioTotal); // 6000
```

¿Sabías que...? El operador + entre un texto y un número no suma matemáticamente: convierte el número a texto y los pega. Por eso "Su cambio es: \$" + cambio produce un mensaje legible. Este comportamiento se llama concatenación y lo veremos a fondo en la unidad de tipos de datos, pero ya lo estás usando desde ahora.

Práctica

Aquí van los ejercicios de la unidad. La regla del juego: intenta cada uno por tu cuenta antes de mirar la solución. Si te atorras más de quince minutos, mira la solución, entiéndela línea por línea, ciérrala y vuelve a escribirla tú sin copiar. El aprendizaje está en el esfuerzo, no en la respuesta.

Ejercicio 1: tarjeta de presentación en consola. Enunciado: declara tres variables con const: tu nombre, tu ciudad y tu lenguaje de programación favorito. Luego usa console.log para mostrar un mensaje como: "Hola, soy [nombre], vivo en [ciudad] y estoy aprendiendo [lenguaje]." usando concatenación con +.

```
// Solución
const nombre = "Esteban";
const ciudad = "Ciudad de México";
const lenguajeFavorito = "JavaScript";

console.log("Hola, soy " + nombre + ", vivo en " + ciudad + " y estoy aprendiendo "
+ lenguajeFavorito + ".");
```

Explicación paso a paso: primero declaramos las tres variables con `const` porque son datos que no van a cambiar durante el programa. Usamos nombres descriptivos en camelCase: `nombre`, `ciudad`, `lenguajeFavorito`. Después construimos el mensaje pegando fragmentos de texto con las variables usando el operador `+`. Fíjate en los espacios: cada fragmento de texto termina o empieza con un espacio ("Hola, soy" tiene espacio al final) para que las palabras no queden pegadas al valor de la variable. Este detalle de los espacios es un error típico: si escribes "soy" + nombre sin espacio, obtienes "soyEsteban". La consola muestra el mensaje completo armado.

Ejercicio 2: convertir temperaturas. Enunciado: declara una variable con `let` llamada `gradosCelsius` con valor 25. Declara una `const` llamada `factorConversion` con el valor 9/5 y otra `const` llamada `ajuste` con valor 32. Calcula los grados Fahrenheit con la fórmula $(\text{celsius} \times 9/5) + 32$, guárdalos en una variable y muéstralos en consola. Después cambia `gradosCelsius` a 30, recalcula y muestra el nuevo resultado.

```
// Solución
let gradosCelsius = 25;
const factorConversion = 9 / 5;
const ajuste = 32;

let gradosFahrenheit = gradosCelsius * factorConversion + ajuste;
console.log(gradosCelsius + "°C son " + gradosFahrenheit + "°F"); // 25°C son 77°F

gradosCelsius = 30; // let permite reasignar
gradosFahrenheit = gradosCelsius * factorConversion + ajuste; // hay que recalcular
console.log(gradosCelsius + "°C son " + gradosFahrenheit + "°F"); // 30°C son 86°F
```

Explicación paso a paso: `gradosCelsius` va con `let` porque el enunciado pide cambiarla después. Los valores 9/5 y 32 son constantes de la fórmula, así que van con `const` y con nombres que explican qué son. Calculamos Fahrenheit multiplicando y sumando según la fórmula; la precedencia de operadores hace primero la multiplicación, igual que en matemáticas. Mostramos el resultado con un mensaje descriptivo. Luego reasignamos

gradosCelsius a 30, lo cual solo es posible por ser `let`, y recalculamos Fahrenheit porque las variables guardan valores, no fórmulas: si no repetimos la operación, `gradosFahrenheit` seguiría valiendo 77. Este ejercicio graba a fuego dos ideas: `let` contra `const`, y que los valores calculados no se actualizan solos.

Ejercicio 3: encuentra el error de scope. Enunciado: el siguiente programa tiene un error de scope. Encuéntralo, explica por qué ocurre y escribe la versión corregida.

```
let edad = 17;
if (edad >= 18) {
  let acceso = "permitido";
} else {
  let acceso = "denegado";
}
console.log("Acceso: " + acceso);
```

```
// Solución corregida
let edad = 17;
let acceso; // declarada en el scope donde se va a usar
if (edad >= 18) {
  acceso = "permitido";
} else {
  acceso = "denegado";
}
console.log("Acceso: " + acceso); // Acceso: denegado
```

Explicación paso a paso: el error es que `acceso` se declara con `let` dentro de cada bloque del `if/else`, así que cada una vive solo dentro de su bloque y muere al cerrarse la llave. Cuando `console.log` intenta leer `acceso` fuera, la variable no existe y se lanza un `ReferenceError`. Además, en el código original hay dos variables distintas llamadas `acceso` (una por bloque), ninguna visible desde afuera. La corrección sigue el patrón estándar: declarar la variable con `let` en el scope exterior sin valor inicial, y asignarle el valor dentro de los bloques según la condición. Así la variable vive donde se necesita y los bloques solo deciden qué valor darle. Este patrón (declarar fuera, asignar dentro) lo vas a usar constantemente.

Ejercicio 4: reescribe con buenos nombres y `const/let` correctos. Enunciado: este código calcula el área de un rectángulo pero usa `var`, malos nombres y declaraciones incorrectas. Reescríbelo siguiendo las buenas prácticas de la unidad.

```
var b = 12;
var a = 8;
var r = b * a;
console.log(r);
```

```
// Solución
const base = 12;
const altura = 8;
const area = base * altura;
console.log("El área del rectángulo es: " + area);
```

Explicación paso a paso: primero eliminamos var y usamos const, porque ninguno de los valores cambia después de asignarse. Después renombramos: b se convierte en base, a en altura y r en area. Los nombres ahora describen qué representa cada dato, así que el programa se entiende sin comentarios. Finalmente mejoramos el console.log con un mensaje descriptivo en lugar de mostrar un número pelado. El programa hace exactamente lo mismo que el original, pero ahora cualquier persona (incluido tu yo del futuro) lo entiende al primer vistazo. Este ejercicio parece trivial y por eso es importante: la legibilidad se entrena con ejercicios pequeños hasta que se vuelve automática.

Ejercicio 5: script externo que saluda según la hora. Enunciado: crea un archivo saludo.js (externo) que declare con const la hora actual usando new Date().getHours(), y con let un mensaje vacío. Usa un if para asignar "Buenos días", "Buenas tardes" o "Buenas noches" según la hora, y muestra el mensaje en consola. Explica dónde pondrías la etiqueta script en el HTML y por qué.

```
// saludo.js
const horaActual = new Date().getHours();
let mensaje;

if (horaActual < 12) {
  mensaje = "Buenos días";
} else if (horaActual < 19) {
  mensaje = "Buenas tardes";
} else {
  mensaje = "Buenas noches";
}

console.log(mensaje + ". Son las " + horaActual + " horas.");
```



```
<!-- En el HTML, dentro del <head>: -->
<script src="saludo.js" defer></script>
```

Explicación paso a paso: horaActual va con const porque se calcula una vez y no cambia. mensaje va con let y se declara vacío porque su valor depende de la condición y se asigna dentro del if. (El if/else lo veremos a fondo en la unidad 2; aquí lo usamos de forma intuitiva: si la hora es menor a 12 es de mañana, si es menor a 19 es de tarde, si no es de noche.) La etiqueta script va en el head con defer porque es un archivo externo: defer garantiza que se descargue en segundo plano y se ejecute cuando el HTML esté completo, así que no bloquea la página y no hay riesgo de tocar elementos inexistentes. Si en lugar de archivo externo fuera código interno, la alternativa sería poner el script al final del body. Este ejercicio conecta los tres temas de la unidad: incluir JavaScript, variables y un adelanto de condicionales.

Ejercicio 6: detecta por qué var te traiciona. Enunciado: ejecuta mentalmente este código y predice qué imprime. Luego explica por qué con let el resultado sería distinto y cuál comportamiento es más seguro.

```
var contador = 0;
if (true) {
  var contador = 100;
}
console.log(contador);
```

```
// Solución y explicación
// Imprime: 100

// Con var, la segunda declaración no crea una variable nueva:
// redeclara la misma variable global y le pisa el valor.
// El if no crea un scope para var, así que ambas líneas hablan
// de la misma cajita.

// Con let, la segunda línea sería un error o una variable distinta:
// Versión con let que muestra el comportamiento seguro:
let contadorSeguro = 0;
if (true) {
  let contadorSeguro = 100; // variable DISTINTA, vive solo en el bloque
}
console.log(contadorSeguro); // 0: la de afuera no se tocó
```

Explicación paso a paso: con `var`, las dos declaraciones de contador se refieren a la misma variable porque `var` ignora los bloques. La asignación dentro del `if` pisa silenciosamente el valor original, y el programa imprime 100 aunque el programador quizá esperaba que el 100 viviera solo dentro del `if`. Con `let`, en cambio, la variable dentro del bloque es una variable completamente distinta que muere al cerrarse la llave, y la de afuera conserva su valor 0. El comportamiento de `let` es más seguro porque hace lo que el código aparenta hacer: lo que pasa dentro del bloque se queda dentro del bloque. Este es exactamente el tipo de bug silencioso por el que `var` quedó jubilado.

Ejercicio 7: mini-reto integrador, la tienda de tacos. Enunciado: una taquería vende cada taco a \$18. Un cliente pide 12 tacos. Calcula el subtotal, el IVA (16%) y el total. Guarda cada dato en variables con nombres descriptivos, usando `const` donde corresponda. Muestra un ticket legible en consola con los tres montos. Después imagina que el cliente pide 3 tacos más: actualiza la cantidad usando `let` donde tenga sentido y vuelve a mostrar el ticket.

```
// Solución
const precioPorTaco = 18;
const tasaIva = 0.16;
let cantidadTacos = 12;

let subtotal = cantidadTacos * precioPorTaco;
let montoIva = subtotal * tasaIva;
let total = subtotal + montoIva;

console.log("--- TICKET ---");
console.log("Tacos: " + cantidadTacos + " x $" + precioPorTaco);
console.log("Subtotal: $" + subtotal);
console.log("IVA (16%): $" + montoIva);
console.log("Total: $" + total);

// El cliente pide 3 tacos más
cantidadTacos = cantidadTacos + 3; // ahora son 15

// Recalculamos todo porque los valores no se actualizan solos
subtotal = cantidadTacos * precioPorTaco;
montoIva = subtotal * tasaIva;
total = subtotal + montoIva;

console.log("--- TICKET ACTUALIZADO ---");
console.log("Tacos: " + cantidadTacos + " x $" + precioPorTaco);
console.log("Subtotal: $" + subtotal);
console.log("IVA (16%): $" + montoIva);
console.log("Total: $" + total);
```

Explicación paso a paso: precioPorTaco y tasaIva son datos fijos del negocio, así que van con const. cantidadTacos va con let porque el enunciado dice que puede cambiar. subtotal, montoIva y total también van con let porque hay que recalcularlos cuando cambia la cantidad. Calculamos en orden lógico: primero el subtotal (cantidad por precio), luego el IVA sobre el subtotal, luego el total sumando ambos. Mostramos un ticket legible con etiquetas claras. Cuando el cliente agrega tacos, actualizamos cantidadTacos sumándole 3 y repetimos los tres cálculos, porque como ya aprendimos, las variables guardan resultados, no fórmulas. Este ejercicio integra todo lo de la unidad: declaraciones correctas, buenos nombres, reasignación con let, protección con const y presentación de resultados. Si dominas este ejercicio sin mirar la solución, la unidad 1 está dominada.

Cuidado con...: un error frecuente en ejercicios como este es declarar todo con const y luego intentar reasignar cantidadTacos. Si te pasa, la consola te lo dirá con un TypeError claro. No te

frustres: ese error es el lenguaje enseñándote. Cambia a `let` solo las variables que realmente cambian y deja `const` en las demás.

¿Sabías que...? El `console.log` acepta varios valores separados por comas y los muestra espaciados: `console.log("Total:", total)` es más limpio que concatenar con `+`. Ambos funcionan; con el tiempo desarrollarás tu preferencia. También existe `console.table` para mostrar arrays y objetos como tablas, que te va a encantar cuando lleguemos a esas unidades.

Galería de errores típicos (y cómo leerlos)

Antes del resumen, hagamos algo muy práctico: un tour por los mensajes de error que más vas a ver en esta etapa. Cada programador profesional los ha visto miles de veces. La diferencia entre un principiante y un veterano no es que el veterano no cometa errores, es que el veterano lee el mensaje, entiende qué pasó y lo corrige en segundos. Vamos a entrenar ese músculo desde ahora.

Error 1: "Uncaught ReferenceError: nombreVariable is not defined". Traducción: intentaste usar una variable que no existe en ese scope. Causas típicas: la escribiste mal (`nombreUsuario` contra `nombreusuario`, recuerda que mayúsculas y minúsculas importan), la declaraste dentro de un bloque y la usas fuera, o simplemente olvidaste declararla. La solución casi siempre es revisar el nombre letra por letra y verificar dónde nació la variable.

```
// Provoca el error a propósito para reconocerlo
console.log(ciudadNatal); // ReferenceError: ciudadNatal is not defined
let ciudadNatal = "Puebla"; // la declaramos DESPUÉS de usarla
```

Fíjate que aquí el problema es el orden: con `let` no puedes usar la variable antes de su declaración por la zona muerta temporal. Si necesitas el valor antes, mueve la declaración más arriba. El lenguaje te está diciendo "todavía no estoy lista, espérame una línea".

Error 2: "Uncaught TypeError: Assignment to constant variable". Traducción: intentaste reasignar una `const`. La solución es decidir: si el valor realmente debe cambiar, la declaración correcta era `let`; si no debe cambiar, el que está mal es la línea que intenta cambiarlo. Este error es tu amigo: te avisa de una contradicción en tu propio diseño.

```
const limiteIntentos = 3;
limiteIntentos = 5; // TypeError: Assignment to constant variable
// Pregúntate: ¿el límite debe poder cambiar? Si sí, usa let. Si no, borra esta
línea.
```

Error 3: "Uncaught TypeError: Cannot read properties of null (reading 'textContent')". Ya lo vimos en los ejemplos: intentaste manipular un elemento HTML que no existe (todavía). El diagnóstico en dos pasos: primero verifica que el id en getElementById esté escrito exactamente igual que en el HTML; segundo verifica dónde está tu script y si el elemento ya existe cuando el código corre. El noventa por ciento de las veces es una de esas dos cosas.

Error 4: "Uncaught SyntaxError: Unexpected token". Traducción: escribiste algo que no es JavaScript válido, y el motor ni siquiera pudo empezar a leer tu programa. Causas típicas: una llave o paréntesis sin cerrar, una comilla sin su pareja, o una palabra mal escrita. El mensaje te dice en qué línea (aproximadamente) está el problema. Empieza a contar llaves: cada { necesita su}.

```
let nombre = "Ana;  
// SyntaxError: la comilla de apertura nunca se cerró.  
// El motor siguió leyendo buscando el cierre y se perdió.
```

Un consejo de veterano sobre los errores: lee el mensaje completo, no solo la primera palabra. Los mensajes modernos de los navegadores son sorprendentemente buenos: te dicen el tipo de error, qué salió mal y en qué archivo y línea. Y cuando busques el error en internet, copia el mensaje exacto entre comillas: alguien ya tuvo tu mismo problema y lo resolvió.

¿Sabías que...? Los programadores experimentados provocan errores a propósito todo el tiempo. Escriben una línea, la ejecutan, ven qué pasa, la rompen, la arreglan. A esa práctica se le llama "programación exploratoria" y es una de las formas más rápidas de aprender un lenguaje nuevo. La consola del navegador es tu laboratorio: úsala sin miedo, porque ahí nada se rompe de verdad.

Cuidado con...: el error más caro de esta unidad no es técnico, es de hábito: copiar y pegar código sin entenderlo. Funciona para el ejercicio, pero el conocimiento no se te queda. Cada ejemplo de este libro está diseñado para que lo escribas tú, lo ejecutes y lo modifiques. Si solo lees, en dos semanas no recordarás nada. Si practicas, en dos semanas estarás resolviendo ejercicios nuevos por tu cuenta.

Mini-retos extra

Reto A: el programador olvidadizo. Declara tu nombre con `const`, tu edad con `let` y tu pasatiempo favorito con `const`. Luego intenta cambiar tu nombre (a propósito) y observa el error. Después cambia tu edad correctamente y muestra los tres datos en un solo `console.log`. Este reto te entrena a distinguir qué debe ser `const` y qué debe ser `let`, y a reconocer el `TypeError` de reasignación cuando lo veas en código real.

Reto B: el scope detective. Sin ejecutar el código, predice qué imprime cada `console.log` y si alguno lanza error. Luego ejecútalo y verifica.

```
let x = 10;
if (x > 5) {
  let y = x * 2;
  console.log("Dentro:", y);
  x = y + 5;
}
console.log("Fuera:", x);
// console.log(y); // ¿qué pasaría si descomentas esta línea?
```

Respuesta: dentro imprime 20, fuera imprime 25 (porque `x` se reasignó dentro del bloque a `20 + 5`, y `x` es la variable global, visible y modificable desde adentro). La línea comentada lanzaría `ReferenceError` porque `y` murió al cerrarse el `if`. Si predijiste todo bien, el scope ya es tuyo.

Reto C: organiza tu escritorio. Escribe un programa que declare variables para describir tu escritorio real: qué hay encima, cuántos objetos hay, si la lámpara está encendida (booleano, adelante de la unidad 2) y el nombre de tu taza favorita. Usa `const` para lo que no cambia y `let` para lo que sí. Muestra una descripción completa en consola. Este reto parece tonto y es poderosísimo: te obliga a modelar el mundo real con variables, que es exactamente lo que hace un programador todos los días.

Caso de estudio: de página estática a página viva

Vamos a cerrar la parte de explicación con un caso de estudio completo que une los tres temas de la unidad. Imagina que tienes una página de presentación personal: tu nombre, tu ciudad, a qué te dedicas. Hoy está escrita a mano en HTML, estática, aburrida. La vamos a convertir en una página viva donde JavaScript genera el contenido desde variables. Este es, en miniatura, el trabajo real de un desarrollador frontend: los datos viven en JavaScript y el HTML es solo el escenario.

Paso 1: el HTML con los huecos. Creamos una estructura con elementos vacíos que JavaScript va a llenar. Cada elemento tiene un id único, que es como su nombre para que JavaScript lo encuentre.

```
<!DOCTYPE html>
<html>
<head>
  <title>Mi presentación</title>
  <script src="presentacion.js" defer></script>
</head>
<body>
  <h1 id="nombre"></h1>
  <p id="descripcion"></p>
  <p id="datoCurioso"></p>
</body>
</html>
```

Fíjate que el h1 y los párrafos están vacíos: no tienen texto dentro. Eso es intencional. El contenido lo va a poner JavaScript. Y el script va en el head con defer, la forma profesional: se descarga mientras el navegador lee el HTML y se ejecuta cuando todo está listo.

Paso 2: el JavaScript con los datos. En presentacion.js declaramos los datos como variables y luego los escribimos en la página.

```
// presentacion.js
// Los datos de la persona viven aquí, en variables.
// Si quieres cambiar la presentación, cambias estas líneas y listo.

const nombreCompleto = "Esteban Rivera";
const ciudad = "Guadalajara";
const profesion = "desarrollador web en formación";
const anioInicio = 2026;
const anioActual = 2026;
let aniosProgramando = anioActual - anioInicio;

// Construimos los textos usando las variables
const titulo = "Hola, soy " + nombreCompleto;
const descripcion = "Vivo en " + ciudad + " y soy " + profesion + ".";
const dato = "Llevo " + aniosProgramando + " años programando.";

// Los escribimos en la página buscando cada elemento por su id
document.getElementById("nombre").textContent = titulo;
document.getElementById("descripcion").textContent = descripcion;
document.getElementById("datoCurioso").textContent = dato;

console.log("Presentación generada desde variables");
```

Paso 3: entiende por qué esto es poderoso. En la versión estática, si cambias de ciudad tienes que editar el HTML. En esta versión, cambias una variable y toda la página se actualiza. Parece poca cosa con tres líneas, pero escala: imagina una tienda con mil productos donde el nombre, el precio y la descripción de cada uno vienen de variables. Cambiar un precio sería editar una línea en lugar de mil. Esa es la magia de separar datos (variables en JavaScript) de presentación (HTML): los datos mandan, la página obedece.

Además observa las decisiones de declaración: nombreCompleto, ciudad y profesion son const porque son datos fijos de la persona. aniosProgramando es let porque se calcula restando años y podría actualizarse. titulo, descripcion y dato son const porque una vez armados no cambian. Cada let y cada const cuenta una historia sobre la intención del programador.

Paso 4: rómpelo y arréglalo (tu tarea). Haz estos experimentos y observa qué pasa: primero cambia el id en el JavaScript a un nombre que no exista en el HTML y mira el error en consola (el famoso null). Luego quita el defer del script y observa qué cambia (pista: puede que todo siga funcionando si el archivo es pequeño y rápido, o puede fallar; esa inconsistencia es justo por qué defer existe: elimina la lotería). Después cambia una const por let y viceversa donde

no corresponda, y lee los errores. Cada experimento te deja una cicatriz de aprendizaje que ningún tutorial te puede dar leído.

Este caso de estudio es el puente perfecto hacia la unidad 2: ya sabes guardar datos y mostrarlos; ahora aprenderás qué clases de datos existen, cómo operarlos y cómo tomar decisiones con ellos. La página viva que acabas de hacer se va a volver inteligente: saludará distinto según la hora, mostrará mensajes diferentes según condiciones y calculará cosas con operadores. Todo eso viene ahora.

Preguntas de repaso

Antes de pasar al resumen, verifica que puedes responder estas preguntas sin mirar atrás. Si alguna te cuesta, regresa a su sección: es más rápido repasarla ahora que arrastrar la duda a la unidad 2.

1. ¿Por qué JavaScript se llama así si no tiene que ver con Java?
2. ¿Qué significa que el navegador "bloquee" la lectura del HTML al encontrar un script?
3. ¿Qué hace exactamente el atributo defer y en qué se diferencia de async?
4. ¿Por qué un script con src ignora el código escrito dentro de la etiqueta?
5. ¿Cuál es la diferencia práctica entre let y const?
6. ¿Por qué se recomienda usar const por defecto?
7. ¿Qué significa que const no congela el contenido de un objeto o array?
8. ¿Por qué var se considera peligroso? Menciona dos comportamientos problemáticos.
9. ¿Qué es el scope y cuál es su regla de oro en una frase?
10. ¿Qué pasa si asignas un valor a un nombre sin declararlo con let o const?
11. ¿Por qué las variables guardan valores y no fórmulas? ¿Qué consecuencia práctica tiene?
12. ¿Qué convención de nombres se usa para variables en JavaScript y por qué importan los buenos nombres?

Resumen de la unidad

JavaScript es el lenguaje de la web: nació en 1995 para dar interactividad a las páginas y hoy corre en navegadores, servidores, móviles y escritorio gracias a motores como V8 y a Node.js. No tiene relación con Java más allá del nombre, que fue una decisión de mercadotecnia.

El JavaScript moderno es el posterior a ES6 (2015): let, const, funciones flecha y demás. Los tutoriales que enseñan var como estándar están desactualizados; reconócelos y evítalos.

Se incluye JavaScript en HTML con la etiqueta script: código interno entre las etiquetas para pruebas rápidas, o archivo externo con src para proyectos reales. Nunca combines src con código interno en la misma etiqueta porque el código se ignora.

El navegador lee el HTML de arriba a abajo y los scripts lo bloquean mientras se ejecutan. Por eso el código que toca elementos de la página debe correr cuando esos elementos ya existen: script al final del body, o archivo externo con defer en el head, que descarga en segundo plano y ejecuta al terminar de leer el documento.

Las variables son espacios de memoria con nombre para guardar datos. Se declaran con let si el valor va a cambiar y con const si no. La regla profesional: const por defecto, let solo cuando el cambio sea necesario.

const impide la reasignación pero no congela el interior de objetos y arrays. const exige valor inicial; let puede declararse sin valor y vale undefined hasta que se le asigne algo.

var está jubilado: ignora los bloques, permite redeclaraciones silenciosas y crea bugs difíciles de rastrear. Existe solo por compatibilidad con código viejo. En código nuevo siempre let y const.

Los nombres de variables usan camelCase, son descriptivos y se leen como historia: precioTotal en lugar de x. Las constantes universales van en MAYÚSCULAS. Los booleanos se nombran como preguntas: esMayorDeEdad.

El scope determina dónde vive cada variable: el código ve hacia afuera pero nunca hacia adentro. let y const mueren al cerrarse su bloque. Para usar un valor fuera de un bloque, declara la variable fuera y asígnale el valor dentro.

Declara siempre tus variables con let o const. Asignar sin declarar crea globales accidentales, una de las peores fuentes de bugs del lenguaje.

El ciclo de aprendizaje que funciona: lee el ejemplo, ejecútalo, predice el resultado, rómpelo a propósito y arréglalo. Los mensajes de error en rojo no son tu enemigo: son el lenguaje diciéndote exactamente qué corregir.

Con esta base ya puedes guardar datos, nombrarlos bien, controlar dónde viven y conectar tu código con una página HTML. En la unidad 2 les daremos tipos, operaciones y decisiones: qué clase de datos guardas, cómo operarlos y cómo hacer que tu programa tome caminos distintos según las condiciones.

Palabra final: si llegaste hasta aquí escribiendo y ejecutando el código, ya no eres la misma persona que empezó la unidad. Ya sabes qué es JavaScript, cómo conectarlo a una página sin romper nada, cómo guardar datos con `let` y `const`, cómo nombrarlos para que se entiendan solos y cómo el scope decide dónde vive cada variable. Son pocos conceptos, pero son los que vas a usar en cada programa que escribas de aquí en adelante. Guárdalos bien, porque la unidad 2 construye directamente sobre ellos: los datos que hoy guardas en variables mañana tendrán tipos, operaciones y decisiones. Nos vemos ahí. Sigue practicando todos los días, aunque sea media hora: la constancia le gana al talento.

UNIDAD 2: Datos y decisiones

Introducción

En la unidad 1 aprendiste a guardar información en variables. Ahora viene la siguiente pregunta natural, la que todo programador se hace cientos de veces al día: ¿qué clase de información estoy guardando? Porque no es lo mismo guardar un nombre que guardar una edad, y la computadora lo sabe. Guardar "25" como texto no es lo mismo que guardar 25 como número, aunque se vean igual en la pantalla. Esa diferencia invisible es la fuente de algunos de los bugs más confusos para los principiantes... y de algunos de los poderes más útiles del lenguaje cuando la entiendes.

Esta unidad se llama "Datos y decisiones" porque cubre las dos habilidades que convierten a un programa tonto en un programa inteligente. Primero los datos: los tipos de datos de JavaScript (textos, números, booleanos y los valores especiales null y undefined), cómo preguntarle al lenguaje qué tipo es un valor con `typeof`, y cómo convertir un tipo en otro cuando lo necesitas. Después las decisiones: los condicionales, la capacidad del programa de tomar caminos distintos según las circunstancias. Un programa sin condicionales es como un tren: avanza por una sola vía sin importar lo que pase. Un programa con condicionales es como un conductor: mira el camino y decide.

Piénsalo con un ejemplo de la vida real. Una tienda en línea necesita calcular el total de una compra. Los precios son números y se suman con aritmética. Pero el nombre del cliente es texto y no se suma, se muestra. Si el cliente tiene cupón de descuento, el programa debe decidir: ¿aplico el descuento o no? Y para decidir necesita comparar: ¿el cupón es válido? ¿la compra supera el mínimo? Cada una de esas preguntas se responde con los temas de esta unidad: tipos de datos para saber qué estás manipulando, operadores aritméticos para calcular, comparaciones y condicionales para decidir.

¿Por qué importa tanto entender los tipos de datos? Porque JavaScript es un lenguaje de tipado dinámico y débil, dos palabras que suenan a insulto pero que en realidad describen su flexibilidad. Tipado dinámico significa que una variable no está atada a un tipo: la misma variable puede guardar un número ahora y un texto después, y el lenguaje no se queja. Tipado débil significa que el lenguaje convierte tipos automáticamente cuando le conviene, a veces ayudándote y a veces traicionándote. Esa flexibilidad hace que JavaScript sea rápido de escribir y fácil de empezar, pero también significa que tú, el programador, debes estar atento a qué tipo tiene cada valor. En lenguajes de tipado fuerte el compilador te grita si mezclas tipos;

en JavaScript nadie te grita, el programa simplemente hace algo raro y te deja adivinar qué pasó. Por eso esta unidad existe: para que seas tú quien manda sobre los tipos, no al revés.

Los condicionales, por su parte, son la primera estructura de control que vas a dominar. Una estructura de control es una instrucción que altera el flujo normal del programa, que de otro modo ejecutaría línea por línea de arriba a abajo. Con `if`, `else if` y `else` le dices al programa: "si pasa esto, haz aquello; si no, haz esto otro". Es la forma más directa de modelar decisiones del mundo real en código: si el usuario es mayor de edad, muéstrale el contenido; si no, muéstrale un aviso. Si la contraseña es correcta, deja entrar; si no, muestra error. Si quedan asientos, vende el boleto; si no, avisa que se agotaron. Toda aplicación que usas está llena de miles de estas micro-decisiones.

Al terminar esta unidad vas a poder escribir programas que reciben datos, los clasifican por tipo, hacen cálculos con ellos y toman decisiones ramificadas. Es un salto enorme respecto a la unidad 1: tus programas van a dejar de ser recetas lineales para convertirse en agentes que reaccionan. Y todo lo que sigue en el curso (ciclos, funciones, arrays, objetos, el DOM, eventos) se construye sobre estos dos pilares. Los ciclos repiten decisiones, las funciones empaquetan decisiones reutilizables, los eventos son decisiones que esperan a que el usuario haga algo. Domina datos y decisiones, y el resto del curso se te va a hacer mucho más natural.

Un aviso honesto: esta unidad tiene fama de ser "fácil" y por eso mucha gente la lee por encima. Error grave. Los temas parecen simples (un `if`, una suma), pero los detalles (coerción de tipos, igualdad estricta contra igualdad débil, anidamiento) son donde se esconden los bugs que persiguen a los programadores por años. Léela con calma, haz todos los ejercicios, y cuando algo te sorprenda, celebra: cada sorpresa es un bug futuro que acabas de evitar.

Explicación detallada

Los tipos de datos: qué clase de dato guardas

Todo valor en JavaScript pertenece a un tipo de dato. El tipo define qué puedes hacer con ese valor y cómo se comporta en las operaciones. Es como la diferencia entre una palabra y un número en la vida real: con palabras formas frases, con números haces cuentas. Si intentas hacer cuentas con palabras, el resultado no tiene sentido, y viceversa.

JavaScript tiene siete tipos primitivos (los valores simples, indivisibles) y los objetos (las estructuras compuestas, que veremos en su unidad). En esta unidad nos concentramos en los

cinco primitivos que vas a usar todos los días: string, number, boolean, undefined y null. (Existen también symbol y bigint, pero son temas avanzados que no necesitas ahora; solo debes saber que existen para no asustarte si los ves mencionados.)

El string es texto. Cualquier secuencia de caracteres entre comillas: nombres, mensajes, direcciones, descripciones. Puedes usar comillas dobles, comillas simples o backticks; las tres formas crean strings, con matices que veremos en la unidad de template literals.

```
let nombre = "Esteban";  
let saludo = 'Hola, ¿cómo estás?';  
let mensajeVacio = ""; // un string vacío también es un string
```

El number es número: enteros, decimales, negativos, todo cabe en el mismo tipo. A diferencia de otros lenguajes, JavaScript no distingue entre enteros y decimales; todo es number.

```
let edad = 30; // entero  
let precio = 99.99; // decimal  
let temperatura = -5; // negativo  
let divisionRara = 1 / 3; // 0.3333333333333333
```

Ese último ejemplo muestra algo importante: las computadoras representan los decimales en binario y algunos no se pueden representar con exactitud, así que obtienes esos decimales infinitos raros. El caso más famoso es $0.1 + 0.2$, que en JavaScript da 0.30000000000000004 en lugar de 0.3 . No es un bug de tu código, es una limitación de cómo las computadoras guardan decimales, y le pasa a casi todos los lenguajes. La lección práctica: nunca compares decimales con igualdad exacta en cálculos de dinero; redondea o trabaja con centavos enteros. Lo veremos con ejemplos.

El boolean es el tipo más simple y uno de los más poderosos: solo tiene dos valores posibles, true (verdadero) y false (falso). Los booleanos son el idioma de las decisiones: cada condición que evalúes en un if termina convertida en un booleano. ¿El usuario es mayor de edad? true o false. ¿La contraseña coincide? true o false. Todo el sistema de decisiones del programa corre sobre booleanos.

```
let esMayorDeEdad = true;  
let tieneDescuento = false;  
let sesionActiva = true;
```

Recuerda la convención de nombres de la unidad 1: los booleanos se nombran como preguntas (es..., tiene..., puede..., esta...) para que el código se lea como frase natural.

undefined significa "esta variable existe pero todavía no tiene valor". Es el valor que JavaScript asigna automáticamente a las variables declaradas con `let` sin inicializar, a los parámetros de función no recibidos y a las propiedades de objeto inexistentes. Es el lenguaje diciéndote "aquí hay un hueco, aún no hay nada".

```
let puntaje;  
console.log(puntaje); // undefined: existe, pero sin valor
```

`null` es distinto y la diferencia importa: `null` significa "aquí decidí intencionalmente que no hay valor". Es un valor que tú pones a propósito para decir "vacío por decisión", mientras que `undefined` suele ser "vacío porque aún no se asignó nada". Piensa en un formulario: un campo que el usuario dejó en blanco podría ser `undefined` (nadie lo tocó), mientras que un campo que el sistema limpió a propósito podría ser `null` (alguien decidió vaciarlo). En la práctica muchos programadores los usan de forma intercambiable, pero entender la distinción te hace escribir código más claro y te evita confusiones al depurar.

```
let respuestaUsuario; // undefined: el usuario aún no responde  
let ganadordelSorteo = null; // null: decidimos que aún no hay ganador
```

¿Sabías que...? `null` es técnicamente un objeto según `typeof`, por un bug histórico del lenguaje que nunca se corrigió por compatibilidad. `typeof null` devuelve "object" en lugar de "null". Es uno de esos fósiles de los diez días de creación que mencionamos en la unidad 1. Cuando necesites verificar si algo es `null`, compáralo directamente con `=== null` en lugar de confiar en `typeof`.

typeof: pregúntale al lenguaje qué tienes

Como JavaScript es de tipado dinámico, los valores cambian de tipo sin avisar y a veces no estás seguro de qué tienes entre manos. Para eso existe `typeof`, un operador que te dice el tipo de cualquier valor como un string.

```
typeof "Hola"; // "string"  
typeof 42; // "number"  
typeof true; // "boolean"  
typeof undefined; // "undefined"  
typeof null; // "object" (el bug histórico, recuerda)
```

typeof es tu herramienta de diagnóstico. Cuando un cálculo da un resultado raro, lo primero que debes preguntarte es "¿qué tipos estoy mezclando?" y typeof te da la respuesta en un segundo. Úsalo en la consola cada vez que algo no cuadre:

```
let edad = "30"; // ¡cuidado! esto es un string, no un number
console.log(typeof edad); // "string": ahí está el problema
console.log(edad + 5); // "305": concatenó en lugar de sumar
```

Ese ejemplo es oro: edad vale "30" como texto (quizá vino de un formulario, donde todo llega como string), y al sumarle 5 el resultado es "305" porque el + con strings concatena. El programa no lanzó ningún error; simplemente hizo algo incorrecto en silencio. Con typeof descubres el culpable en segundos. Sin typeof, podrías pasar horas revisando la lógica cuando el problema eran los tipos desde el inicio.

Conversiones: cambiar de tipo a propósito

Muchas veces necesitas convertir un valor de un tipo a otro: el usuario escribe su edad en un campo de texto (string) y tú necesitas ese dato como número para calcular. O tienes un número y necesitas mostrarlo como parte de un mensaje (string). Las conversiones explícitas son la forma segura y legible de hacerlo.

De string a number tienes tres herramientas principales. Number(valor) convierte el string completo a número; si el string no es un número válido, devuelve NaN (Not a Number, "no es un número", un valor especial del tipo number que significa "esto intentó ser número y falló"). parseInt(valor) extrae el entero inicial del string, ignorando lo demás. parseFloat(valor) extrae el decimal inicial.

```
Number("42"); // 42
Number("42.5"); // 42.5
Number("hola"); // NaN
Number(""); // 0 (¡cuidado! el string vacío se convierte a 0)
parseInt("42 años"); // 42: extrae el número inicial
parseInt("42.9"); // 42: corta los decimales
parseFloat("42.9"); // 42.9: conserva los decimales
parseFloat("cuesta 42"); // NaN: no empieza con número
```

De number (o cualquier cosa) a string: String(valor) convierte explícitamente, y el truco valor + "" (concatenar con string vacío) hace lo mismo de forma implícita. Ambos son comunes; String() es más legible para quien lee tu código.


```
String(42); // "42"  
String(true); // "true"  
let precio = 99.99;  
let etiqueta = "Precio: $" + precio; // "Precio: $99.99" (conversión implícita)
```

A boolean: Boolean(valor) convierte cualquier valor a true o false según las reglas de truthy y falsy que veremos en la sección de condicionales. Por ahora quédate con la idea de que la conversión existe y es explícita.

La regla de oro de las conversiones: hazlas explícitas y hazlas temprano. En cuanto recibas un dato del usuario o de un formulario, conviértelo al tipo que necesitas antes de operar con él. No dejes que el lenguaje adivine: cuando JavaScript adivina, a veces acierta y a veces te entrega un "305" donde esperabas un 35. Los programadores veteranos convierten en la puerta de entrada de los datos, y a partir de ahí trabajan tranquilos.

La coerción: cuando JavaScript adivina por ti

Si las conversiones explícitas son tú tomando el control, la coerción es JavaScript tomando decisiones por ti cuando mezclas tipos en una operación. A veces el resultado es conveniente, a veces es una trampa. Necesitas conocer las reglas básicas para no caer en las trampas.

La regla más importante: el operador + prefiere los strings. Si alguno de los dos operandos es un string, el + convierte el otro a string y concatena. Los demás operadores aritméticos (-, *, /) prefieren los números: intentan convertir los strings a número y operar.

```
"5" + 3; // "53": el + vio un string y concatenó  
"5" - 3; // 2: el - convirtió "5" a número y restó  
"5" * 2; // 10: el * también convierte a número  
"10" / "2"; // 5: ambos se convirtieron a número
```

Esto explica el famoso "305": "30" + 5 concatena porque hay un string. Y también explica por qué "30" - 5 sí da 25: la resta no sabe concatenar, así que convierte. Es un comportamiento consistente una vez que conoces la regla, pero traicionero si no la conoces.

Cuidado con los casos raros que nacen de la coerción: `Number("")` es 0, `Number(" ")` también es 0, `Number(null)` es 0, pero `Number(undefined)` es NaN. `true` se convierte a 1 y `false` a 0 en operaciones numéricas. `null == undefined` es `true` pero `null === undefined` es `false`. No necesitas memorizar toda la tabla; necesitas el hábito de convertir explícitamente para no depender de estas reglas. Cada vez que veas un comportamiento extraño con tipos mezclados, la solución profesional es la misma: convierte tú los tipos antes de operar y el misterio desaparece.

Operadores aritméticos: la calculadora del programa

Con los tipos claros, pasemos a operar con números. JavaScript te da los operadores aritméticos clásicos: `+` suma, `-` resta, `*` multiplicación, `/` división, `%` módulo (el residuo de la división) y `**` potencia. Además tienes los operadores de incremento `++` y decremento `--`, y las formas abreviadas de asignación (`+=`, `-=`, `*=`, `/=`, `%=`).

```
let a = 10;
let b = 3;
console.log(a + b); // 13
console.log(a - b); // 7
console.log(a * b); // 30
console.log(a / b); // 3.3333333333333335
console.log(a % b); // 1: 10 entre 3 cabe 3 veces y sobra 1
console.log(a ** b); // 1000: 10 elevado a la 3
```

Detengámonos en el módulo `%`, porque es el operador más subestimado y uno de los más útiles. El módulo te da el residuo de una división entera: `a % b` es lo que sobra al repartir `a` entre `b`. Sus usos son legendarios: saber si un número es par o impar (`n % 2 === 0` es par), ciclar por turnos, repartir elementos en grupos, validar múltiplos.

```
// ¿Es par o impar?
let numero = 7;
console.log(numero % 2); // 1: impar (los pares dan 0)

// Repartir 10 dulces entre 3 niños: ¿cuántos sobran?
console.log(10 % 3); // 1: sobra un dulce

// ¿El año es bisiesto? (regla simplificada: divisible entre 4)
let anio = 2028;
console.log(anio % 4 === 0); // true
```

La precedencia de operadores sigue las reglas de las matemáticas que aprendiste en la escuela: primero paréntesis, luego potencias, luego multiplicación/división/módulo, al final suma/resta. Y como en la escuela, cuando dudes, usa paréntesis: no cuestan nada y eliminan toda ambigüedad.

```
console.log(2 + 3 * 4); // 14, no 20: primero la multiplicación
console.log((2 + 3) * 4); // 20: los paréntesis mandan
```

Los operadores de asignación abreviada son azúcar sintáctica para el patrón más común del mundo: tomar una variable, operarla con algo y guardar el resultado en ella misma.

```
let puntos = 10;
puntos += 5; // puntos = puntos + 5 -> 15
puntos -= 3; // puntos = puntos - 3 -> 12
puntos *= 2; // puntos = puntos * 2 -> 24
puntos /= 4; // puntos = puntos / 4 -> 6
puntos %= 4; // puntos = puntos % 4 -> 2
```

Y el incremento/decremento: contador++ suma uno, contador-- resta uno. Son ubicuos en ciclos (unidad siguiente) y en contadores de todo tipo. Existe una diferencia entre ++x (pre-incremento) y x++ (post-incremento) cuando se usan dentro de expresiones más grandes, pero como buena práctica úsalos solos en su propia línea y esa diferencia jamás te va a morder.

Comparaciones: hacer preguntas al programa

Para decidir, el programa necesita hacer preguntas, y las preguntas en código son comparaciones. JavaScript te da: === igualdad estricta, !== desigualdad estricta, > mayor que, < menor que, >= mayor o igual, <= menor o igual. (Existen también == y !=, la igualdad débil, pero vas a ver por qué debes evitarlos.)

El resultado de toda comparación es un booleano: true o false. Ese booleano es lo que los condicionales van a evaluar.

```
let edad = 20;
console.log(edad > 18); // true
console.log(edad < 18); // false
console.log(edad >= 20); // true
console.log(edad <= 19); // false
```

Ahora el tema estrella: `===` contra `==`. La igualdad estricta `===` compara valor Y tipo: solo es true si ambos lados son del mismo tipo y tienen el mismo valor. La igualdad débil `==` compara solo el valor, convirtiendo tipos automáticamente antes de comparar. Esa conversión automática es la puerta de entrada a una familia entera de sorpresas desagradables:

```
"5" === 5; // false: string contra number, tipos distintos
"5" == 5; // true: convierte el string a número y compara

0 == false; // true: ¡el 0 se convierte a false!
"" == false; // true: ¡el string vacío también!
null == undefined; // true: pero son cosas distintas
```

¿Ves el peligro? Con `==`, el lenguaje decide por ti qué significa "igual", y sus decisiones son discutibles. Con `===`, tú tienes el control total: sabes exactamente qué se está comparando. Por eso la regla profesional es absoluta y sin excepciones: usa siempre `===` y `!==`, nunca `==` ni `!=`. Si necesitas comparar un string con un número, convierte tú uno de los dos explícitamente y luego usa `===`. Así el código dice exactamente lo que hace y no hay magia escondida.

Esta regla es tan importante que las herramientas profesionales de análisis de código (linters) marcan el `==` como error automáticamente. Grábatela como un mandamiento: igualdad estricta siempre.

Condicionales: el programa toma decisiones

Llegamos al corazón de la unidad: el `if`. Su estructura es simple y su poder es inmenso.

```
if (condicion) {
  // este bloque se ejecuta SOLO si la condicion es true
}
```

El programa evalúa la condición, que debe resultar en un booleano (o en algo convertible a booleano). Si es true, ejecuta el bloque entre llaves. Si es false, lo salta y sigue con lo que sigue después. Así de simple, así de poderoso.

```
let edad = 20;

if (edad >= 18) {
  console.log("Puedes entrar al concierto");
}

console.log("Fin del programa");
```

Si edad es 20, se imprime el mensaje y luego "Fin del programa". Si edad fuera 15, el mensaje se salta y solo se imprime "Fin del programa". El programa tomó un camino distinto según el dato. Eso es una decisión.

Pero la vida real rara vez es "haz esto o no hagas nada"; normalmente es "haz esto o haz esto otro". Para eso está el else:

```
let edad = 15;

if (edad >= 18) {
  console.log("Puedes entrar al concierto");
} else {
  console.log("Lo sentimos, necesitas un adulto");
}
```

Uno de los dos bloques se ejecuta siempre, nunca ambos, nunca ninguno. Es una bifurcación completa: el programa elige un camino y sigue.

¿Y cuando hay más de dos caminos? Para eso está el else if, que encadena condiciones en orden. El programa las evalúa de arriba a abajo y ejecuta el bloque de la PRIMERA que sea true; las demás ni se revisan.

```

let calificacion = 85;

if (calificacion >= 90) {
  console.log("Excelente");
} else if (calificacion >= 80) {
  console.log("Muy bien");
} else if (calificacion >= 70) {
  console.log("Bien");
} else if (calificacion >= 60) {
  console.log("Suficiente");
} else {
  console.log("Reprobado");
}

```

Con calificacion en 85, el programa revisa: ¿>= 90? No. ¿>= 80? Sí: imprime "Muy bien" y sale de toda la cadena. Fíjate en algo crucial: el orden importa. Si pusieras la condición >= 60 primero, una calificación de 95 caería ahí y nunca llegaría al "Excelente". Cuando las condiciones son rangos encadenados, ordénalas de la más exigente a la menos exigente (o al revés, pero de forma consistente y consciente).

Los bloques {} definen qué pertenece a cada rama. Si el bloque tiene una sola línea, JavaScript te permite omitir las llaves, pero no lo hagas: es una de esas libertades que generan bugs. El clásico es agregar después una segunda línea creyendo que está dentro del if, cuando en realidad quedó fuera:

```

// PELIGRO: sin llaves, solo la primera línea pertenece al if
if (edad >= 18)
  console.log("Puedes entrar");
  console.log("Te damos la bienvenida"); // ¡esto SIEMPRE se ejecuta!

// SEGURO: con llaves, la intención es clara
if (edad >= 18) {
  console.log("Puedes entrar");
  console.log("Te damos la bienvenida");
}

```

Bueno, ese console con espacio en medio fue un error de dedo ilustrativo: ni lo intentes, es SyntaxError. El punto queda: usa siempre las llaves. Tu yo del futuro, agregando líneas a las 2 de la mañana, te lo va a agradecer.

Anidamiento: decisiones dentro de decisiones

Un if puede vivir dentro de otro if. Eso se llama anidamiento y sirve para decisiones en dos pasos: primero verificas algo general y luego, dentro, afinas el detalle.

```
let edad = 22;
let tieneBoleto = true;

if (edad >= 18) {
  console.log("Edad verificada");

  if (tieneBoleto) {
    console.log("Adelante, disfruta el concierto");
  } else {
    console.log("Necesitas comprar un boleto primero");
  }
} else {
  console.log("Necesitas venir con un adulto");
}
```

El if interno solo se evalúa si el externo fue true. Es como un filtro en dos etapas: primero la edad, luego el boleto. El anidamiento es perfectamente válido y a veces es la forma más clara de expresar una lógica. Pero tiene un límite: cuando anidas tres, cuatro o más niveles, el código se convierte en una pirámide difícil de leer (los programadores le dicen "callback hell" o "pirámide de la perdición" en otros contextos). La alternativa elegante es combinar condiciones con operadores lógicos (&&, ||), que es justo el tema de la siguiente clase del curso y que veremos brevemente aquí como adelanto.

Truthy y falsy: cuando no todo es booleano

El if no exige un booleano puro: acepta cualquier valor y lo convierte a booleano con sus propias reglas. Los valores que se convierten a false se llaman falsy, y son exactamente seis: false, 0, "" (string vacío), null, undefined y NaN. Todo lo demás es truthy: se convierte a true.

```

if ("hola") {
  console.log("Los strings no vacíos son truthy");
}

if (0) {
  console.log("Esto nunca se imprime: 0 es falsy");
} else {
  console.log("El cero es falsy");
}

let nombre = "";
if (!nombre) {
  console.log("Nombre vacío: ¡pide el dato al usuario!");
}

```

Este último patrón es ubicuo en código real: `if (!nombre)` significa "si nombre está vacío o no existe", y sirve para validar que el usuario sí escribió algo. El operador `!` niega el booleano: convierte `truthy` en `false` y `falsy` en `true`. Lo veremos a fondo con los operadores lógicos, pero ya puedes ir usando este patrón de validación.

Una advertencia: no abuses de `truthy/falsy` para comparar cosas distintas. `if (edad)` no es lo mismo que `if (edad > 0)`: el primero también rechaza `edad = 0`, que podría ser un valor válido en tu lógica (un recién nacido tiene 0 años). Sé explícito en tus condiciones: escribe exactamente la pregunta que quieres hacer. La claridad le gana siempre a la astucia.

Errores comunes con condicionales

Error clásico 1: usar `=` (asignación) en lugar de `===` (comparación) dentro del `if`. Es el error más famoso de la programación y le ha costado millones a la industria.

```

let rol = "invitado";

if (rol = "admin") { // ¡ASIGNACIÓN, no comparación!
  console.log("Bienvenido, admin");
}

```


Con un solo `=`, no estás preguntando si `rol` es `"admin"`: estás GUARDANDO `"admin"` en `rol`. Y como una asignación devuelve el valor asignado (`"admin"`, un string no vacío, truthy), el `if` siempre es `true`. El programa deja entrar a todo el mundo como `admin`. La solución: el hábito de escribir siempre `===`, y si tu editor o linter te avisa de una asignación dentro de un `if`, hazle caso.

Error clásico 2: condiciones que se pisan por mal orden, como vimos con las calificaciones. Siempre pregúntate: "si el valor cumple varias condiciones, ¿cuál quiero que gane?" y pon esa primero.

Error clásico 3: comparar decimales con `===` después de operaciones. Por el tema de `0.1 + 0.2` que vimos, evita `if (total === 0.3)` con decimales calculados; mejor redondea o compara con un margen.

Error clásico 4: olvidar que el `else if` solo se evalúa si los anteriores fallaron. No son condiciones independientes: son una cadena de intentos en orden. Si necesitas verificar varias cosas independientes entre sí, usa `ifs` separados, no una cadena `else if`.

Preguntas frecuentes de la unidad

¿Por qué `"5" + 5` da `"55"` pero `"5" - 5` da `0`? Porque el `+` con un string concatena (convierte el `5` a texto) mientras que el `-` solo sabe restar números (convierte el `"5"` a número). Es la regla de preferencia de cada operador, y es consistente una vez que la conoces.

¿Debo memorizar la tabla de coerción? No. Debes desarrollar el reflejo de convertir explícitamente. La tabla existe para entender código ajeno y depurar; tu código nuevo no debería depender de ella.

¿Cuándo uso `Number()` y cuándo `parseInt()`? `Number()` cuando el string debe ser completamente un número (`"42"` -> `42`, `"42px"` -> `NaN`, avisándote del problema). `parseInt()` cuando quieres extraer el número inicial de un texto (`"42px"` -> `42`). Para datos de usuario que deben ser números limpios, `Number()` es más estricto y por tanto más seguro.

¿El `else` es obligatorio? No. Un `if` solo, sin `else`, es perfectamente válido y muy común: significa "si pasa esto haz algo, si no, no hagas nada especial". El `else` aparece cuando el "si no" también requiere acción.

¿Puedo meter un `if` dentro de otro indefinidamente? Puedes, pero no debes pasar de dos o tres niveles. Más allá de eso, el código se vuelve difícil de seguir. La solución elegante es combinar condiciones con `&&` y `||`, o extraer la lógica a funciones (próximas unidades).

¿Qué tan grande puede ser una cadena else if? Todo lo que necesites, pero si tienes más de cuatro o cinco ramas haciendo cosas muy distintas, quizá el problema pide otra estructura (un switch, un objeto como mapa, o funciones separadas). No te preocupes por eso ahora; con else if bien ordenado llegas lejos.

Ejemplos paso a paso

Vamos a convertir la teoría en práctica con ejemplos completos. Como siempre: cópialos, ejecútalos, modifícalos y rómpelos a propósito. La consola del navegador (F12) es tu laboratorio.

Ejemplo 1: el detective de tipos. Este programa recibe valores de distintos tipos y los clasifica. Es el ejercicio perfecto para grabarte typeof y los cinco tipos de la unidad.

```
// Declaramos una variable de cada tipo
let nombre = "Lucía";           // string
let edad = 28;                  // number
let esEstudiante = true;        // boolean
let direccion;                  // undefined (declarada sin valor)
let apodo = null;               // null (vacío intencional)

// Preguntamos el tipo de cada una
console.log("nombre es de tipo: " + typeof nombre);           // string
console.log("edad es de tipo: " + typeof edad);                // number
console.log("esEstudiante es de tipo: " + typeof esEstudiante); // boolean
console.log("direccion es de tipo: " + typeof direccion);      // undefined
console.log("apodo es de tipo: " + typeof apodo);              // object (el bug histórico)

// Experimento: cambia el valor de una variable y vuelve a preguntar
edad = "veintiocho"; // ahora edad guarda un string
console.log("edad ahora es de tipo: " + typeof edad); // string
console.log("La variable no está atada al tipo: el VALOR manda.");
```

Observa la última parte: la variable edad pasó de number a string sin que JavaScript proteste. Eso es el tipado dinámico en acción. Útil y peligroso a la vez: útil porque te da flexibilidad, peligroso porque un dato que esperabas numérico puede volverse texto en cualquier momento (típico cuando viene de un formulario). Por eso el hábito de verificar tipos con typeof cuando algo se comporta raro.

Ejemplo 2: la trampa del formulario y su rescate. Este ejemplo simula lo que pasa todos los días en el desarrollo web: los datos que escribe el usuario en un formulario llegan como strings, y si no los conviertes, los cálculos salen mal. Vamos a provocar el bug y luego a corregirlo, para que veas la diferencia con tus propios ojos.

```
// Simulación: el usuario escribió su edad y el precio en un formulario.
// Los formularios SIEMPRE entregan strings.
let edadTexto = "25";
let precioTexto = "100";

// INTENTO 1: operar sin convertir (el bug clásico)
let sumaMal = edadTexto + 5;
console.log("Sin convertir: " + sumaMal); // "255" ¡concatena en lugar de sumar!

// INTENTO 2: convertir explícitamente ANTES de operar (la solución)
let edad = Number(edadTexto);
let precio = Number(precioTexto);
let sumaBien = edad + 5;
console.log("Convirtiendo: " + sumaBien); // 30, correcto

// Un cálculo completo de tienda con datos convertidos
const iva = 0.16;
let subtotal = precio * 2; // 2 artículos
let total = subtotal + subtotal * iva;
console.log("Total de la compra: $" + total); // 232
```

La moraleja en una frase: convierte en la puerta de entrada. En cuanto el dato llega (del formulario, del usuario, de donde sea), conviértelo al tipo que necesitas y trabaja con la versión convertida de ahí en adelante. Los bugs de tipos casi siempre nacen de operar con el dato crudo.

Ejemplo 3: el módulo en la vida real. Tres mini-programas que muestran por qué es tan querido: par o impar, repartos y turnos.

```
// 1) ¿Par o impar? El truco más famoso del módulo.
let numero = 17;
if (numero % 2 === 0) {
  console.log(numero + " es par");
} else {
  console.log(numero + " es impar"); // se imprime este
}

// 2) Repartir pizzas: 8 rebanadas entre 3 amigos, ¿sobran?
let rebanadas = 8;
let amigos = 3;
let porPersona = (rebanadas - (rebanadas % amigos)) / amigos;
let sobran = rebanadas % amigos;
console.log("A cada uno le tocan " + porPersona + " y sobran " + sobran);

// 3) Turnos rotativos: 4 trabajadores, día 10 del mes, ¿a quién le toca?
let trabajadores = 4;
let dia = 10;
let turno = dia % trabajadores; // 2
console.log("El día " + dia + " le toca al trabajador número " + turno);
// Con el módulo, los turnos rotan solos sin contadores complicados.
```

Fíjate en el ejemplo 3: sin módulo, calcular turnos rotativos requeriría llevar la cuenta manualmente y reiniciarla cada 4 días. Con `dia % 4`, el ciclo se repite solo para siempre: 1, 2, 3, 0, 1, 2, 3, 0... El módulo es la herramienta natural para todo lo cíclico: turnos, calendarios, paginación, repartos. Cada vez que pienses "esto se repite cada N", piensa en %.

Ejemplo 4: `===` contra `==`, la demostración definitiva. Ejecuta este bloque y observa con tus propios ojos por qué la igualdad débil es una trampa.

```

console.log("=== estricta ===");
console.log("5" === 5);           // false: tipos distintos
console.log(0 === false);        // false: number contra boolean
console.log("" === false);       // false
console.log(null === undefined); // false: son cosas distintas

console.log("== débil ==");
console.log("5" == 5);           // true: convierte y compara
console.log(0 == false);        // true: ¡peligroso!
console.log("" == false);       // true: ¡peligroso!
console.log(null == undefined); // true: ¡confuso!

// La forma profesional: convierte TÚ y compara estricto
let entradaUsuario = "5"; // viene de un formulario como string
let numeroEsperado = 5;
console.log(Number(entradaUsuario) === numeroEsperado); // true, sin magia

```

El último `console.log` es el patrón que debes llevarte: si los tipos no coinciden, convierte explícitamente uno de los lados y usa `===`. El código queda claro, predecible y libre de sorpresas. Cuando veas `==` en código ajeno (tutoriales viejos, respuestas de foros), tradúcelo mentalmente con sospecha: pregúntate qué conversiones está haciendo por debajo.

Ejemplo 5: el semáforo de calificaciones. Un programa completo que usa la cadena `if/else` `if/else` con rangos. Prueba cambiando el valor de calificación y observa cómo el programa toma caminos distintos.

```
let calificacion = 73;
let mensaje;

if (calificacion >= 90) {
  mensaje = "Excelente, sigue así";
} else if (calificacion >= 80) {
  mensaje = "Muy bien";
} else if (calificacion >= 70) {
  mensaje = "Bien, pero puedes mejorar";
} else if (calificacion >= 60) {
  mensaje = "Apenas pasaste, a estudiar más";
} else {
  mensaje = "Reprobado, a repetir el tema";
}

console.log("Tu calificación: " + calificacion);
console.log(mensaje);

// Experimento: cambia calificacion a 95, 82, 64, 40 y ejecuta de nuevo.
// Observa que SIEMPRE se ejecuta exactamente una rama.
```

Nota el patrón declarar-afuera-asignar-adentro que aprendiste en la unidad 1: mensaje se declara con let fuera de la cadena y cada rama le asigna su texto. Es el patrón estándar para "la condición decide el valor". También nota el orden de las condiciones: de la más exigente (≥ 90) a la menos (≥ 60). Si invirtieras el orden, todas las calificaciones altas caerían en la primera condición floja. El orden no es decoración: es lógica.

Ejemplo 6: decisiones anidadas, el control de acceso al club. Dos verificaciones en secuencia: primero la edad, luego la membresía. Solo quien pase ambas entra.

```
let edad = 25;
let tieneMembresia = true;

if (edad >= 18) {
  console.log("Edad OK, revisando membresía...");

  if (tieneMembresia) {
    console.log("Acceso total: pista de baile y terraza");
  } else {
    console.log("Puedes entrar, pero solo a la zona general");
  }
} else {
  console.log("Acceso denegado: ven con un adulto");
}

// Prueba las 4 combinaciones: (25, true), (25, false), (16, true), (16, false)
// y verifica que cada una toma el camino correcto.
```

Este ejemplo te entrena a leer código anidado: sigue la indentación como un mapa. Cada nivel de indentación es un nivel de profundidad en la decisión. Y el ejercicio de probar las cuatro combinaciones no es opcional: es exactamente lo que hace un programador profesional al probar lógica condicional, cubrir todos los caminos posibles. Acostúmbrate a preguntarte "¿qué combinaciones existen?" y a probarlas todas.

Ejemplo 7: validando con `truthy/falsy`. Un mini-sistema que verifica que el usuario llenó su nombre antes de continuar. Este patrón lo vas a ver en todos los formularios del mundo.

```

let nombreUsuario = ""; // simula que el usuario no escribió nada

if (!nombreUsuario) {
  console.log("Error: debes escribir tu nombre");
} else {
  console.log("Bienvenido, " + nombreUsuario);
}

// Ahora simula que sí escribió:
nombreUsuario = "Sofía";

if (!nombreUsuario) {
  console.log("Error: debes escribir tu nombre");
} else {
  console.log("Bienvenido, " + nombreUsuario); // se imprime este
}

```

El truco está en `!nombreUsuario`: el string vacío es falsy, así que `!""` es true y entra al error. Cuando el string tiene contenido es truthy, `! "Sofía"` es false y va al else. Es conciso, legible y es el estándar de la industria para validar campos. Pero recuerda la advertencia de la teoría: úsalo para "¿existe y tiene contenido?", no para preguntas numéricas precisas donde el 0 sea un valor válido.

Ejemplo 8: mini-reto guiado, la pizzería. Vamos a juntar tipos, conversiones, aritmética, módulo y condicionales en un solo programa: una pizzería que calcula el pedido, detecta si hay descuento y reparte las rebanadas. Lee el enunciado, intenta resolverlo tú, y luego compara con la solución.

Enunciado: cada pizza cuesta \$149 y trae 8 rebanadas. Un cliente pide N pizzas (simula que N llega como string de un formulario: "3"). Calcula el subtotal; si el subtotal supera \$400, aplica 10% de descuento; muestra el total final. Además calcula cuántas rebanadas son en total y cuántas sobran si se reparten entre 5 personas.


```
// Solución del mini-reto
let pizzasTexto = "3"; // dato crudo del "formulario": string
let pizzas = Number(pizzasTexto); // conversión en la puerta de entrada

const precioPizza = 149;
const rebanadasPorPizza = 8;
const personas = 5;

let subtotal = pizzas * precioPizza;
let descuento = 0;

if (subtotal > 400) {
  descuento = subtotal * 0.10;
  console.log("¡Descuento del 10% aplicado!");
}

let total = subtotal - descuento;

let rebanadasTotales = pizzas * rebanadasPorPizza;
let porPersona = (rebanadasTotales - (rebanadasTotales % personas)) / personas;
let sobran = rebanadasTotales % personas;

console.log("Pizzas: " + pizzas);
console.log("Subtotal: $" + subtotal);
console.log("Descuento: $" + descuento);
console.log("Total a pagar: $" + total);
console.log("Rebanadas: " + rebanadasTotales + ", a cada persona le tocan " +
porPersona + " y sobran " + sobran);
```

Revisa todo lo que este programita usa: conversión explícita del dato de entrada, const para los datos fijos del negocio, let para los calculados, un if que decide el descuento con el operador >, aritmética para los montos, módulo para el reparto. Es un programa de verdad, del tipo que corre en negocios reales todos los días. Si lo entiendes línea por línea sin ayuda, esta unidad ya es tuya.

¿Sabías que...? El operador% tiene un primo menos conocido en otros lenguajes, pero en JavaScript el% con números negativos da resultados que sorprenden: -7% 3 da -1, no 2. El signo del resultado sigue al dividendo. No es algo que necesites todos los días, pero si alguna vez trabajas con coordenadas o desplazamientos que pueden ser negativos, este detalle te va a ahorrar una tarde de depuración.

Práctica

Los mismos de siempre: intenta cada ejercicio por tu cuenta antes de mirar la solución. Si te atorras más de quince minutos, estudia la solución, ciérrala y reescríbela tú. El músculo se entrena con esfuerzo, no con lectura.

Ejercicio 1: clasificador de tipos. Enunciado: declara cinco variables, una de cada tipo (string, number, boolean, undefined, null). Luego escribe un programa que use typeof para imprimir el tipo de cada una en un mensaje legible como "La variable X es de tipo Y". Incluye el caso curioso de null y explícalo en un comentario.

```
// Solución
let producto = "Laptop";      // string
let precio = 12999;           // number
let enOferta = true;          // boolean
let descripcion;              // undefined
let garantiaExtendida = null; // null

console.log("producto es de tipo: " + typeof producto);
console.log("precio es de tipo: " + typeof precio);
console.log("enOferta es de tipo: " + typeof enOferta);
console.log("descripcion es de tipo: " + typeof descripcion);
// null devuelve "object" por un bug histórico del lenguaje que nunca se corrigió
console.log("garantiaExtendida es de tipo: " + typeof garantiaExtendida);
```

Explicación paso a paso: declaramos una variable por tipo usando los valores más representativos de cada uno. descripcion se declara sin valor para que valga undefined de forma natural. garantiaExtendida se inicializa en null a propósito para representar "aún no se decide la garantía". Luego interrogamos cada variable con typeof y mostramos el resultado con mensajes descriptivos. El comentario sobre null documenta la rareza histórica para quien lea el código después. Este ejercicio graba los cinco tipos en la memoria muscular: después de hacerlo un par de veces, typeof te saldrá solo cuando depures.

Ejercicio 2: rescatando datos del formulario. Enunciado: simula que de un formulario llegan estos strings: cantidadTexto = "4" y precioTexto = "59.99". Convierte ambos a número, calcula el subtotal (cantidad × precio), el IVA del 16% y el total. Muestra los tres montos. Además muestra qué habría pasado si no convertías (el bug de concatenación).

```
// Solución
let cantidadTexto = "4";
let precioTexto = "59.99";

// Conversión explícita en la puerta de entrada
let cantidad = Number(cantidadTexto);
let precio = Number(precioTexto);

const iva = 0.16;
let subtotal = cantidad * precio;
let montoIva = subtotal * iva;
let total = subtotal + montoIva;

console.log("Subtotal: $" + subtotal);    // 239.96
console.log("IVA: $" + montoIva);        // 38.3936
console.log("Total: $" + total);          // 278.3536

// El bug que evitamos:
console.log("Sin convertir, cantidad + 1 sería: " + (cantidadTexto + 1)); // "41"
```

Explicación paso a paso: primero convertimos los strings a número con `Number()`, porque los datos de formularios siempre llegan como texto. Usamos `Number()` y no `parseInt()` porque el precio tiene decimales y `parseInt` los cortaría (59 en lugar de 59.99, un error de dinero real). Calculamos subtotal multiplicando, IVA aplicando el 16% y total sumando. La última línea demuestra el bug evitado: `cantidadTexto + 1` concatena ("41") en lugar de sumar (5). Nota los decimales largos del IVA (38.3936): en un sistema real redondearías a dos decimales para mostrar moneda, con `total.toFixed(2)` por ejemplo, tema que verás más adelante. Lo importante aquí es el flujo: convertir primero, operar después, siempre.

Ejercicio 3: par o impar con validación. Enunciado: escribe un programa que reciba un número (decláralo tú con `let`) e imprima si es par o impar usando el módulo. Agrega una validación: si el valor no es un número válido (NaN), debe imprimir "Eso no es un número válido" en lugar de intentar el cálculo. Prueba con 10, 7 y con `Number("hola")`.

```
// Solución
let numero = 10;

if (Number.isNaN(numero)) {
  console.log("Eso no es un número válido");
} else if (numero % 2 === 0) {
  console.log(numero + " es par");
} else {
  console.log(numero + " es impar");
}

// Pruebas:
// numero = 10 -> "10 es par"
// numero = 7 -> "7 es impar"
// numero = Number("hola") -> "Eso no es un número válido"
```

Explicación paso a paso: lo primero que hacemos es validar la entrada con `Number.isNaN()`, que detecta el valor especial `NaN`. Esta validación va PRIMERO en la cadena, porque no tiene sentido calcular par/impar sobre algo que no es número. Si pasa la validación, usamos `numero % 2 === 0` para detectar pares: todo par dividido entre 2 deja residuo 0. El `else` captura los impares. El orden de la cadena importa: validación primero, lógica después. Este patrón (validar entrada -> procesar -> responder) es el esqueleto de casi toda función que procesa datos de usuario, y lo vas a repetir cientos de veces en tu carrera.

Ejercicio 4: el semáforo de la tienda. Enunciado: una tienda tiene esta política: si el cliente compra más de \$1000, obtiene 15% de descuento; si compra más de \$500 (pero no más de \$1000), obtiene 10%; si no, no hay descuento. Escribe el programa con una cadena `if/else if/else` que calcule e imprima el descuento y el total final. Prueba con 1200, 750 y 300.

```
// Solución
let compra = 1200;
let porcentajeDescuento = 0;

if (compra > 1000) {
  porcentajeDescuento = 15;
} else if (compra > 500) {
  porcentajeDescuento = 10;
} else {
  porcentajeDescuento = 0;
}

let descuento = compra * (porcentajeDescuento / 100);
let total = compra - descuento;

console.log("Compra: $" + compra);
console.log("Descuento (" + porcentajeDescuento + "%): $" + descuento);
console.log("Total a pagar: $" + total);

// Pruebas:
// compra = 1200 -> 15% -> descuento 180 -> total 1020
// compra = 750 -> 10% -> descuento 75 -> total 675
// compra = 300 -> 0% -> descuento 0 -> total 300
```

Explicación paso a paso: declaramos la compra y una variable para el porcentaje, inicializada en 0. La cadena evalúa de arriba a abajo: primero pregunta si supera 1000 (el rango más exigente va primero, como aprendimos). Si no, pregunta si supera 500. Si tampoco, el else asigna 0. Una vez decidido el porcentaje, calculamos el descuento en dinero y el total. Fíjate que la decisión (qué porcentaje) está separada del cálculo (cuánto dinero): primero decides, luego calculas. Esa separación hace el código más claro y más fácil de modificar: si mañana la tienda cambia los porcentajes, solo tocas la cadena de ifs. Prueba los tres valores y verifica que cada uno cae en la rama correcta; probar todos los caminos es el hábito profesional.

Ejercicio 5: acceso al evento (anidamiento). Enunciado: un evento pide dos cosas: ser mayor de 18 años y tener boleto. Escribe el programa con ifs anidados que imprima exactamente uno de estos mensajes: "Bienvenido al evento" (cumple ambas), "Necesitas un boleto" (edad ok, sin boleto), "Ven con un adulto" (menor de edad). Prueba las tres combinaciones relevantes.

```
// Solución
let edad = 20;
let tieneBoleto = false;

if (edad >= 18) {
  if (tieneBoleto) {
    console.log("Bienvenido al evento");
  } else {
    console.log("Necesitas un boleto");
  }
} else {
  console.log("Ven con un adulto");
}

// Pruebas:
// (20, true) -> "Bienvenido al evento"
// (20, false) -> "Necesitas un boleto"
// (16, true) -> "Ven con un adulto" (el boleto ni se revisa)
```

Explicación paso a paso: el if externo filtra por edad: si no cumple, el programa ni siquiera pregunta por el boleto y va directo al mensaje de adulto. Solo si la edad pasa, entramos al bloque y el if interno decide según el boleto. Este diseño refleja la lógica del mundo real: no tiene sentido pedir el boleto a quien no puede entrar por edad. Observa que con (16, true) el resultado es "Ven con un adulto" aunque tenga boleto: el anidamiento expresa prioridad, la edad manda sobre el boleto. Cuando pruebes, cambia los valores y recorre mentalmente el código línea por línea siguiendo la indentación: ese recorrido mental es la habilidad de "ejecutar código en tu cabeza" que distingue a quien programa de quien solo lee.

Ejercicio 6: encuentra los tres bugs. Enunciado: este programa debería saludar según la hora (0-23) pero tiene tres errores: uno de tipos, uno de comparación y uno de lógica/orden. Encuéntralos, explícalos y escribe la versión corregida.

```

let hora = "9";

if (hora = 12) {
  console.log("Es mediodía");
} else if (hora < 12) {
  console.log("Buenos días");
} else if (hora < 19) {
  console.log("Buenas tardes");
} else {
  console.log("Buenas noches");
}

```

```

// Solución corregida
let hora = 9; // Bug 1 (tipos): era string "9"; las comparaciones numéricas con
strings funcionan por coerción, pero es frágil. Mejor número desde el inicio.

if (hora === 12) { // Bug 2 (comparación): era = (asignación), que siempre daba
truthy y saludaba "mediodía" a toda hora. Con === comparamos de verdad.
  console.log("Es mediodía");
} else if (hora < 12) {
  console.log("Buenos días");
} else if (hora < 19) {
  console.log("Buenas tardes");
} else {
  console.log("Buenas noches");
}

// Bug 3 (orden): la rama de mediodía debe ir ANTES que hora < 12,
// porque las 12 también cumplen < 12... espera, ¿12 < 12? No.
// Revisemos: con hora = 12, la primera condición hora === 12 es true e imprime
mediodía. Correcto.
// El verdadero bug de orden habría sido poner hora < 19 antes que hora < 12.
// En este código el orden está bien; el bug era solo la asignación. ¡Verificar el
orden también es parte del ejercicio!

```

Explicación paso a paso: el bug 1 es de tipos: hora como string "9" funciona en las comparaciones < por coerción, pero es una bomba de tiempo; si el dato viene de un formulario, conviértelo a número al recibirlo. El bug 2 es el clásico mortal: un solo = asigna en lugar de comparar, la asignación devuelve "9" (truthy) y el programa siempre imprime "Es mediodía" sin importar la hora. Con === el programa pregunta de verdad. El bug 3 nos sirvió para practicar la revisión del orden: en este caso el orden estaba bien, pero el hábito de preguntarte "¿algún valor puede caer en la rama equivocada?" es lo que te salva en cadenas

más complejas. Nota cómo el ejercicio te entrenó a depurar en tres dimensiones: tipos, operadores y lógica. Esa es la trinidad de la depuración.

Ejercicio 7: mini-reto integrador, el cajero de la cafetería. Enunciado: una cafetería vende café a \$35 y pan dulce a \$20. El cliente pide cantidades que llegan como strings de un formulario: `cafeTexto = "3"`, `panTexto = "5"`. Convierte los datos, calcula el subtotal, aplica esta política: si el subtotal supera \$200, descuento del 10%; si no, sin descuento. Calcula el total. Además, si el total supera \$180, el cliente se gana un café gratis: imprímelo como mensaje extra usando un `if` separado. Muestra un ticket completo.

```
// Solución
let cafeTexto = "3";
let panTexto = "5";

// Conversión en la puerta de entrada
let cafeCantidad = Number(cafeTexto);
let panCantidad = Number(panTexto);

const precioCafe = 35;
const precioPan = 20;

let subtotal = cafeCantidad * precioCafe + panCantidad * precioPan;
let descuento = 0;

if (subtotal > 200) {
  descuento = subtotal * 0.10;
  console.log("¡Descuento del 10% aplicado!");
}

let total = subtotal - descuento;

console.log("--- TICKET CAFETERÍA ---");
console.log("Cafés: " + cafeCantidad + " x $" + precioCafe);
console.log("Panes: " + panCantidad + " x $" + precioPan);
console.log("Subtotal: $" + subtotal);
console.log("Descuento: $" + descuento);
console.log("Total: $" + total);

// Promoción independiente: if separado, no else if
if (total > 180) {
  console.log("¡Felicidades! Te ganaste un café gratis.");
}
```


Explicación paso a paso: convertimos las cantidades a número nada más recibirlas, porque vienen como strings del formulario. Los precios van en const por ser datos fijos del negocio. Calculamos el subtotal sumando ambos productos. El primer if decide el descuento con la política de los \$200; observa que el mensaje de descuento está dentro del if para que solo aparezca cuando aplica. Calculamos el total restando el descuento. Mostramos el ticket completo con etiquetas claras. El segundo if es independiente del primero (no es else if) porque la promoción del café gratis no es alternativa al descuento: son dos decisiones separadas que pueden ocurrir juntas. Esa distinción (cadena else if para caminos excluyentes, ifs separados para decisiones independientes) es uno de los conceptos más importantes de la unidad. Con $\text{subtotal} = 3 \times 35 + 5 \times 20 = 205$, hay descuento de 20.5, total 184.5, y como supera 180, también hay café gratis. Verifica la aritmética a mano: entrenar el cálculo mental con tus programas te hace detectar resultados absurdos al instante.

Cuidado con...: en ejercicios de dinero es tentador comparar decimales con `===`, como `if (total === 184.5)`. Por el tema de la representación binaria de decimales, evita igualdades exactas con resultados de operaciones decimales; prefiere `>`, `<` o redondea antes de comparar. Y recuerda: muestra el dinero con dos decimales usando `total.toFixed(2)` cuando lo presentes al usuario; 184.5 se ve mejor como 184.50 en un ticket.

Galería de errores típicos (y cómo leerlos)

Vamos a entrenar tu ojo de depurador con los mensajes de error más comunes de esta unidad. Cada uno incluye el mensaje, la traducción a humano y la causa típica.

Error 1: "Uncaught ReferenceError: X is not defined". Ya lo conoces de la unidad 1: usaste un nombre que no existe en ese scope. En esta unidad aparece mucho con variables mal escritas en condiciones: `if (edaad >= 18)` con una a de más. El mensaje te dice exactamente qué nombre falló; búscalo letra por letra.

Error 2: "NaN" como resultado silencioso. NaN no lanza error, y eso lo hace más peligroso: tu programa sigue corriendo con un "no es un número" metido en los cálculos, y todo lo que toque se contagia (cualquier operación con NaN da NaN). Si ves NaN en tu consola, el diagnóstico es: en algún punto convertiste a número algo que no era número. Rastrea hacia atrás con `typeof` hasta encontrar el string culpable. `Number("hola")`, `Number(undefined)` y `parseInt("abc")` son las fábricas habituales de NaN.

```
let precio = Number("gratis"); // NaN: "gratis" no es número
let total = precio * 2;         // NaN: el contagio
let iva = total * 0.16;         // NaN: sigue contagiando
console.log(total);             // NaN: el programa "funcionó" pero todo está mal
```

La defensa profesional: valida las conversiones justo después de hacerlas. Si Number() te devuelve NaN, detén el proceso y avisa, no sigas calculando con un dato podrido.

Error 3: el "305" fantasma. No es un mensaje de error, es un resultado incorrecto silencioso: esperabas 35 y obtuviste "305". Ya sabes la causa: concatenación por mezclar string con number en un +. El diagnóstico es typeof en cada operando. La cura es convertir antes de operar. Este bug es tan común que debería tener su propio día festivo entre programadores.

Error 4: el if que siempre entra (o nunca entra). Si tu if siempre ejecuta el bloque aunque la condición "debería" ser falsa, sospecha de un = solitario en lugar de ==. Si nunca entra aunque "debería", sospecha de tipos: quizá comparas "18" (string) con == 18 (number) y la igualdad estricta dice false con toda la razón. En ambos casos, un console.log de la condición antes del if te muestra la verdad:

```
let edad = "18";
console.log(edad == 18); // false: ¡ahí está! string contra number
console.log(Number(edad) == 18); // true: convertido, funciona
```

Ese hábito (imprimir la condición antes del if cuando algo no cuadra) te va a salvar cientos de veces. Los programadores veteranos no adivinan: imprimen, observan y corrigen.

Error 5: "Uncaught SyntaxError: Unexpected token" en condicionales. Casi siempre es una llave sin cerrar o un paréntesis perdido en la condición. Cuando el mensaje apunte a una línea, mira también la anterior: muchas veces el error real está una línea arriba, donde olvidaste cerrar algo.

¿Sabías que...? Existe una técnica de depuración llamada "depuración con patito de goma" (rubber duck debugging): le explicas tu código en voz alta, línea por línea, a un patito de goma (o a cualquier objeto). El acto de verbalizar te obliga a leer lo que realmente escribiste en lugar de lo que crees que escribiste, y el bug suele revelarse solo a mitad de la explicación. Funciona de verdad, y los programadores más serios del mundo la usan sin vergüenza. Si un día te atorras una hora con un if, prueba con el patito.

Cuidado con...: el error más caro de esta unidad es confiar en la coerción "porque funciona". Funciona hoy, con estos datos. Mañana llega un dato distinto (un string vacío, un null, un

undefined) y la coerción decide otra cosa. El código profesional no depende de la suerte de los datos: convierte explícitamente y compara con ===. Es un poco más de escritura a cambio de mucha menos depuración.

Mini-retos extra

Reto A: el adivinador de tipos. Sin ejecutar, predice el resultado y el tipo de cada expresión. Luego verifícalo en consola con typeof.

```
"10" + 5      // ¿"105" string o 15 number?
"10" - 5      // ¿?
"10" * "2"    // ¿?
10 + true     // ¿? (pista: true se convierte a 1)
"hola" - 3    // ¿? (pista: NaN)
```

Respuestas: "105" (string, el + concatena), 5 (number, el - convierte), 20 (number), 11 (number), NaN (number, sí, NaN es de tipo number). Si acertaste las cinco incluyendo la última rareza, la coerción ya no tiene secretos para ti.

Reto B: el guardia del bar. Escribe un programa que reciba edad (como string, simulando formulario: "21") y decida: si la edad convertida es válida y >= 18, "Adelante"; si es válida pero < 18, "Prohibido menores"; si no es un número válido, "Edad inválida". Debe manejar los tres caminos con validación primero. Este reto combina conversión, validación con isNaN y cadena condicional: el flujo profesional completo en miniatura.

Reto C: fizzbuzz simplificado. Para los números del 1 al 15 (escríbelos a mano, uno por uno, aún no conoces los ciclos), imprime "Fizz" si es múltiplo de 3, "Buzz" si es múltiplo de 5, "FizzBuzz" si es múltiplo de ambos, y el número si no es múltiplo de ninguno. Usa el módulo y una cadena if/else if/else. Pista: verifica primero el caso de ambos múltiplos, porque si verificas el de 3 primero, el 15 caería ahí y nunca llegaría al combinado. Este es el ejercicio de entrevistas de trabajo más famoso del mundo en su versión mini; dominarlo a mano te prepara para cuando los ciclos lo automaticen.

Caso de estudio: el sistema de envíos de una tienda

Vamos a unir toda la unidad en un caso de estudio realista: una tienda en línea calcula el costo de envío según el monto de la compra y la zona de destino. Es el tipo de lógica que corre en miles de tiendas todos los días.

Reglas del negocio: el monto llega como string de un formulario. Si el monto no es un número válido o es negativo, se rechaza el cálculo. Si la compra supera \$999, el envío es gratis. Si la zona es "local", el envío cuesta \$49; si es "nacional", \$99; si es "internacional", \$199. Cualquier otra zona es inválida. El programa debe mostrar el desglose completo o el mensaje de error correspondiente.

```
// Caso de estudio: calculadora de envíos
let montoTexto = "1250"; // dato crudo del formulario
let zona = "nacional";

let monto = Number(montoTexto);
let costoEnvio;
let mensaje;

// Validación primero: ¿el dato sirve?
if (Number.isNaN(monto) || monto < 0) {
  mensaje = "Error: monto inválido";
} else if (monto > 999) {
  costoEnvio = 0;
  mensaje = "¡Envío gratis por compra mayor a $999!";
} else if (zona === "local") {
  costoEnvio = 49;
  mensaje = "Envío local";
} else if (zona === "nacional") {
  costoEnvio = 99;
  mensaje = "Envío nacional";
} else if (zona === "internacional") {
  costoEnvio = 199;
  mensaje = "Envío internacional";
} else {
  mensaje = "Error: zona de envío no válida";
}

// Solo mostramos el desglose si no hubo error
if (mensaje.indexOf("Error") === -1) {
  let total = monto + costoEnvio;
  console.log("Monto de compra: $" + monto);
  console.log(mensaje + ": $" + costoEnvio);
  console.log("Total: $" + total);
} else {
  console.log(mensaje);
}
```

Analiza las decisiones de diseño: la validación del monto va primero, antes que cualquier lógica de negocio, porque no tiene sentido calcular envíos sobre un dato podrido. El envío gratis se verifica antes que las zonas, porque es la condición de mayor prioridad: un monto alto gana sin importar la zona. Las zonas se comparan con `===` porque son strings exactos. La variable `costoEnvio` se declara sin inicializar y cada rama le asigna su valor (el patrón declarar-afuera-asignar-adentro una vez más). Y al final, un `if` separado decide si mostrar el desglose o el error, usando `indexOf` para detectar la palabra "Error" en el mensaje (un adelanto de métodos de string que verás a fondo después; por ahora basta saber que busca texto dentro de texto).

Prueba el caso con distintos valores: `montoTexto = "abc"` (error de monto), `montoTexto = "-50"` (monto negativo), `zona = "luna"` (zona inválida), `montoTexto = "500"` con zona "local" (envío 49), `montoTexto = "1250"` con zona "internacional" (gratis por monto alto). Cada prueba recorre un camino distinto del programa. Esa matriz de pruebas es exactamente lo que un profesional hace antes de dar por terminado un módulo.

Este caso de estudio es también un anticipo de tu futuro: los programas reales son cadenas de validaciones y decisiones como esta, solo que más largas y con los datos viniendo de bases de datos y APIs. El esqueleto mental es el mismo: valida la entrada, decide por prioridades, calcula, presenta. Domínalo aquí en pequeño y lo reconocerás en grande.

Preguntas de repaso

1. ¿Qué significa que JavaScript sea de tipado dinámico y de tipado débil? Da un ejemplo de cada uno.
2. ¿Cuál es la diferencia entre `undefined` y `null`? Da un ejemplo donde usarías cada uno.
3. ¿Por qué `typeof null` devuelve "object" y cómo verificas correctamente si algo es `null`?
4. ¿Cuándo conviene `Number()` y cuándo `parseInt()`? ¿Qué devuelve cada uno con "42px"?
5. Explica con tus palabras por qué `"5" + 3` da "53" pero `"5" - 3` da 2.
6. ¿Para qué sirve el operador módulo? Menciona tres usos prácticos.
7. ¿Por qué debes usar `===` en lugar de `==`? Da dos ejemplos de igualdades engañosas con `==`.
8. ¿Qué hace el `else if` y en qué se diferencia de usar varios `if` separados?
9. ¿Por qué el orden de las condiciones importa en una cadena `else if` con rangos?
10. ¿Cuáles son los seis valores falsy? ¿Qué significa que un valor sea truthy?

11. ¿Qué bug causa escribir = en lugar de === dentro de un if? ¿Por qué el programa no lanza error?
12. Describe el flujo profesional para procesar datos de usuario en el orden correcto.

Glosario de la unidad

Repaso rápido de los términos que debes dominar al dedillo antes de seguir:

Tipo de dato: la clase a la que pertenece un valor (string, number, boolean, etc.). Define qué operaciones tienen sentido con ese valor.

Tipado dinámico: la variable no está atada a un tipo; puede guardar un número ahora y un texto después. El tipo lo tiene el valor, no la variable.

Tipado débil: el lenguaje convierte tipos automáticamente al mezclarlos en operaciones. Conveniente a veces, traicionero otras.

Coerción: la conversión automática de tipos que hace JavaScript por ti. La explícita la haces tú con Number() o String(); la implícita la hace el lenguaje y es la que causa sorpresas.

NaN: valor especial de tipo number que significa "esto intentó ser un número y falló". Contagia: cualquier operación con NaN devuelve NaN.

Truthy y falsy: cómo el if interpreta valores no booleanos. Solo seis valores son falsy: false, 0, "" (string vacío), null, undefined y NaN. Todo lo demás es truthy.

Igualdad estricta (===): compara valor y tipo sin conversiones. La única igualdad que debes usar.

Igualdad débil (==): compara convirtiendo tipos automáticamente. Evítala siempre; produce igualdades engañosas.

Módulo (%): operador que devuelve el residuo de una división. Clave para par/impar, repartos y ciclos.

Condicional: estructura que ejecuta código según una condición. if (una rama), else (el camino contrario), else if (varios caminos en orden).

Bloque: código entre llaves {} que pertenece a una rama del condicional. Siempre usa las llaves, aunque sea una sola línea.

Anidamiento: un condicional dentro de otro. Expresa decisiones en etapas con prioridad. No pases de dos o tres niveles.

Validación: verificar que los datos de entrada sirvan antes de procesarlos. Va primero, siempre: validar, luego decidir, luego calcular.

Resumen de la unidad

JavaScript es de tipado dinámico (las variables no están atadas a un tipo) y tipado débil (convierte tipos automáticamente al mezclarlos). Esa flexibilidad exige que tú controles los tipos en lugar de dejar que el lenguaje adivine.

Los cinco tipos primitivos de uso diario: string (texto entre comillas), number (enteros y decimales en un solo tipo), boolean (true/false, el idioma de las decisiones), undefined (existe pero sin valor asignado) y null (vacío intencional, distinto de undefined).

typeof te dice el tipo de cualquier valor y es tu primera herramienta de diagnóstico cuando un resultado sale raro. Recuerda la rareza histórica: typeof null devuelve "object".

Convierte tipos explícitamente con Number(), parseInt(), parseFloat() y String(). Haz la conversión en la puerta de entrada, en cuanto el dato llega (formularios, entradas de usuario), y opera con la versión convertida. Number() es estricto (todo o nada), parseInt/parseFloat extraen el número inicial de un texto.

La coerción es la conversión automática del lenguaje: el + prefiere strings (concatena si hay un string), los demás operadores aritméticos prefieren números. No dependas de ella en tu código; úsala solo para entender y depurar.

Operadores aritméticos: + - * /% **, más ++, -- y las asignaciones abreviadas (+=, -=, etc.). El módulo% da el residuo: sirve para par/impar, repartos y todo lo cíclico. Respeta la precedencia matemática y usa paréntesis cuando dudes.

Comparaciones: === y!== comparan valor y tipo (igualdad estricta, la única que debes usar); > < >= <= para orden. Evita siempre == y!=: su conversión automática produce igualdades falsas como 0 == false.

El if ejecuta un bloque solo si la condición es true; el else cubre el camino contrario; el else if encadena varios caminos y ejecuta solo el primero cuya condición sea true. Usa siempre las llaves {} aunque el bloque sea de una línea.

En cadenas else if con rangos, el orden importa: de la condición más exigente a la menos exigente. Los ifs anidados expresan decisiones en etapas con prioridad; no anides más de dos o tres niveles.

Truthy y falsy: solo seis valores son falsy (false, 0, "", null, undefined, NaN); todo lo demás es truthy. El patrón if (!valor) valida campos vacíos. Sé explícito en tus condiciones: escribe la pregunta exacta que quieres hacer.

Los tres bugs clásicos: = en lugar de === dentro del if (asigna en vez de comparar), condiciones en mal orden que se pisan, y operar con strings sin convertir (el "305" en lugar del 35).

El flujo profesional para procesar datos: recibir -> convertir tipos -> validar -> decidir con condicionales -> calcular -> mostrar. Si dominas ese flujo, ya programas como se programa en el mundo real.

En la unidad 3 conocerás a los primos de los condicionales: los operadores lógicos (&&, ||, !) para combinar decisiones, y los ciclos (for, while) para repetirlas. Tus programas están a punto de volverse mucho más poderosos.

Palabra final: si llegaste hasta aquí ejecutando cada ejemplo y peleando con cada ejercicio, tu forma de pensar ya cambió. Ya no ves un programa como una lista de instrucciones, sino como un agente que clasifica datos, hace cuentas y toma decisiones. Sabes que un "25" entre comillas no es lo mismo que un 25 sin comillas, sabes convertir tipos sin miedo, sabes que === es tu mejor amigo y que el = solitario en un if es tu peor enemigo. Esos reflejos valen más que cualquier sintaxis memorizada, porque son los que te van a salvar cuando el programa haga algo raro a las 2 de la mañana. En la unidad 3 les daremos superpoderes a tus decisiones: los operadores lógicos para combinarlas y los ciclos para repetirlas. Tus programas están a punto de dejar de ser lineales para siempre. Sigue practicando todos los días, aunque sea media hora: la constancia le gana al talento, y cada bug que depuras hoy es un bug que mañana resolverás en segundos.

UNIDAD 3: Comparaciones y lógica

Hasta este punto del curso ya sabes qué es JavaScript, ya sabes cómo meterlo dentro de una página web, ya sabes declarar variables con `let` y `const`, conoces los tipos de datos y sabes hacer operaciones aritméticas. En otras palabras: tu programa ya sabe guardar información y ya sabe hacer cuentas con ella. Pero te falta algo enorme, algo que es la diferencia entre una calculadora tonta y un programa de verdad: tu programa todavía no sabe tomar decisiones. No sabe preguntarse "¿esto es igual a aquello?", no sabe decir "si la contraseña es correcta, deja entrar; si no, bloquea el acceso", no sabe repetir una tarea mil veces sin que tú la escribas mil veces. Esta unidad cambia eso de una vez por todas. Aquí vas a aprender a comparar valores, a combinar condiciones con lógica booleana y a repetir código con el ciclo `for`. Cuando termines esta unidad, tus programas van a empezar a comportarse como cosas inteligentes: van a reaccionar según lo que pase, van a validar datos, van a filtrar información y van a hacer trabajo repetitivo en fracciones de segundo.

Vamos a detenernos un momento en por qué esto es tan importante, porque a veces cuando uno está aprendiendo a programar piensa que lo difícil son las matemáticas o los algoritmos complicados, y la realidad es que el noventa por ciento de lo que hace un programa real son comparaciones. Piénsalo: cuando abres tu correo, el programa compara tu usuario y tu contraseña con los que tiene guardados. Cuando compras algo en línea, el programa compara el saldo de tu tarjeta con el precio del producto. Cuando ves un video, el programa compara tu conexión con la calidad disponible y decide si te manda el video en alta o en baja definición. Cuando un banco te aprueba un crédito, hay un programa comparando tu historial, tus ingresos y tus deudas. Todo, absolutamente todo en el software, se reduce en el fondo a preguntas de sí o no: ¿es mayor?, ¿es igual?, ¿está dentro del rango?, ¿cumple las dos condiciones? Si dominas las comparaciones y la lógica, dominas el corazón de la programación. Lo demás son detalles.

Y luego está el ciclo `for`, que es tu primer superpoder de repetición. Imagina que tienes que saludar a cien personas. Sin un ciclo, tendrías que escribir cien líneas de código, una por saludo. Con un ciclo `for`, escribes tres líneas y la computadora hace las cien repeticiones sola. Ahora imagina que no son cien personas sino un millón de registros en una base de datos, o diez mil productos de una tienda, o cada píxel de una imagen. Los ciclos son lo que convierte a la computadora en una máquina incansable: ella no se aburre, no se cansa, no se distrae. Tú le enseñas una vez qué hacer y ella lo repite las veces que le pidas con la misma precisión la primera vez que la vez número un millón. Por eso los programadores decimos que no hay que repetirse a uno mismo: si vas a hacer algo más de dos veces, mejor escríbelo como un ciclo.

Esta unidad está dividida de la siguiente manera para que la puedas digerir en orden. Primero vas a ver las comparaciones: todos los operadores que sirven para preguntarle cosas a JavaScript (`==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`), con especial atención a la diferencia entre `==` y `===`, que es una de las trampas más famosas del lenguaje y la causa de una cantidad impresionante de errores en programas reales. Después vas a entrar a los operadores lógicos: `&&` (Y), `||` (O) y `!` (NO), que te permiten combinar varias comparaciones en condiciones complejas, como "si el usuario tiene más de 18 años Y tiene identificación válida, déjalo entrar". Vas a aprender las tablas de verdad, el cortocircuito (un comportamiento de JavaScript que al principio parece magia y después se vuelve una herramienta) y la precedencia de operadores, que es el orden en que JavaScript evalúa todo cuando mezclas comparaciones y lógica en una sola línea. Y para cerrar, el ciclo `for`: su sintaxis de tres partes (inicio, condición, incremento), cómo recorrer rangos de números, cómo anidar un `for` dentro de otro `for` (que suena complicado pero es la forma natural de trabajar con cosas como tablas o cuadrículas), y cómo usar `break` para salir de un ciclo antes de tiempo y `continue` para saltarte una vuelta.

Una advertencia honesta antes de empezar: esta unidad tiene fama de ser donde muchos principiantes se atorán, y no porque sea difícil, sino porque es densa. Hay varios conceptos que se parecen entre sí (`==` y `===`, `&&` y `||`, `for` y `while`) y es fácil confundirlos si los lees rápido. Mi consejo es que no corras. Lee con calma, prueba cada ejemplo en tu propia computadora o en la consola del navegador, cambia los valores, rompe el código a propósito y luego arréglalo. La programación se aprende con las manos, no con los ojos. Cada vez que veas un bloque de código en esta unidad, tu tarea es copiarlo, ejecutarlo y jugar con él. Si algo no te queda claro a la primera, relea esa parte; nadie te está cronometrando.

También quiero que sepas algo que te va a dar tranquilidad: todos, absolutamente todos los programadores, incluso los que llevan veinte años en esto, siguen consultando cómo funcionan algunas de estas cosas. La diferencia entre `==` y `===` la googlean programadores senior. El orden de precedencia lo revisan en la documentación hasta los más experimentados. No se trata de memorizar todo como si fuera un examen de la escuela; se trata de entender la idea general y saber dónde buscar el detalle cuando lo necesites. Lo que sí debes llevarte grabado de esta unidad son las ideas grandes: cómo preguntarle cosas a JavaScript, cómo combinar preguntas, y cómo repetir acciones. Los detalles de sintaxis los vas a ir dominando con la práctica.

Otro punto importante: en esta unidad vas a ver mucho código "malo" a propósito. Lo hago porque creo que aprendes más viendo un error típico y entendiendo por qué está mal que viendo solo código perfecto. Cuando te muestre un ejemplo malo, va a estar claramente marcado, y siempre viene seguido de la versión correcta y de la explicación de la diferencia.

No te asustes si al principio el código malo te parece razonable; precisamente por eso es una trampa común. La idea es que cuando te topes con ese error en tu propio código (porque te va a pasar, nos pasa a todos), lo reconozcas de inmediato.

Por último, una nota sobre el tono de este libro: te hablo de tú porque así hablamos entre colegas, y porque aprender a programar es más fácil cuando sientes que alguien te está explicando las cosas en confianza y no dictándote una cátedra. Si algo suena muy técnico, lo voy a bajar a tierra con una analogía de la vida real: semáforos, porteros de antro, recetas de cocina, listas del súper. Las analogías no son el concepto real, son un puente para llegar a él; úsalas para entender y luego quédate con el concepto técnico. ¿Listo? Vamos a enseñarle a tu programa a tomar decisiones.

Antes de entrar de lleno, hagamos un repaso rapidísimo de dónde estamos parados para que todo conecte. En la unidad anterior aprendiste los operadores aritméticos: +, -, *, /, % y los atajos como ++, --, +=. Esos operadores toman números y devuelven números. Cinco más tres te da ocho. Eso es aritmética. Lo que vas a aprender ahora es otro tipo de operadores: los que toman valores y devuelven una respuesta de sí o no, es decir, un booleano (true o false). Cinco mayor que tres te da true. Esa es la diferencia fundamental: antes calculabas, ahora preguntas. Y las respuestas a esas preguntas son las que van a controlar el flujo de tu programa: qué se ejecuta y qué se salta, cuántas veces se repite algo, cuándo se detiene.

Fíjate en algo curioso: los booleanos true y false ya los conoces de la unidad de tipos de datos, pero hasta ahora eran solo valores guardados en variables, como quien guarda una moneda en un cajón sin usarla. En esta unidad esos booleanos se vuelven los protagonistas, porque cada comparación produce un booleano y cada booleano decide el camino que toma tu programa. Es como si hasta ahora hubieras estado coleccionando llaves y ahora por fin vas a aprender a abrir puertas con ellas.

También vale la pena que tengas presente cómo se conecta esta unidad con lo que sigue. En la próxima unidad vas a ver el ciclo while y las funciones, y en las siguientes vas a ver condicionales if/else más a fondo, arrays, objetos y el DOM. Todo eso se construye sobre lo que aprendas aquí. No puedes entender un if si no entiendes las comparaciones que van dentro de sus paréntesis. No puedes entender un while si no entiendes la condición que lo mantiene vivo. No puedes filtrar un array si no sabes comparar sus elementos. Esta unidad es, literalmente, la base sobre la que se sostiene todo lo demás. Tómala en serio y el resto del curso te va a parecer mucho más fácil.

Déjame contarte una pequeña anécdota para que veas lo real que es esto. Hace años, un amigo mío programó el sistema de descuentos de una tienda en línea. La regla era: "si el

cliente compra más de 1000 pesos O es cliente frecuente, aplica 10% de descuento". Él escribió la condición al revés sin darse cuenta y durante dos días la tienda le dio descuento a todo el mundo, incluso a compras de 50 pesos. Perdieron una lana. El error no fue de matemáticas complicadas ni de un algoritmo exótico: fue un operador lógico mal usado, exactamente de los que vas a aprender en esta unidad. Cuando entiendes `&&` y `||` de verdad, ese tipo de errores dejan de pasarte. Y cuando te pasen (porque alguno se te va a escapar alguna vez), vas a saber exactamente dónde buscar.

Otra historia, esta de un ciclo `for`. Una compañera tenía que generar los reportes mensuales de su empresa: eran como doscientas filas de datos que copiaba y pegaba a mano en un formato. Tardaba toda una tarde. Un día aprendió el ciclo `for`, escribió un programita de quince líneas que hacía el trabajo en tres segundos, y su jefe pensó que había contratado a alguien más para ayudarlo. No es broma: los ciclos son así de poderosos. La diferencia entre hacer algo a mano doscientas veces y escribir un ciclo que lo haga es la diferencia entre ser el que carga los ladrillos y ser el que opera la grúa.

Con esto en mente, entremos a la materia. La unidad tiene tres grandes bloques: comparaciones, lógica y el ciclo `for`. Los tres están relacionados: las comparaciones producen las preguntas, la lógica combina las preguntas, y el ciclo `for` usa las preguntas para decidir cuándo seguir repitiendo y cuándo parar. Al final todo se junta. Empecemos.

Una última cosa antes de entrar a la explicación: quiero que hagas un trato contigo mismo para esta unidad. Cada vez que aprendas un operador nuevo, vas a probarlo en la consola del navegador con al menos tres valores distintos: uno donde esperes `true`, uno donde esperes `false` y uno raro (un texto vacío, un cero, un valor con mayúsculas). Ese hábito de probar los tres casos te va a dar una intuición que ningún libro puede darte solo con palabras. Los programadores que entienden la lógica de verdad no son los que memorizaron las tablas, son los que las comprobaron con sus propias manos cien veces. Sé de esos.

Y una última reflexión para que entres con la mentalidad correcta: programar no es memorizar operadores, es desarrollar instinto para decidir. Cada vez que dudes entre `===` y `==`, entre `&&` y `||`, entre `<` y `<=`, detente dos segundos y pregúntate qué quieres decir en español. Si lo puedes decir claro en palabras, la traducción a código sale sola. Los mejores programadores que conozco no son los que escriben más rápido, son los que se detienen a pensar esos dos segundos antes de cada condición. Sé de esos desde hoy.

Explicación detallada

Las comparaciones: hacerle preguntas a JavaScript

Comparar en programación es exactamente lo que parece: tomar dos valores y preguntar cómo se relacionan entre sí. ¿Son iguales? ¿Uno es más grande que el otro? ¿Son diferentes? JavaScript te da un conjunto de operadores para hacer estas preguntas, y cada vez que los usas, JavaScript te responde con un booleano: true (sí, es verdad) o false (no, es falso). Grábate esto porque es la idea central de toda la primera mitad de la unidad: comparar es preguntar, y la respuesta siempre es true o false.

Vamos a conocer a cada operador uno por uno, como si fueran personajes, porque cada uno tiene su personalidad y sus mañas.

El operador de igualdad estricta `===` pregunta: "¿estos dos valores son exactamente iguales, del mismo valor Y del mismo tipo?" Es el comparador más exigente y, spoiler, es el que deberías usar casi siempre. Mira:

```
5 === 5
```

Esto da true, porque ambos son el número 5. Ahora mira esto:

```
5 === "5"
```

Esto da false. El primero es un número y el segundo es un texto que dice "5". Se ven igual para un humano, pero para JavaScript son cosas distintas, como un billete de 5 pesos y una foto de un billete de 5 pesos: se parecen, pero no son lo mismo. El `===` es quisquilloso a propósito, y esa quisquillosidad te salva de errores.

Su contraparte es `!==`, la desigualdad estricta, que pregunta: "¿estos dos valores son diferentes en valor O en tipo?" Es básicamente lo contrario de `===`. Si `===` dice true, `!==` dice false, y viceversa. Ejemplo: `5 !== "5"` da true, porque aunque el valor numérico coincide, los tipos son distintos.

Ahora, los operadores de orden: `>` (mayor que), `<` (menor que), `>=` (mayor o igual que) y `<=` (menor o igual que). Estos los usaste en la escuela en matemáticas y funcionan igual aquí, con una diferencia importante: en JavaScript casi siempre los usas con números, y cuando los usas con textos, comparan por orden alfabético según el código de cada letra, lo cual da resultados que sorprenden a los principiantes. Ya llegaremos a eso en los casos borde. Por ahora, con números:

```
10 > 5
```

Da true. Diez es mayor que cinco, sin misterio.

```
10 >= 10
```

Da true, porque "mayor o igual" se cumple con la parte del "igual". Este es un error clásico de principiantes: pensar que `>=` significa "estrictamente mayor". No: el "o igual" está ahí por algo. Si quieres estrictamente mayor, usa `>` solito.

```
3 < 3
```

Da false. Tres no es menor que tres, es igual. Parece obvio cuando lo lees así, pero cuando estás programando una condición como "si la edad es menor que 18 no puede entrar" y alguien tiene exactamente 18, esa diferencia entre `<` y `<=` decide si la persona entra o no. Los límites importan muchísimo en programación; a estos errores de "uno más, uno menos" les decimos errores off-by-one y son de los más comunes del mundo.

Hasta aquí todo bien. Ahora viene el personaje problemático de la familia: el `==`, la igualdad no estricta (también llamada igualdad abstracta o "floja"). Este operador, antes de comparar, intenta convertir los valores para que sean del mismo tipo. Es como ese amigo que con tal de no pelear dice "bueno, más o menos son lo mismo". Mira:

```
5 == "5"
```

Esto da true. JavaScript convierte el texto "5" a número 5 y luego compara. ¿Ves el problema? Tú escribiste algo que se ve igual que el ejemplo anterior con `===`, pero el resultado es opuesto. Con `===` era false, con `==` es true. Esa es la trampa.

Y su compañero `!=` (desigualdad no estricta) hace lo mismo pero al revés: `5 != "5"` da false, porque después de la conversión resultan iguales.

¿Por qué existe el `==` si es tan tramposo? Por historia. JavaScript nació en 1995 y se diseñó en diez días (sí, diez días, esa es una historia real), y en esa prisa se tomaron decisiones que hoy nos parecen cuestionables. El `==` con conversión automática de tipos parecía conveniente en ese entonces: "qué práctico que el lenguaje me ayude comparando un número con un texto". Pero con los años, la industria aprendió que esa "ayuda" causa más errores de los que evita, porque esconde diferencias de tipo que normalmente indican un problema en tus datos.

Déjame mostrarte qué tan raras se pueden poner las cosas con `==`, porque esto parece chiste pero es real y pasa en programas de verdad:

```
0 == false
```

Da true. El número cero y el booleano false se consideran "iguales" con `==`.

```
"" == false
```

Da true también. Un texto vacío es "igual" a false.

```
null == undefined
```

Da true. Y `null === undefined` da false. Este es el único caso donde muchos programadores aceptan usar `==`, porque `null` y `undefined` son los dos valores que significan "vacío" o "sin valor", y a veces quieres tratarlos igual. Pero incluso aquí, la mayoría de los equipos prefieren ser explícitos.

```
"0" == false
```

Da true. Un texto con un cero es igual a false. Si esto no te parece una locura, no sé qué decirte.

La regla de oro, la que te va a ahorrar horas de debugging en tu carrera, es esta: usa siempre `===` y `!==`, y evita `==` y `!=`. Punto. Si algún día ves `==` en código ajeno, ya sabes que hay que revisarlo con lupa. Los linters (herramientas que revisan tu código automáticamente) de hecho te marcan el `==` como advertencia. No es casualidad: la industria entera llegó a la conclusión de que la igualdad estricta es más segura.

¿Sabías que...? Hay un meme famoso en la comunidad de programadores que muestra las comparaciones raras de JavaScript con `==`, como `[] == false` dando true o `[1] == "1"` dando true. Esas rarezas vienen de las reglas de conversión de tipos, que son largas y complicadas. La moraleja del meme es siempre la misma: usa `===`. Cuando domines el lenguaje a fondo podrás entender exactamente por qué `[] == false` es true, pero mientras tanto, la regla práctica te protege.

Un error típico de principiante que veo una y otra vez: confundir `=` con `==` con `===`. Son tres cosas totalmente distintas y el teclado no te ayuda porque se ven parecidas. El `=` solito es asignación: mete un valor en una variable (`let edad = 25`). El `==` y `===` son comparación:

preguntan si dos valores son iguales. El error clásico es escribir `if (edad = 18)` en lugar de `if (edad === 18)`. ¿Qué pasa? Que `edad = 18` no pregunta nada: le asigna 18 a la variable `edad`, y como una asignación "exitosa" se evalúa como el valor asignado (18, que es un valor *truthy*, concepto que veremos), el `if` siempre se ejecuta. Tu programa entonces hace cosas rarísimas: la variable `edad` cambia de valor sin que te des cuenta y la condición siempre es verdadera. Si tu `if` se comporta como si la condición siempre fuera `true`, lo primero que debes revisar es si escribiste `=` en lugar de `===`. Todos hemos cometido ese error. Todos.

Otro error común: comparar textos con mayúsculas y minúsculas sin darse cuenta. `"Ana" === "ana"` da `false`, porque la A mayúscula y la a minúscula son letras distintas para la computadora (tienen códigos diferentes). Si estás comparando nombres de usuario o respuestas de un formulario, casi siempre quieres normalizar antes: convertir todo a minúsculas con `toLowerCase()` y luego comparar. Este detalle muerde mucho en validaciones de formularios: el usuario escribe "Sí", tu código compara con "sí", no coinciden, y el usuario jura que lo escribió bien. Y tiene razón: el error es tuyo por no normalizar.

Hablemos ahora de comparar con `>=` y `<=` en situaciones reales, porque ahí es donde brillan. Piensa en un sistema de calificaciones: reprobado si es menor que 6, aprobado si es mayor o igual que 6. O en un descuento: "envío gratis en compras mayores o iguales a 499 pesos". Esos límites exactos (6, 499) son los que separan un caso del otro, y elegir entre `>` y `>=` es una decisión de negocio, no solo de código. Mi consejo: cuando defines un límite, pregúntate en voz alta "¿el valor exacto del límite entra o no entra?" y escribe un ejemplo con ese valor exacto para comprobarlo. Esa prueba de dos segundos te ahorra discusiones con clientes que juran que su compra de exactamente 499 pesos debería tener envío gratis.

Caso borde importante: comparar valores que podrían no ser números. Si haces `"abc" > 5`, JavaScript intenta convertir `"abc"` a número, obtiene `NaN` (Not a Number), y cualquier comparación con `NaN` da `false`. Hasta `"abc" < 5` da `false`. Y `NaN === NaN` da `false` también (sí, `NaN` no es igual ni a sí mismo; es una de las rarezas oficiales del lenguaje). Si necesitas detectar `NaN`, existe la función `Number.isNaN()`. Esto importa cuando lees datos de un formulario: todo lo que escribe el usuario llega como texto, y si lo comparas directamente con números sin convertirlo primero, puedes obtener resultados falsos silenciosos. La buena práctica es convertir explícitamente con `Number()` antes de comparar, y validar que el resultado no sea `NaN`.

Otro caso borde: los decimales. En JavaScript (y en casi todos los lenguajes), `0.1 + 0.2` no es exactamente 0.3, sino `0.30000000000000004`, por cómo se guardan los decimales en binario. Entonces `(0.1 + 0.2) === 0.3` da `false`. Esto no es un error de JavaScript, es una característica de la aritmética de punto flotante, pero sorprende a todos la primera vez. Si comparas

cantidades de dinero con decimales, la práctica segura es trabajar en centavos (números enteros) o comparar con un margen de tolerancia en lugar de igualdad exacta. No te voy a mentir: este es de esos detalles que separan al programador que "hace que funcione" del que "hace que funcione bien".

Vamos a hablar de buenas prácticas con comparaciones, en forma de reglas cortas que puedes pegar en tu mente. Primera: siempre `===` y `!==`, nunca `==` ni `!=`, salvo que tengas una razón documentada. Segunda: cuando compares rangos, escribe la condición de forma que se lea como una frase: `edad >= 18 && edad <= 65` se lee "edad entre 18 y 65". Tercera: evita las negaciones dobles; en lugar de `if (!(edad < 18))` escribe `if (edad >= 18)`, que se entiende a la primera. Cuarta: si una comparación se repite mucho, guárdala en una variable con nombre descriptivo: `const esMayorDeEdad = edad >= 18`; después usas `esMayorDeEdad` en tus condiciones y tu código se lee solo. Quinta: cuidado con comparar textos sin normalizar mayúsculas, minúsculas y espacios (los espacios al inicio o al final también rompen comparaciones: `"ana" !== "ana"`).

Pregunta frecuente: "¿entonces nunca uso `==`?" Casi nunca. La única excepción que la comunidad acepta a medias es comparar con `null` cuando quieres atrapar tanto `null` como `undefined` de un jalón: `valor == null` es `true` tanto si `valor` es `null` como si es `undefined`. Pero incluso eso, muchos equipos lo escriben explícito: `valor === null || valor === undefined`. Más largo, pero nadie tiene que adivinar qué quisiste decir. En código profesional, la claridad le gana a la brevedad casi siempre.

Pregunta frecuente: "¿puedo comparar objetos o arrays con `===`?" Puedes, pero no hace lo que esperas: `[1,2] === [1,2]` da `false`, porque `===` con objetos compara la referencia (si es el mismo objeto en memoria), no el contenido. Son dos arrays distintos aunque se vean iguales, como dos gemelos: se parecen pero no son la misma persona. Comparar el contenido de objetos y arrays es un tema más avanzado que verás después; por ahora quédate con que `===` funciona como esperas con números, textos y booleanos, que es lo que vas a comparar el 95% del tiempo.

Pregunta frecuente: "¿qué es `truthy` y `falsy`?" En JavaScript, cuando un valor que no es booleano aparece donde se espera un booleano (como dentro de un `if`), el lenguaje lo convierte: algunos valores se comportan como `false` (los llamamos `falsy`) y todo lo demás se comporta como `true` (`truthy`). Los `falsy` son exactamente estos seis: `false`, `0`, `""` (texto vacío), `null`, `undefined` y `NaN`. Todo lo demás es `truthy`, incluyendo cosas que sorprenden: `"0"` (texto con cero) es `truthy`, `[]` (array vacío) es `truthy`, `{}` (objeto vacío) es `truthy`. Esto conecta con los operadores lógicos que verás en la siguiente sección, especialmente con el cortocircuito, así que tenlo presente.

Operadores lógicos: combinar preguntas como un profesional

Ya sabes hacer preguntas simples: ¿es mayor?, ¿es igual? Ahora viene lo divertido: combinar varias preguntas en una sola condición compleja. Para eso están los operadores lógicos: && (Y lógico, AND), || (O lógico, OR) y ! (NO lógico, NOT). Con estos tres puedes expresar cualquier regla del mundo real, desde "puede entrar al antro" hasta "aplica para el crédito hipotecario".

Empecemos con &&, que significa "Y". La regla es simple: el resultado es true solo si AMBAS condiciones son true. Si cualquiera de las dos es false, el resultado es false. Su tabla de verdad (una tablita que muestra todos los casos posibles) es así, escrita en palabras para que no necesites dibujarla: true && true da true; true && false da false; false && true da false; false && false da false. En resumen: con &&, basta un false para arruinarlo todo. Es como un equipo donde todos tienen que estar de acuerdo: si uno dice que no, no se hace.

Ejemplo de la vida real: para entrar a un antro necesitas tener 18 o más Y traer identificación. Si tienes 20 pero no traes ID, no entras. Si traes ID pero tienes 16, no entras. Solo entras si cumples las dos. En código:

```
const edad = 20;
const traeID = true;
const puedeEntrar = edad >= 18 && traeID;
```

puedeEntrar vale true. Cambia edad a 16 y vale false. Cambia traeID a false y vale false. Así de directo.

Ahora ||, que significa "O". La regla: el resultado es true si AL MENOS UNA de las condiciones es true. Solo da false cuando ambas son false. Su tabla de verdad en palabras: true || true da true; true || false da true; false || true da true; false || false da false. Con ||, basta un true para salvarlo todo. Es como tener dos llaves distintas para la misma puerta: con que una funcione, entras.

Ejemplo: una tienda da descuento si compras más de 1000 pesos O si eres cliente frecuente. Con que cumplas una de las dos, tienes descuento. Solo te quedas sin descuento si no cumples ninguna.

```
const compra = 800;
const esFrecuente = true;
const tieneDescuento = compra > 1000 || esFrecuente;
```

tieneDescuento vale true porque esFrecuente es true, aunque la compra no llegue a 1000. ¿Te suena? Es el ejemplo de la anécdota de la introducción, donde mi amigo usó mal el operador y regaló descuentos. Si hubiera usado && en lugar de ||, el descuento solo aplicaría a clientes frecuentes que además compraran más de 1000. Una letrita de diferencia (&& vs ||) y el negocio pierde dinero. Por eso esta sección importa.

Y!, el NO lógico, que es el más simple: invierte el valor.!true da false,!false da true. Se usa para preguntar "¿no se cumple esto?" Ejemplo:!esMayorDeEdad significa "no es mayor de edad". Es útil, pero hay que usarlo con moderación porque las negaciones múltiples confunden:!valor (doble negación) convierte cualquier valor a su booleano equivalente, un truco que verás en código ajeno. Y!(a && b) se puede reescribir como!a ||!b (esto se llama leyes de De Morgan y te puede salvar cuando una condición negada se vuelve un trabalenguas).

Aquí viene una de las ideas más importantes y menos intuitivas de JavaScript: el cortocircuito (short-circuit evaluation). Resulta que && y || no siempre evalúan ambos lados. JavaScript es flojo de forma inteligente: si con el primer valor ya puede saber el resultado, ni siquiera mira el segundo. Con &&: si el primer valor es falsy, el resultado ya se sabe (va a ser ese valor falsy) y JavaScript no evalúa lo demás. Con ||: si el primer valor es truthy, el resultado ya se sabe y no evalúa lo demás. Esto no es solo una optimización: es una herramienta que los programadores usan todos los días.

El uso más común del cortocircuito con && es ejecutar algo solo si una condición se cumple, en una sola línea. Por ejemplo: usuario && mostrarPerfil(usuario). Si usuario es null (no hay nadie logueado), la expresión se detiene ahí y mostrarPerfil nunca se ejecuta, evitando un error. Si usuario existe, entonces sí se ejecuta la función. Es un guardián: "solo sigue si esto existe". Verás este patrón muchísimo en código real.

Con ||, el uso clásico es dar valores por defecto: const nombre = nombreIngresado || "Invitado". Si nombreIngresado es un texto vacío (falsy), se usa "Invitado". Si el usuario sí escribió algo (truthy), se usa lo que escribió. Es elegante y corto. Pero cuidado con la trampa: si nombreIngresado es 0 o "" y esos son valores válidos en tu lógica (por ejemplo, una cantidad de 0 productos), el || los va a reemplazar por el valor por defecto aunque sean legítimos. Para esos casos existe el operador?? (fusión de nulos), que solo reemplaza null y undefined, pero eso lo verás más adelante; por ahora quédate con que || con valores por defecto es práctico pero hay que conocer su límite.

Otro detalle del cortocircuito que causa errores sutiles: si el segundo operando tiene un efecto secundario (como llamar a una función que guarda algo), ese efecto puede no ocurrir si el

primero ya decidió el resultado. Ejemplo: `valido && guardarDatos()`. Si `valido` es `false`, `guardarDatos` nunca se llama. Normalmente eso es justo lo que quieres, pero si no entiendes el cortocircuito, te vas a preguntar horas por qué tu función "a veces no se ejecuta". Ahora ya sabes por qué.

Hablemos de precedencia de operadores, que es el orden en que JavaScript evalúa una expresión cuando mezcla varios operadores. Es como en matemáticas: en $2 + 3 * 4$ primero se hace la multiplicación. En lógica, el orden es: primero! (NOT), después `&&` (AND), y al final `||` (OR). Y las comparaciones (`>`, `===`, etc.) se evalúan antes que los lógicos. Entonces en la expresión `edad >= 18 && tieneID || esVIP`, JavaScript primero evalúa `edad >= 18`, luego hace el `&&` con `tieneID`, y al final el `||` con `esVIP`. El resultado es `(edad >= 18 && tieneID) || esVIP`: entra si (es mayor y trae ID) o si es VIP.

¿Te confunde? A todos. Por eso la buena práctica número uno con precedencia es: no confíes en tu memoria, usa paréntesis. Escribe `((edad >= 18 && tieneID) || esVIP)` aunque los paréntesis sean técnicamente innecesarios. Los paréntesis no le cuestan nada a la computadora y le ahorran minutos de confusión al humano que lea tu código (que muchas veces vas a ser tú mismo dentro de tres meses). El código se escribe una vez y se lee diez veces; optimiza para la lectura.

Un error típico con precedencia: querer decir "si el día es sábado o domingo y hace sol, vamos a la playa" y escribir `dia === "sabado" || dia === "domingo" && haceSol`. Por precedencia, esto se evalúa como `dia === "sabado" || (dia === "domingo" && haceSol)`: si es sábado, vamos a la playa aunque esté lloviendo. Lo que querías era `(dia === "sabado" || dia === "domingo") && haceSol`. Un par de paréntesis y el plan del fin de semana se salva. Este ejemplo parece tonto hasta que te pasa con reglas de negocio reales.

Otro error clásico: encadenar comparaciones como en matemáticas. En matemáticas escribes $18 \leq \text{edad} \leq 65$ y se entiende. En JavaScript, `18 <= edad <= 65` NO hace eso. Se evalúa de izquierda a derecha: primero `18 <= edad` da `true` o `false`, y luego ese booleano se compara con `65` (`true` se convierte a `1`, y `1 <= 65` siempre es `true`). Resultado: casi siempre `true`, y tu validación de rango no sirve para nada. La forma correcta es `edad >= 18 && edad <= 65`. Este error es traicionero porque el código "se ve bien" y no da error, simplemente deja pasar valores que no debería. Si validas rangos (edades, precios, calificaciones), usa siempre el `&&` explícito.

Las tablas de verdad merecen un momento más porque son tu herramienta para no perderte con condiciones complejas. Cuando tengas una condición con dos o tres variables y no estés seguro de qué hace, escribe todos los casos en un comentario o en papel: ¿qué pasa si A es

true y B es true? ¿A true B false? ¿A false B true? ¿A false B false? Son solo cuatro casos con dos variables. Hacer esto toma un minuto y te evita el 90% de los errores lógicos. Los programadores experimentados lo hacen mentalmente sin darse cuenta; tú puedes hacerlo en papel hasta que se vuelva automático. No es de principiantes, es de profesionales cuidadosos.

¿Sabías que...? Los operadores `&&` y `||` en JavaScript no devuelven necesariamente true o false: devuelven uno de los operandos. `true && "hola"` devuelve "hola", no true. `0 || "valor"` devuelve "valor". `false && cualquierCosa` devuelve false sin evaluar cualquierCosa. Esto es consecuencia directa del cortocircuito y es lo que hace útiles los patrones de guardián y valor por defecto que vimos. Al principio parece raro, pero cuando lo interiorizas, empiezas a leer código real con mucha más fluidez porque esos patrones están por todas partes.

Cuidado con... mezclar! con comparaciones sin paréntesis. `!edad >= 18` no significa "la edad no es mayor o igual a 18" como parece: el `!` se aplica primero a `edad` (convirtiéndola a booleano e invirtiéndola) y luego se compara ese booleano con 18. El resultado es un sinsentido. Lo correcto es `!(edad >= 18)`. Otra vez: paréntesis, paréntesis, paréntesis.

Cuidado con... usar `&&` y `||` con valores no booleanos sin querer. Si haces `const resultado = a && b` donde `a` y `b` son números, resultado será un número, no un booleano. Si después comparas `resultado === true`, te vas a llevar una sorpresa. Cuando necesites un booleano de verdad, envuelve la expresión: `Boolean(a && b)` o `!!(a && b)`.

Vamos a ver ahora cómo se combinan comparaciones y lógica en validaciones típicas, porque validar es probablemente el uso número uno de todo esto. Validar significa revisar que los datos cumplan las reglas antes de usarlos: que la contraseña tenga al menos 8 caracteres, que el correo contenga un `@`, que la edad esté en un rango válido, que el usuario haya aceptado los términos. Cada validación es una comparación o una combinación de comparaciones. Un formulario de registro típico tiene cinco o seis validaciones, cada una con su mensaje de error. Si dominas esta sección, dominas los formularios, y los formularios son el pan de cada día del desarrollo web.

Una buena práctica al validar: en lugar de una sola condición gigante imposible de leer, divide en validaciones pequeñas con mensajes específicos. Código malo: `if (nombre.length > 2 && correo.includes("@") && clave.length >= 8 && edad >= 18) {registrar();} else {alert("Datos inválidos");}`. El usuario no sabe qué está mal. Código bueno: revisar cada cosa por separado y decir exactamente qué falló. Es más líneas, pero es mejor programa. La brevedad no es una virtud cuando sacrifica la claridad.

Mini-reto mental antes de seguir: sin correr código, ¿qué devuelve `(5 > 3 && 10 < 20) || false`? Primero `5 > 3` es true, `10 < 20` es true, `true && true` es true, `true || false` es true. Respuesta: true. Si lo sacaste bien, vas por buen camino. Si no, repasa las tablas de verdad; no hay prisa.

El ciclo for: tu primera máquina de repetir

Llegamos al tercer bloque de la unidad y a uno de los conceptos más satisfactorios de la programación: el ciclo for. Un ciclo (loop) es una estructura que repite un bloque de código varias veces. El for es el ciclo que usas cuando sabes de antemano cuántas veces quieres repetir algo, o cuando quieres recorrer un rango de valores paso a paso.

La sintaxis del for tiene tres partes dentro de los paréntesis, separadas por punto y coma. Memorízala como una frase: "empiezo en X; sigo mientras Y; cada vuelta hago Z". En código:

```
for (let i = 0; i < 5; i++) {  
  console.log(i);  
}
```

Vamos a diseccionar esto con calma porque cada parte tiene su trabajo. Primera parte: `let i = 0`. Es la inicialización: crea una variable contadora (por convención se llama `i`, de "índice" o "iterador") y le da su valor inicial. Cero es el inicio más común en programación, y hay una razón histórica y práctica: los arrays empiezan en cero, así que contar desde cero te prepara para recorrerlos. Segunda parte: `i < 5`. Es la condición: antes de cada vuelta, JavaScript la evalúa; si es true, ejecuta el bloque; si es false, el ciclo termina. Tercera parte: `i++`. Es la actualización: se ejecuta al final de cada vuelta, y aquí incrementamos el contador en uno. Sin esta parte, `i` siempre valdría 0, la condición siempre sería true, y tendrías un ciclo infinito (del que hablaremos en la próxima unidad con más detalle, pero adelanto: es malo).

El flujo completo de ese ejemplo: `i` empieza en 0; ¿`0 < 5`? sí, imprime 0; `i` sube a 1; ¿`1 < 5`? sí, imprime 1; `i` sube a 2; imprime 2; `i` sube a 3; imprime 3; `i` sube a 4; imprime 4; `i` sube a 5; ¿`5 < 5`? no, se acabó. Resultado: imprime 0, 1, 2, 3, 4. Cinco vueltas, del 0 al 4. Fíjate: con condición `i < 5` obtienes los números del 0 al 4, no hasta el 5. Este es el famoso "empieza en cero, termina en `n` menos uno", y es la convención más usada en programación. Si quieres los números del 1 al 5, puedes hacer `for (let i = 1; i <= 5; i++)`. Ambas formas son correctas; lo importante es que sepas exactamente cuántas vueltas da tu ciclo y qué valores toma `i` en cada una. Cuando tengas duda, cuenta con los dedos las primeras vueltas; no da pena, da precisión.

Analogía de la vida real: el ciclo for es como una receta que dice "repite los pasos 3 al 5 para cada invitado". No reescribes los pasos para cada invitado; los escribes una vez y dices cuántas veces aplicarlos. O como un sello: tallas el diseño una vez y lo estampas cien veces. El bloque dentro de las llaves es el sello; las tres partes del paréntesis dicen cuántas veces estamparlo y cómo va cambiando el contador en cada estampada.

Vamos a ver para qué sirve en la práctica recorrer rangos. El uso más típico: hacer algo N veces. Por ejemplo, mostrar los primeros 10 múltiplos de 3, o simular 100 lanzamientos de un dado, o pedir 5 calificaciones. Otro uso: recorrer un rango de números para buscar algo, como encontrar todos los números pares entre 1 y 100, o sumar los números del 1 al 100 (que da 5050, por cierto; hay una anécdota famosa de Gauss haciéndolo de niño en segundos, pero con un for lo haces en milisegundos sin ser Gauss). Otro uso: generar secuencias, como la tabla de multiplicar del 7, o una cuenta regresiva.

La cuenta regresiva es un buen ejemplo para ver que el contador puede ir hacia atrás: for (let i = 10; i > 0; i--). Aquí la inicialización es 10, la condición es i > 0, y la actualización es i-- (decremento). Imprime 10, 9, 8... 1. Útil para temporizadores, para mostrar "quedan X intentos", o para recorrer cosas al revés.

También puedes cambiar el paso: no tienes que ir de uno en uno. for (let i = 0; i <= 100; i += 10) va de 10 en 10. for (let i = 1; i <= 1024; i *= 2) duplica en cada vuelta: 1, 2, 4, 8... 1024. El tercer segmento del for acepta cualquier expresión que modifique el contador. Esto es útil para, por ejemplo, recorrer solo los números pares (i += 2) o para retroceder de dos en dos.

Ahora, los ciclos for anidados: un for dentro de otro for. Suena a trabalenguas, pero la idea es simple: por cada vuelta del ciclo exterior, el ciclo interior se ejecuta completo. Es como las manecillas del reloj: la manecilla de los minutos da una vuelta completa por cada hora que avanza la de las horas. O como una tabla: el ciclo exterior recorre las filas y el interior recorre las columnas de cada fila.

El ejemplo clásico es dibujar patrones o trabajar con cuadrículas:

```
for (let fila = 1; fila <= 3; fila++) {  
  for (let columna = 1; columna <= 3; columna++) {  
    console.log(fila + "," + columna);  
  }  
}
```

Esto imprime: 1,1; 1,2; 1,3; 2,1; 2,2; 2,3; 3,1; 3,2; 3,3. Nueve combinaciones: todas las parejas posibles. Este patrón de "todas las combinaciones" aparece en montones de problemas reales:

comparar cada producto con cada otro producto, probar cada usuario con cada permiso, generar todas las casillas de un tablero.

Otro ejemplo: la tabla de multiplicar completa del 1 al 10 son dos for anidados (uno para cada factor). Un calendario mensual es filas (semanas) por columnas (días). Una imagen es filas de píxeles por columnas de píxeles. Cada vez que tu problema tenga dos dimensiones, piensa en for anidados.

Advertencia honesta sobre los for anidados: el número total de vueltas se multiplica. Si el exterior da 100 vueltas y el interior 100, el bloque interior se ejecuta 10,000 veces. Con tres niveles de 100, son un millón. Esto no es malo en sí, pero debes ser consciente: los for anidados crecen rápido y si los usas con datos grandes sin pensar, tu programa se puede poner lento. Para los ejercicios de este curso no hay problema, pero es un buen hábito preguntarte "¿cuántas vueltas totales da esto?".

Dos palabras mágicas para controlar ciclos: break y continue. break rompe el ciclo por completo: sale inmediatamente, sin importar cuántas vueltas faltaban. Es como decir "ya encontré lo que buscaba, no sigas buscando". Ejemplo típico: buscar un nombre en una lista; cuando lo encuentras, break, porque ya no tiene sentido seguir. Sin break, el ciclo seguiría revisando el resto inútilmente.

continue es más sutil: no rompe el ciclo, solo se salta el resto de la vuelta actual y pasa a la siguiente. Es como decir "esta vuelta no me interesa, sigue con la que sigue". Ejemplo: imprimir solo los números impares del 1 al 10; cuando i es par, continue, y te saltas la impresión. O procesar una lista de usuarios saltándote los que están inactivos.

Mira la diferencia con un ejemplo concreto. Con break: buscas el número 7 entre el 1 y el 100; cuando i llega a 7, break, y el ciclo termina habiendo dado solo 7 vueltas. Con continue: quieres sumar solo los números impares; cuando i es par, continue salta la suma pero el ciclo sigue hasta el 100. break es "ya terminé todo", continue es "esta vuelta me la salto".

Un error común con break y continue en for anidados: break solo rompe el ciclo más interno, no todos. Si estás en un for dentro de otro for y haces break, sales del interior pero el exterior sigue. Si necesitas salir de ambos, hay técnicas (como una variable bandera o las etiquetas de ciclo), pero son temas más avanzados; por ahora, con saber que break afecta solo a su ciclo directo es suficiente para no confundirte.

Otro error típico: modificar el contador i dentro del bloque del for sin darte cuenta de que el tercer segmento también lo modifica. Por ejemplo, hacer i++ dentro del bloque además del i++ del paréntesis hace que el contador avance de dos en dos sin que sea obvio al leer el

encabezado. A veces es intencional, pero casi siempre es un bug. Regla práctica: trata a `i` como de solo lectura dentro del bloque; si necesitas otra variable que cambie, crea una nueva.

El error más famoso de todos con `for`: el ciclo infinito por una condición que nunca se vuelve `false`. Ejemplo: `for (let i = 0; i >= 0; i++)`. La condición `i >= 0` siempre es `true` porque `i` solo crece. El ciclo nunca termina, tu programa se congela y tienes que cerrarlo a la fuerza. En el navegador, la pestaña se traba. Da miedo la primera vez, pero no rompe nada: cierras la pestaña y listo. La lección es revisar siempre que tu condición pueda volverse `false` en algún momento. En la próxima unidad veremos el ciclo `while`, donde los infinitos son aún más fáciles de provocar, y hablaremos de cómo escapar de ellos.

¿Sabías que...? El `for` con sus tres partes viene del lenguaje C, creado en los años 70, y casi todos los lenguajes modernos lo copiaron. Es una de las sintaxis más longevas e imitadas de la historia de la computación. Cuando escribes un `for` en JavaScript, estás usando una idea que tiene más de cincuenta años y que ha sobrevivido porque funciona muy bien.

Cuidado con... declarar el contador con `var` en lugar de `let` dentro del `for`. Con `var`, la variable `i` sobrevive después del ciclo y puede causar problemas si la reutilizas; con `let`, `i` solo existe dentro del ciclo, que es lo que quieres casi siempre. En este curso usamos `let` siempre. Es un detalle pequeño que evita dolores de cabeza grandes.

Cuidado con... el punto y coma de más: `for (let i = 0; i < 5; i++); {console.log(i);}`. ¿Ves ese punto y coma después del paréntesis? Eso termina el ciclo ahí mismo: el `for` repite "nada" cinco veces, y luego el bloque entre llaves se ejecuta una sola vez, con `i` valiendo 5. El programa no da error, simplemente hace algo rarísimo. Si tu `for` "no hace lo que debería", busca un punto y coma intruso.

Pregunta frecuente: "¿cuándo uso `for` y cuándo uso `while`?" La regla práctica: si sabes cuántas veces vas a repetir (o recorres un rango, o una lista), usa `for`; si repites hasta que pase algo que no sabes cuándo va a pasar (como "repetir hasta que el usuario escriba 'salir'"), usa `while`. En la próxima unidad verás el `while` a fondo y esta decisión te quedará clarísima.

Pregunta frecuente: "¿puedo usar `for` para recorrer un array?" Sí, y es uno de sus usos principales, pero en este curso lo verás en la unidad de arrays con todo detalle, incluyendo alternativas modernas como `forEach`. Por ahora, quédate con que `for (let i = 0; i < array.length; i++)` es el patrón clásico para visitar cada elemento por su posición.

Vamos a cerrar la explicación con una idea que une los tres bloques de la unidad. Fíjate en el `for`: su segunda parte es una comparación (`i < 5`) y su funcionamiento depende de la lógica booleana (mientras sea `true`, sigue). Las comparaciones, la lógica y los ciclos no son tres

temas separados: son tres caras de lo mismo. Comparar produce booleanos, la lógica combina booleanos, y los ciclos obedecen booleanos. Cuando escribes `for (let i = 0; i < 5 && !terminado; i++)`, estás usando los tres bloques en una sola línea: comparación (`i < 5`), lógica (`!terminado, &&`) y repetición (`for`). Esa es la verdadera meta de esta unidad: que estas tres herramientas se vuelvan un solo reflejo en tus manos.

Un último consejo antes de los ejemplos: cuando un ciclo no hace lo que esperas, la técnica de debugging más efectiva es imprimir el contador. Pon un `console.log(i)` dentro del ciclo y mira qué valores toma. El noventa por ciento de las veces, el problema es que `i` no vale lo que creías: empieza donde no debe, termina donde no debe, o salta valores. Ver los valores reales te saca de la confusión en segundos. Los programadores profesionales hacen esto todos los días; no es trampa, es la técnica oficial.

Ejemplos paso a paso

En esta sección vamos a ver ejemplos completos, explicados línea por línea, como si estuviéramos programando juntos y yo te fuera narrando lo que pienso mientras escribo. La idea no es que copies y pegues: es que entiendas el razonamiento detrás de cada línea, porque ese razonamiento es lo que vas a usar cuando escribas tus propios programas.

Ejemplo 1: ¿Puede entrar al antro? (comparaciones básicas y ===)

Vamos a empezar con algo sencillo pero que usa comparaciones de verdad. La regla: una persona puede entrar a un evento si tiene 18 años o más. Vamos a escribirlo paso a paso.

Primero, pensemos qué necesitamos: la edad de la persona y una forma de decidir. La decisión es una comparación: ¿la edad es mayor o igual a 18? Fíjate que uso `>=` y no `>`, porque alguien con exactamente 18 años sí puede entrar. Ese detalle del "o igual" es exactamente el tipo de decisión que separa un programa correcto de uno con un bug.

```
let edad = 20;

// Preguntamos: ¿la edad es mayor o igual a 18?
// El operador >= compara y devuelve true o false.
let puedeEntrar = edad >= 18;

// Mostramos el resultado
console.log(puedeEntrar); // true, porque 20 >= 18
```

Paso a paso: declaramos edad con valor 20. Luego evaluamos `edad >= 18`: JavaScript compara 20 con 18, ve que 20 es mayor, y la expresión completa se convierte en `true`. Ese `true` se guarda en `puedeEntrar`. Al imprimirlo, vemos `true`.

Ahora cambia la edad a 16 en tu cabeza (o mejor, en tu consola) y sigue el mismo razonamiento: `16 >= 18` es `false`, `puedeEntrar` vale `false`. Y con edad = 18: `18 >= 18` es `true`, porque el "o igual" se cumple. Prueba los tres casos siempre: por debajo del límite, exactamente en el límite y por encima. Esa es la disciplina que te salva de los errores off-by-one.

Vamos a hacerlo un poco más real: en lugar de un número fijo, la edad viene de una variable que podría ser texto (porque los formularios devuelven texto). Aquí entra la conversión explícita y el `===`:

```
let edadTexto = "20"; // así llega de un formulario: como texto

// Convertimos a número ANTES de comparar. Buena práctica.
let edad = Number(edadTexto);

// Ahora sí comparamos números con números
let puedeEntrar = edad >= 18;
console.log(puedeEntrar); // true
```

¿Por qué convertimos primero? Porque si comparas `"20" >= 18`, JavaScript lo convierte solo y funciona, pero estás dependiendo de la conversión automática, y ya vimos que depender de la magia automática es mala idea. Conviertes tú, explícito, y luego comparas con `===` o `>=` entre números. Código predecible es código confiable.

Y un ejemplo de lo que NO debes hacer, para que lo reconozcas:

```
// CÓDIGO MALO: no lo escribas así
let edad = "20";
if (edad == 18) {
  console.log("tiene 18");
}
```

Esto usa `==` con un texto y un número, depende de la conversión automática y además compara con 18 exacto en lugar del rango. La versión correcta usa `Number()` y `>=`. Cuando veas `==` en código, ya sabes: bandera roja.

Ejemplo 2: Validador de contraseña (===,!==, length)

Vamos a validar que una contraseña cumpla dos reglas: que tenga al menos 8 caracteres y que no sea igual al nombre de usuario (porque eso es inseguro). Necesitamos comparar longitudes y comparar textos con ===.

```
// Datos de ejemplo
let usuario = "ana2024";
let clave = "gato123";

// Regla 1: la clave debe tener 8 o más caracteres
// .length nos dice cuántos caracteres tiene un texto
let largoOK = clave.length >= 8;

// Regla 2: la clave no debe ser igual al usuario
// Usamos !== (desigualdad estricta)
let esDistinta = clave !== usuario;

// La clave es válida solo si cumple LAS DOS reglas: &&
let claveValida = largoOK && esDistinta;

console.log(largoOK);    // false: "gato123" tiene 7 caracteres
console.log(esDistinta); // true: "gato123" !== "ana2024"
console.log(claveValida); // false: con && basta un false
```

Sigue el razonamiento: "gato123" tiene 7 letras, $7 \geq 8$ es false, así que largoOK es false. La clave sí es distinta del usuario, esDistinta es true. Pero claveValida necesita ambas en true; false && true da false. La clave se rechaza, y sabemos exactamente por qué: es muy corta. Fíjate qué útil es guardar cada comparación en su propia variable con nombre descriptivo: cuando algo falla, sabes cuál de las reglas falló. Si hubiéramos escrito todo en una sola línea gigante, diagnosticar sería más difícil.

Ahora prueba con clave = "gato12345": tiene 9 caracteres, largoOK es true, esDistinta es true, claveValida es true. Y prueba con clave = "ana2024" (igual al usuario): largoOK es true (7... no, espera, "ana2024" tiene 7 caracteres, así que largoOK sería false). Prueba con usuario = "ana" y clave = "ana": largoOK false, esDistinta false, claveValida false. Juega con los valores; cada combinación te enseña algo.

Nota sobre.length: es una propiedad de los textos que devuelve su longitud. La verás muchísimo en validaciones. Y recuerda normalizar si comparas textos escritos por humanos: usuario.toLowerCase() === "ana" atrapa "ANA", "Ana" y "ana".

Ejemplo 3: Descuento de la tienda (|| y && combinados, precedencia)

Retomemos la anécdota de la introducción: una tienda da 10% de descuento si la compra es mayor a 1000 pesos O el cliente es frecuente. Vamos a escribirlo bien, paso a paso, y luego vamos a ver qué pasa si lo escribimos mal.

```
// Datos de la compra
let totalCompra = 800;
let esFrecuente = true;

// Regla: descuento si compra > 1000 O es cliente frecuente
// Con || basta que UNA se cumpla
let tieneDescuento = totalCompra > 1000 || esFrecuente;

console.log(tieneDescuento); // true: es frecuente, aunque la compra es 800
```

Razonamiento: `totalCompra > 1000` es `false` (800 no es mayor que 1000). `esFrecuente` es `true`. `false || true` da `true`. Descuento aplicado. Ahora, ¿qué habría pasado con `&&`? `totalCompra > 1000 && esFrecuente` sería `false && true = false`: sin descuento. ¿Cuál es la correcta? Depende de la regla del negocio. Si la regla dice "O", es `||`. El bug de mi amigo fue escribir una cuando la regla pedía la otra (o al revés). Moraleja: antes de escribir el operador, escribe la regla en español en un comentario, y luego traduce: "o" es `||`, "y" es `&&`, "ambas" es `&&`, "al menos una" es `||`.

Ahora un caso con las tres cosas mezcladas y paréntesis explícitos. Nueva regla: descuento si (la compra supera 1000 Y el cliente trae cupón) O si el cliente es VIP. El VIP siempre tiene descuento, sin condiciones.

```
let totalCompra = 1200;
let traeCupon = false;
let esVIP = false;

// Sin paréntesis sería confuso; los ponemos aunque "no hagan falta"
let tieneDescuento = (totalCompra > 1000 && traeCupon) || esVIP;

console.log(tieneDescuento); // false: (true && false) || false = false
```

Paso a paso por precedencia: primero se evalúan las comparaciones (`totalCompra > 1000` es `true`). Luego el `&&` dentro del paréntesis: `true && false` da `false`. Luego el `||`: `false || false` da `false`. Sin descuento, correcto: la compra es grande pero no trae cupón, y no es VIP. Cambia `traeCupon` a `true` y el resultado cambia a `true`. Cambia `esVIP` a `true` y también da `true` sin

importar lo demás. Esa es la belleza de escribir casos de prueba mentales: con tres variables booleanas hay 8 combinaciones; no necesitas probarlas todas, pero prueba las que importan: todas false, todas true, y las de la frontera.

Ejemplo 4: Semáforo con cortocircuito (&& como guardián)

Vamos a ver el cortocircuito en acción con un ejemplo práctico: solo queremos saludar al usuario si hay un usuario logueado (si no hay, no hacemos nada y, sobre todo, no provocamos un error).

```
// Simulamos dos situaciones: con usuario y sin usuario
let usuarioLogueado = "Carlos";

// El && actúa como guardián: si lo primero es falsy, lo segundo ni se ejecuta
usuarioLogueado && console.log("Hola, " + usuarioLogueado + "!");
// Imprime: Hola, Carlos!

// Ahora sin usuario (null significa "nadie logueado")
usuarioLogueado = null;
usuarioLogueado && console.log("Hola, " + usuarioLogueado + "!");
// No imprime nada. El && vio null (falsy) y se detuvo ahí.
```

Este patrón es oro en el desarrollo real: antes de usar un valor que podría no existir, lo pones a la izquierda del &&. Es como tocar la puerta antes de entrar: si no hay nadie (null), no entras y no rompes nada. Sin este guardián, intentar usar usuarioLogueado.nombre cuando es null te daría un error que tumba tu programa.

Y el || para valores por defecto:

```
let nombreEscrito = ""; // el usuario no escribió nada
let nombreAMostrar = nombreEscrito || "Invitado";
console.log(nombreAMostrar); // "Invitado": el texto vacío es falsy

nombreEscrito = "Lucía";
nombreAMostrar = nombreEscrito || "Invitado";
console.log(nombreAMostrar); // "Lucía": el texto no vacío es truthy
```

Pero recuerda la trampa que vimos en la explicación: si el valor 0 es válido en tu lógica, || lo va a reemplazar. Ejemplo: un contador de intentos donde 0 es un valor legítimo.

```
let intentos = 0;
let intentosAMostrar = intentos || 5; // CUIDADO: da 5, no 0
console.log(intentosAMostrar); // 5, ¡pero el usuario llevaba 0 intentos!
```

Aquí 0 es falsy, así que || lo reemplaza por 5 aunque 0 era el valor real. Si 0 es un dato válido para ti, este patrón te miente. La lección: el || para valores por defecto solo sirve cuando los valores falsy (0, "", false) realmente significan "vacío" en tu contexto. Si no, necesitas una comparación explícita.

Ejemplo 5: Tabla de multiplicar con for (sintaxis y rangos)

Vamos a construir la tabla de multiplicar del 7, del 7x1 al 7x10. Es el ejemplo perfecto para ver las tres partes del for trabajando juntas.

```
// Queremos i = 1, 2, 3... hasta 10
for (let i = 1; i <= 10; i++) {
  // En cada vuelta, i vale algo distinto
  let resultado = 7 * i;
  console.log("7 x " + i + " = " + resultado);
}
```

Vamos vuelta por vuelta las primeras para que veas el mecanismo. Inicialización: $i = 1$. Condición: ¿ $1 \leq 10$? Sí. Bloque: resultado = 7, imprime "7 x 1 = 7". Actualización: $i++ \rightarrow i = 2$. Condición: ¿ $2 \leq 10$? Sí. Bloque: resultado = 14, imprime "7 x 2 = 14". Actualización: $i = 3...$ y así hasta $i = 10$: imprime "7 x 10 = 70", i sube a 11, ¿ $11 \leq 10$? No. Fin. Diez vueltas exactas, del 1 al 10.

Fíjate en la decisión $i \leq 10$ en lugar de $i < 10$: como empezamos en 1 y queremos incluir el 10, necesitamos el "o igual". Si hubiéramos empezado en 0 con $i < 10$, tendríamos 10 vueltas pero del 0 al 9, y la tabla empezaría en "7 x 0 = 0", que no es lo que queremos. Moraleja: el inicio y la condición se deciden juntos, pensando en qué valores debe tomar i .

Variante: solo los pares del 1 al 20, usando un paso de 2.

```
// Empezamos en 2 y avanzamos de 2 en 2
for (let i = 2; i <= 20; i += 2) {
  console.log(i); // 2, 4, 6, 8... 20
}
```

Y una cuenta regresiva para lanzar un cohete:

```
for (let i = 5; i > 0; i--) {  
  console.log(i); // 5, 4, 3, 2, 1  
}  
console.log("¡Despegue!");
```

Aquí el contador va hacia atrás: inicialización en 5, condición $i > 0$, actualización $i--$. Cuando i llega a 0, la condición es false y el ciclo termina; el "¡Despegue!" está fuera del ciclo, así que se ejecuta una sola vez al final. Ojo con un detalle: si la condición fuera $i \geq 0$, imprimiría también el 0. ¿Lo quieres o no? Depende de tu cohete. Otra vez: los límites se deciden conscientemente.

Ejemplo 6: Buscar un nombre con break (salir temprano)

Tenemos una lista de invitados (todavía no hemos visto arrays a fondo, así que simulamos la lista con una serie de valores) y queremos saber si "Marta" está invitada. En lugar de revisar toda la lista siempre, nos detenemos en cuanto la encontramos: eso es break.

```
// Simulamos una lista con un texto separado por comas (truco temporal)  
let invitados = "Pedro,Ana,Marta,Luis";  
let buscada = "Marta";  
let encontrada = false;  
  
// Vamos a revisar 4 posiciones: 0, 1, 2, 3  
for (let i = 0; i < 4; i++) {  
  // Sacamos el invitado número i (esto lo entenderás mejor en la unidad de arrays)  
  let invitado = invitados.split(",")[i];  
  console.log("Revisando: " + invitado);  
  
  if (invitado === buscada) {  
    encontrada = true;  
    break; // ¡La encontramos! Ya no tiene caso seguir buscando  
  }  
}  
  
console.log("¿Está Marta? " + encontrada);
```


El flujo: $i=0$, revisa "Pedro", no es Marta, sigue. $i=1$, revisa "Ana", no es, sigue. $i=2$, revisa "Marta", ¡sí es! encontrada = true, break: el ciclo se rompe de inmediato. "Luis" ni siquiera se revisa. Resultado: encontrada es true y solo dimos 3 vueltas en lugar de 4. Con listas grandes, ese ahorro es enorme: si buscas un nombre entre un millón y está en la posición 10, break te ahorra 999,990 vueltas inútiles.

¿Qué pasaría sin el break? El ciclo seguiría hasta $i=3$, revisaría a Luis innecesariamente y el resultado sería el mismo, pero habrías desperdiciado trabajo. En listas pequeñas no importa; en datos grandes, break es la diferencia entre un programa rápido y uno lento. Acostúmbrate a preguntarte: "¿ya terminé? Entonces break".

Ejemplo 7: Sumar solo impares con continue (saltar vueltas)

Queremos sumar los números impares del 1 al 10. Con continue nos saltamos los pares sin romper el ciclo.

```
let suma = 0;

for (let i = 1; i <= 10; i++) {
  // Si i es par, nos saltamos esta vuelta
  // Un número es par si el residuo de dividir entre 2 es 0
  if (i % 2 === 0) {
    continue; // brinca al siguiente valor de i, sin sumar
  }
  // Solo llegamos aquí si i es impar
  suma = suma + i;
  console.log("Sumé " + i + ", total: " + suma);
}

console.log("Suma final: " + suma); // 25 (1+3+5+7+9)
```

Vuelta por vuelta: $i=1$, ¿ $1 \% 2 === 0$? No (residuo 1), no se ejecuta continue, suma = 1. $i=2$, ¿ $2 \% 2 === 0$? Sí, continue: se salta la suma y brinca a $i=3$. $i=3$ suma, $i=4$ se salta... Al final suma = $1+3+5+7+9 = 25$. Fíjate cómo continue te permite escribir la lógica al revés de lo habitual: en lugar de "si es impar, suma", escribes "si es par, sáltate", y el código principal queda sin anidar. Este estilo se llama "retorno temprano" o "salida temprana" y a muchos programadores les parece más legible porque elimina un nivel de indentación.

Compara break vs continue en este ejemplo: si hubiéramos usado break en lugar de continue, el ciclo habría terminado en i=2 (el primer par) y la suma sería solo 1. break = "ya acabé todo", continue = "esta vuelta no, la que sigue". No los confundas; pregúntate siempre: "¿quiero detener el ciclo o solo saltarme esta vuelta?".

Ejemplo 8: Cuadrícula con for anidados (todas las combinaciones)

Vamos a generar todas las casillas de un tablero de 3x3, como el del gato (tic-tac-toe). Necesitamos filas y columnas: dos for anidados.

```
// El ciclo exterior recorre las filas
for (let fila = 1; fila <= 3; fila++) {
  // Por CADA fila, el ciclo interior recorre todas las columnas
  for (let columna = 1; columna <= 3; columna++) {
    console.log("Casilla fila " + fila + ", columna " + columna);
  }
  console.log("--- fin de la fila " + fila + " ---");
}
```

El flujo es clave para entender los anidados: fila=1, y con fila fija en 1, el ciclo interior corre completo: columna 1, 2, 3. Termina el interior, se imprime el separador de fin de fila. Luego fila=2, y el interior OTRA VEZ completo: columna 1, 2, 3. Luego fila=3 igual. Total: 3 filas × 3 columnas = 9 casillas. Si lo corres, verás el patrón: las filas cambian lento, las columnas cambian rápido. Como el reloj: la hora cambia lento, los minutos rápido.

Vamos a usarlo para algo visual: dibujar un triángulo de asteriscos.

```

let linea = "";
// 5 filas; la fila i tiene i asteriscos
for (let fila = 1; fila <= 5; fila++) {
  linea = ""; // reiniciamos la línea en cada fila
  for (let col = 1; col <= fila; col++) {
    linea = linea + "*";
  }
  console.log(linea);
}
// Imprime:
// *
// **
// ***
// ****
// *****

```

Fíjate en el truco: el ciclo interior no siempre llega a 5, llega hasta fila. En la fila 1, el interior da 1 vuelta (un *). En la fila 2, da 2 vueltas (**). Por eso sale el triángulo. Los for anidados no tienen que ser cuadrados: el interior puede depender del exterior. Este patrón aparece en dibujos, en tablas y en muchos algoritmos. Y nota el detalle de reiniciar `linea = ""` en cada vuelta del exterior: si no lo haces, los asteriscos se acumulan y sale cualquier cosa. Los errores más comunes en for anidados son olvidar reiniciar el acumulador interno.

¿Sabías que...? Con for anidados puedes generar la tabla de multiplicar completa del 1 al 10 en unas 6 líneas: el exterior es un factor, el interior es el otro, y adentro multiplicas. Pruébalo como mini-reto: dos for, un `console.log` con `fila + "x" + columna + " = " + (fila * columna)`. Si te sale, ya dominas los anidados.

Ejemplo 9: Validador de rango con todo junto (el examen final de la unidad)

Vamos a juntar los tres bloques en un ejemplo realista: validar que una calificación esté en el rango válido (0 a 10) y clasificarla. Usa comparaciones, lógica y un for para probar varios casos.

```
function clasificar(calificacion) {
  // Validación del rango: debe estar entre 0 y 10, ambos incluidos
  let enRango = calificacion >= 0 && calificacion <= 10;

  if (!enRango) {
    return "Calificación inválida: debe ser de 0 a 10";
  }
  if (calificacion < 6) {
    return "Reprobado";
  }
  if (calificacion < 8) {
    return "Aprobado";
  }
  if (calificacion < 9) {
    return "Notable";
  }
  return "Excelente";
}

// Probamos con varios valores usando un for
let pruebas = [11, 10, 9.5, 8, 7, 6, 5.9, 0, -1];
for (let i = 0; i < pruebas.length; i++) {
  let c = pruebas[i];
  console.log(c + " -> " + clasificar(c));
}
```

Analicemos: primero validamos el rango con `calificacion >= 0 && calificacion <= 10`, el patrón de rango que aprendiste (nada de `0 <= calificacion <= 10`, que no funciona). Si no está en rango, devolvemos el error de inmediato. Luego clasificamos por umbrales: como ya sabemos que está entre 0 y 10, basta comparar con `< 6`, `< 8`, `< 9` en orden. Fíjate que el orden importa: si preguntáramos `< 9` antes que `< 6`, un 5 caería en "Notable" por error. Las condiciones se evalúan en orden y la primera que sea true gana.

El for de prueba recorre el array de casos: 11 (inválida), 10 (Excelente), 9.5 (Excelente), 8 (Notable), 7 (Aprobado), 6 (Aprobado, porque `6 < 6` es false y `6 < 8` es true), 5.9 (Reprobado), 0 (Reprobado), -1 (inválida). Incluimos los valores frontera (6, 8, 9, 10, 0) y los fuera de rango (11, -1) porque son los que más bugs esconden. Este hábito de probar fronteras te va a distinguir como programador cuidadoso.

Todavía no hemos visto funciones a fondo (eso es la próxima unidad), pero fíjate cómo clasificar empaqueta la lógica con un nombre claro: después la usas sin pensar en los detalles. Esa es una probadita de lo que viene.

Práctica

Aquí van los ejercicios. La regla de oro: intenta resolverlos TÚ primero, con lápiz y papel o en la consola, antes de ver la solución. Si te atorras más de quince minutos, mira la solución, entiéndela, ciérrala y vuelve a escribirla sin ver. Así es como el conocimiento se queda.

Ejercicio 1: Mayor de dos números (comparaciones)

Enunciado: Escribe un programa que tenga dos variables, a y b, con valores numéricos. Debe imprimir cuál de los dos es mayor, o si son iguales. Prueba con (10, 5), (3, 8) y (7, 7).

Solución:

```
let a = 10;
let b = 5;

if (a > b) {
  console.log(a + " es mayor que " + b);
} else if (b > a) {
  console.log(b + " es mayor que " + a);
} else {
  console.log("Son iguales");
}
// Con a=10, b=5 imprime: 10 es mayor que 5
```

Explicación paso a paso: Primero comparamos $a > b$. Con $a=10$ y $b=5$, $10 > 5$ es true, así que entra al primer bloque e imprime el mensaje. Los otros bloques se saltan. Con $a=3$ y $b=8$: $3 > 8$ es false, pasamos al else if: $8 > 3$ es true, imprime "8 es mayor que 3". Con $a=7$ y $b=7$: $7 > 7$ es false, $7 > 7$ es false, cae al else: "Son iguales". Fíjate que el orden de las preguntas importa: primero preguntamos si a es mayor, luego si b es mayor, y si ninguna es true, por descarte son iguales. No necesitamos una tercera comparación $a === b$, aunque también funcionaría. Este patrón de "pregunta, si no, pregunta otra cosa, si no, caso por defecto" es el esqueleto de casi todas las decisiones en programación.

Ejercicio 2: ¿Puedes votar? (operadores lógicos)

Enunciado: En un país ficticio se puede votar si la persona tiene 18 años o más Y está registrada en el padrón, O si tiene 16 o 17 años Y trae permiso firmado de sus papás. Escribe las variables edad, registrada y permisoFirmado, calcula puedeVotar con una sola expresión usando &&, || y paréntesis, e imprímela. Prueba con al menos tres combinaciones distintas.

Solución:

```
let edad = 17;
let registrada = false;
let permisoFirmado = true;

// (mayor de edad Y registrada) O (adolescente con permiso)
let puedeVotar = (edad >= 18 && registrada) || (edad >= 16 && permisoFirmado);

console.log(puedeVotar); // true: 17 años con permiso firmado
```

Explicación paso a paso: Traducimos la regla del español a lógica. "Tiene 18 o más Y está registrada" es `edad >= 18 && registrada`. "Tiene 16 o 17 Y trae permiso" lo escribimos como `edad >= 16 && permisoFirmado` (si tiene 18 o más, la primera parte ya lo cubre; nota que un chavo de 20 con permiso pero sin registro daría true en la segunda parte... ¿es correcto? Depende de la regla: si la regla dice que los de 16-17 con permiso pueden votar aunque no estén registrados, está bien; si no, habría que acotar con `edad <= 17`. Estos matices son decisiones de negocio: cuando programes reglas reales, pregunta los casos borde). Unimos ambas mitades con `||` porque con cumplir UNA basta. Los paréntesis dejan clarísimo el agrupamiento. Prueba con `edad=20, registrada=true, permisoFirmado=false`: `(true && true) || ... = true`. Con `edad=15, registrada=false, permisoFirmado=true`: `(false && false) || (false && true) = false`. Con `edad=16, registrada=false, permisoFirmado=false`: `(false) || (true && false) = false`. Tres pruebas, tres resultados correctos.

Ejercicio 3: Números del 1 al 50, pero solo múltiplos de 3 y 5 marcados (for + lógica)

Enunciado: Usa un ciclo for para imprimir los números del 1 al 50. Si el número es múltiplo de 3, imprime el número seguido de "(múltiplo de 3)". Si es múltiplo de 5, "(múltiplo de 5)". Si es múltiplo de ambos (o sea, de 15), "(múltiplo de 3 y 5)". Un número es múltiplo de otro si el `residuo%` es 0.

Solución:

```

for (let i = 1; i <= 50; i++) {
  let esMult3 = i % 3 === 0;
  let esMult5 = i % 5 === 0;

  if (esMult3 && esMult5) {
    console.log(i + " (múltiplo de 3 y 5)");
  } else if (esMult3) {
    console.log(i + " (múltiplo de 3)");
  } else if (esMult5) {
    console.log(i + " (múltiplo de 5)");
  } else {
    console.log(i);
  }
}

```

Explicación paso a paso: El for recorre del 1 al 50 ($i \leq 50$ para incluir el 50). En cada vuelta calculamos dos booleanos: `esMult3` ($i \% 3 === 0$?) y `esMult5` ($i \% 5 === 0$?). Guardarlos en variables con nombre hace el código legible. Luego la decisión: PRIMERO preguntamos si es múltiplo de ambos, porque si preguntáramos solo `esMult3` primero, el 15 caería ahí y nunca llegaría a la marca de "3 y 5". El orden de las condiciones importa cuando los casos se traslapan: del más específico al más general. Con $i=15$: `esMult3 true`, `esMult5 true`, entra al primer if: "15 (múltiplo de 3 y 5)". Con $i=9$: `true/false` -> "9 (múltiplo de 3)". Con $i=10$: `false/true` -> "10 (múltiplo de 5)". Con $i=7$: `false/false` -> "7". Este ejercicio es famoso en entrevistas de trabajo (es una variante del "FizzBuzz"): si lo dominas, ya tienes una historia que contar.

Ejercicio 4: Suma de pares con continue y conteo con break (control de ciclo)

Enunciado: Escribe un for del 1 al 100 que sume solo los números pares (usa `continue` para saltarte los impares). Además, si la suma supera 1000, detén el ciclo con `break` e imprime en qué número te detuviste. Al final imprime la suma total.

Solución:

```

let suma = 0;
let detenidoEn = 0;

for (let i = 1; i <= 100; i++) {
  // Saltamos los impares: esta vuelta no suma
  if (i % 2 !== 0) {
    continue;
  }
  suma = suma + i;

  // Si ya pasamos de 1000, no tiene caso seguir
  if (suma > 1000) {
    detenidoEn = i;
    break;
  }
}

console.log("Suma: " + suma);
console.log("Me detuve en el número: " + detenidoEn);

```

Explicación paso a paso: El ciclo va del 1 al 100. En cada vuelta, si i es impar ($i \% 2 !== 0$), `continue` salta el resto de la vuelta: ni suma ni revisa el límite. Solo los pares llegan a `suma = suma + i`. Después de sumar, revisamos si `suma > 1000`: si sí, guardamos el i actual en `detenidoEn` y `break` rompe el ciclo. Vamos a estimar dónde se detiene: la suma de pares $2+4+\dots+n$ con n par es aproximadamente $n^2/4$... mejor hagámoslo directo: $2+4+\dots+44 = 2 \times (1+2+\dots+22) = 2 \times 253 = 506$. Sumando 46: 552. Sumando 48: 600. Sumando 50: 650. Sumando 52: 702. Sumando 54: 756. Sumando 56: 812. Sumando 58: 870. Sumando 60: 930. Sumando 62: 992. Sumando 64: $1056 > 1000$. Se detiene en $i=64$ con `suma=1056`. Fíjate cómo `continue` y `break` trabajan en equipo: uno filtra vueltas, el otro termina el trabajo temprano. Este patrón (acumular hasta un límite) aparece en presupuestos, inventarios y paginación de datos.

Ejercicio 5: Patrón de for anidados (cuadrado hueco)

Enunciado: Dibuja con asteriscos un cuadrado hueco de 5×5 usando `for` anidados. El borde lleva `*`, el interior lleva espacios. Debe verse así: la primera y última fila son `"*****"`, y las filas 2 a 4 son `"* *"`.

Solución:


```

let tamaño = 5;

for (let fila = 1; fila <= tamaño; fila++) {
  let linea = "";
  for (let col = 1; col <= tamaño; col++) {
    // Es borde si es primera/última fila o primera/última columna
    let esBorde = fila === 1 || fila === tamaño || col === 1 || col === tamaño;
    if (esBorde) {
      linea = linea + "*";
    } else {
      linea = linea + " ";
    }
  }
  console.log(linea);
}

```

Explicación paso a paso: El ciclo exterior maneja las filas (1 a 5), el interior las columnas (1 a 5): 25 casillas en total. Para cada casilla decidimos si es borde: es borde si está en la primera fila (fila === 1) o la última (fila === tamaño) o la primera columna (col === 1) o la última (col === tamaño). Con || basta que UNA se cumpla: las esquinas cumplen dos a la vez, y está bien. Si es borde, agregamos *; si no, un espacio. Al terminar las columnas de una fila, imprimimos la línea completa y el exterior pasa a la siguiente fila, reiniciando linea = "". Fila 1: todas las columnas son borde -> "*****". Fila 2: col 1 borde (*), col 2-4 interior (espacios), col 5 borde (*) -> "* * * * *". Igual filas 3 y 4. Fila 5: "*****". Cuadrado hueco logrado. Este ejercicio combina todo: for anidados, comparaciones ===, lógica || y construcción de texto. Si lo resolviste sin ver la solución, ya piensas como programador.

Ejercicio 6: Adivina el número (reto integrador)

Enunciado: El programa "piensa" un número secreto del 1 al 20 (fíjalo tú en una variable). Tiene un máximo de 5 intentos (usa un for). En cada intento hay un "intento del jugador" (simúlalo con un array de 5 valores que tú definas). Si el intento es igual al secreto, imprime "¡Ganaste!" y termina con break. Si es menor, imprime "muy bajo"; si es mayor, "muy alto". Si se acaban los intentos, imprime "Perdiste, era X".

Solución:

```

let secreto = 13;
let intentos = [5, 18, 10, 13, 7]; // jugadas simuladas
let gano = false;

for (let i = 0; i < 5; i++) {
  let jugada = intentos[i];
  console.log("Intento " + (i + 1) + ": " + jugada);

  if (jugada === secreto) {
    console.log("¡Ganaste!");
    gano = true;
    break; // ya no necesitamos más intentos
  } else if (jugada < secreto) {
    console.log("Muy bajo");
  } else {
    console.log("Muy alto");
  }
}

if (!gano) {
  console.log("Perdiste, el número era " + secreto);
}

```

Explicación paso a paso: Fijamos el secreto en 13 y simulamos las jugadas con un array (en la unidad de arrays verás esto a fondo; por ahora créeme que `intentos[i]` saca la jugada número *i*). El `for` da máximo 5 vueltas. En cada vuelta comparamos: `===` para ganar, `<` para "muy bajo", `else` para "muy alto" (si no es igual ni menor, por descarte es mayor). Si gana, marcamos `gano = true` y `break` termina el ciclo: no tiene caso seguir jugando. Al salir del `for`, si `!gano` (nunca ganó), mostramos el mensaje de derrota con el número secreto. Sigue las jugadas: `5 < 13` "muy bajo"; `18 > 13` "muy alto"; `10 < 13` "muy bajo"; `13 === 13` "¡Ganaste!", `break`. Solo se usaron 4 intentos. Cambia los valores del array para probar la derrota (que ningún intento sea 13) y verás el mensaje final. Este mini-juego usa comparaciones, lógica, `for`, `break` y una bandera booleana (`gano`): es un resumen jugable de toda la unidad.

Resumen de la unidad

Vamos a cerrar con los puntos clave, esos que si los recuerdas dentro de un año, la unidad cumplió su misión.

Primero: comparar es preguntar y la respuesta siempre es true o false. Los operadores de comparación son `===` (igual estricto), `!==` (distinto estricto), `>` (mayor), `<` (menor), `>=` (mayor o igual) y `<=` (menor o igual). Usa siempre las versiones estrictas y evita `==` y `!=`, porque la conversión automática de tipos produce resultados tramposos como `0 == false` o `"" == false`.

Segundo: el `=` asigna, el `===` compara. Si tu condición siempre parece true, revisa que no hayas escrito un solo `=` por accidente. Es el error más común del mundo y a todos nos ha pasado.

Tercero: los operadores lógicos combinan preguntas. `&&` (Y) exige que todo sea true; `||` (O) se conforma con que algo sea true; `!` (NO) invierte el valor. Las tablas de verdad son tu herramienta para no perderte: con dos variables solo hay cuatro casos, escríbelos cuando dudes.

Cuarto: el cortocircuito hace que `&&` se detenga en el primer falsy y `||` en el primer truthy. Úsalo como guardián (usuario `&&` `hacerAlgo()`) y para valores por defecto (nombre `||` `"Invitado"`), pero recuerda que `0` y `""` son falsy y a veces son valores legítimos.

Quinto: la precedencia es `!`, luego `&&`, luego `||`, con las comparaciones antes que todo lo lógico. No la memorices como examen: usa paréntesis para que tu intención sea obvia.

Sexto: los rangos se escriben con `&&` explícito: `edad >= 18 && edad <= 65`. Nunca encadenes como en matemáticas (`18 <= edad <= 65` no funciona en JavaScript).

Séptimo: el ciclo for tiene tres partes: inicialización (`let i = 0`), condición (`i < 5`) y actualización (`i++`). La condición se revisa antes de cada vuelta; cuando es false, el ciclo termina. Cuenta desde cero por convención y decide conscientemente si el límite lleva `<` o `<=`.

Octavo: `break` rompe el ciclo por completo (ya terminé), `continue` salta solo la vuelta actual (esta no me interesa). No los confundas.

Noveno: los for anidados ejecutan el interior completo por cada vuelta del exterior; el total de vueltas se multiplica. Son la herramienta natural para tablas, cuadrículas y combinaciones.

Décimo: cuando un ciclo se porta raro, imprime el contador con `console.log(i)`. Nueve de cada diez bugs de ciclo se ven en los valores de `i`.

Y la idea que une todo: las comparaciones producen booleanos, la lógica los combina y los ciclos los obedecen. Dominar esta unidad es dominar el arte de que tu programa tome decisiones y repita trabajo sin cansarse. En la próxima unidad le sumamos el ciclo while y las funciones, y tu caja de herramientas va a estar casi completa para construir programas de verdad.

UNIDAD 4: Repetición y funciones

En la unidad anterior le enseñaste a tu programa a tomar decisiones y a repetir tareas con el ciclo `for`. Fue un gran salto: tu código dejó de ser una lista de instrucciones ciegas y se volvió algo que reacciona y que trabaja en serie. Pero todavía hay dos piezas enormes que te faltan para programar de verdad. La primera es el otro ciclo grande del lenguaje: el `while`, que repite mientras una condición sea verdadera sin que sepas de antemano cuántas vueltas va a dar. La segunda, y más importante, son las funciones: la herramienta que te permite empaquetar un pedazo de lógica con un nombre, usarlo cuantas veces quieras y no repetirlo nunca más. Si la unidad anterior fue sobre decisiones, esta es sobre organización y reutilización. Cuando la termines, vas a escribir programas que se ven profesionales: cortos, ordenados y fáciles de entender.

Déjame ponerte en contexto con una analogía que vas a recordar todo el curso. Imagina que estás construyendo una casa. En las primeras unidades aprendiste a hacer ladrillos (variables, tipos de datos) y a pegarlos (operadores). En la unidad anterior aprendiste a poner hileras de ladrillos rapidísimo (el `for`) y a decidir dónde va cada cosa (comparaciones y lógica). Ahora vas a aprender dos cosas: el `while` es como tener un ayudante que sigue pegando ladrillos hasta que tú le digas "ya basta", sin que le hayas dicho cuántos desde el inicio. Y las funciones son como tener moldes y herramientas: en lugar de tallar cada ventana a mano, haces un molde una vez y sacas todas las ventanas idénticas. El ayudante y los moldes no te hacen más fuerte; te hacen más eficiente. Eso es exactamente lo que `while` y las funciones hacen por tu código.

¿Por qué importa el `while` si ya tienes el `for`? Porque hay repeticiones donde no sabes cuántas vueltas necesitas. Piensa en un juego: "sigue jugando mientras el jugador quiera continuar". ¿Cuántas partidas va a jugar? No lo sabes; depende del jugador. Piensa en leer datos hasta encontrar una marca de fin, o en esperar a que el usuario escriba la contraseña correcta: no sabes cuántos intentos va a necesitar. El `for` es para cuando conoces el recorrido (del 1 al 10, cada elemento de una lista); el `while` es para cuando conoces la condición de salida pero no el número de vueltas ("mientras no sea correcto, repite"). Saber elegir entre los dos es una de las marcas de un programador que piensa, no solo que escribe.

Y las funciones... las funciones son el concepto más importante de esta unidad y uno de los más importantes de toda la programación. Todo programa grande está hecho de funciones. Cuando usas `console.log`, estás llamando a una función que alguien más escribió. Cuando una página web valida tu contraseña, hay funciones detrás. Cuando un videojuego mueve a tu personaje, son funciones. Una función es un bloque de código con nombre que hace una tarea

específica: recibe datos de entrada (parámetros), hace su trabajo y puede devolver un resultado (return). La magia está en la reutilización: escribes la lógica una vez y la usas cien veces, en cien lugares distintos, sin copiar y pegar nada.

Copiar y pegar código es el pecado original del programador principiante, y las funciones son la redención. Te cuento por qué es tan grave copiar y pegar: imagina que tienes la fórmula para calcular el IVA copiada en quince lugares de tu programa. Un día cambia la tasa del IVA (pasa en la vida real). Con código copiado, tienes que encontrar y corregir los quince lugares, y si se te olvida uno, tu programa calcula mal en silencio en ese rincón. Con una función `calcularIVA(precio)`, cambias UN solo lugar y los quince usos se corrigen solos. A esto le decimos el principio DRY: Don't Repeat Yourself (no te repitas). No es una sugerencia de estilo; es una estrategia de supervivencia para cuando tus programas crezcan.

Esta unidad está organizada en dos grandes bloques que se complementan. Primero, el ciclo `while` y su primo `do...while`: sintaxis, cómo se compara con el `for`, cuándo usar cada uno, y el tema serio de los ciclos infinitos (qué son, por qué pasan, cómo evitarlos y qué hacer si te topas con uno, porque te va a pasar y no es el fin del mundo). Segundo, las funciones a fondo: qué son conceptualmente, cómo se declaran y cómo se invocan (llaman), la diferencia entre parámetros y argumentos (dos palabras que todos confunden al inicio), los valores por defecto, el `return` y qué pasa cuando no hay `return`, el alcance de las variables (scope local vs global, uno de los temas que más confunde a los principiantes y que aquí vas a dominar), y la filosofía de las funciones como herramienta de reutilización: cómo partir un problema grande en funciones pequeñas.

Una advertencia de ritmo: esta unidad tiene dos temas que piden tipos distintos de atención. El `while` es mecánico: lo entiendes rápido si ya dominas el `for`, porque la idea de "repetir mientras" ya la traes. Las funciones, en cambio, piden un cambio de mentalidad: dejar de pensar en "líneas que se ejecutan de arriba a abajo" y empezar a pensar en "piezas con nombre que se usan cuando se necesitan". Ese cambio no pasa en una leída; pasa cuando escribes tus primeras diez o quince funciones y sientes en carne propia lo cómodo que es reutilizar. Así que con las funciones, más que con ningún otro tema del curso, te pido: escribe código. Inventa funciones tontas: una que salude, una que sume, una que convierta pesos a dólares. La teoría te da el mapa; solo escribiendo recorres el territorio.

También quiero adelantarte algo para que no te frustres: el alcance de variables (scope) es el tema donde más preguntas me hacen los estudiantes, y es normal. La idea de que una variable "existe" en un lugar del programa y "no existe" en otro suena rara al principio, como si las variables fueran fantasmas que aparecen y desaparecen. Pero tiene una lógica hermosa detrás: el alcance existe para protegerte de ti mismo, para que una variable que creaste para

un cálculo temporal no interfiera con otra que se llama igual en otro lado. Cuando entiendas el scope, vas a ver que no es una complicación del lenguaje: es una característica de seguridad. Y te voy a dar una regla de oro que simplifica el 90% de los casos.

Otro tema que genera nudos en la cabeza: la diferencia entre parámetros y argumentos. Te lo adelanto para que llegues preparado: los parámetros son los nombres que pones en la definición de la función (los "espacios vacíos" que la función espera), y los argumentos son los valores reales que le pasas cuando la llamas (lo que llena esos espacios). Es como una receta: la receta dice "ingredientes: harina, huevos" (parámetros); cuando cocinas, usas 500 gramos de harina y 3 huevos de verdad (argumentos). La receta es el molde, la cocinada es el uso. Con esa analogía en la mente, la sección te va a entrar suavemente.

Hablemos de cómo se conecta esta unidad con el resto del curso, porque aquí es donde todo empieza a unirse de verdad. En las próximas unidades vas a ver arrays y objetos, y adivina qué: casi todo lo que hagas con ellos será dentro de funciones y ciclos. Vas a escribir funciones que reciben arrays, funciones que devuelven objetos, ciclos while que recorren datos hasta encontrar algo. El DOM (la unidad donde JavaScript toca la página web) es básicamente funciones que reaccionan a eventos: "cuando el usuario haga clic, ejecuta esta función". Los eventos son funciones esperando su momento. Sin funciones, no hay interactividad; sin interactividad, no hay páginas web modernas. Así que lo que aprendas aquí no es un tema más: es el pegamento de todo lo que sigue.

Déjame contarte una anécdota de funciones para que veas el antes y el después. Un conocido tenía un programa que calculaba el precio final de productos con impuestos y descuentos. La fórmula estaba copiada en doce lugares. Un día el gobierno cambió un impuesto y él actualizó once lugares. El doceavo se le olvidó. Durante tres semanas, una categoría de productos se vendió con el impuesto viejo. Nadie lo notó hasta que contabilidad encontró una diferencia de miles de pesos. El error no fue matemático: fue arquitectónico. Si la fórmula hubiera estado en una función, el cambio habría sido en un solo lugar y el bug no habría existido. Cada vez que sientas la tentación de copiar y pegar, acuérdate de esta historia y escribe una función.

Y una anécdota de while, más ligera. Un compañero hizo un programa que leía mensajes de un chat hasta encontrar la palabra "fin". Usó un while perfecto... excepto que olvidó actualizar la variable dentro del ciclo. El programa se quedó leyendo el mismo mensaje eternamente, el ventilador de su laptop empezó a sonar como turbina de avión y tuvo que cerrar todo a la fuerza. Se rió después, pero aprendió la lección más importante del while: el ciclo solo termina si algo dentro de él acerca la condición a su fin. Esa frase te la voy a repetir varias veces en la unidad hasta que te la sepas de memoria, porque es la ley fundamental del while.

Antes de entrar a la materia, un repaso de conexión con la unidad anterior. Allá aprendiste que el `for` repite un número conocido de veces con su contador `i`. El `while` hace lo mismo pero sin contador obligatorio: solo tiene una condición entre paréntesis y un bloque. De hecho, todo `for` se puede escribir como `while` y viceversa; son dos formas de expresar la repetición. La diferencia es de intención: el `for` dice "voy a recorrer esto", el `while` dice "voy a insistir hasta que pase aquello". Y las funciones usan todo lo anterior por dentro: una función puede tener comparaciones, lógica, `for`, `while`, todo. Las funciones no son un tema aislado; son el contenedor donde vive todo lo demás.

También conviene que recuerdes el concepto de booleano, porque el `while` vive de booleanos igual que el `for`: su condición se evalúa antes de cada vuelta y el ciclo sigue mientras sea `true`. Si dominaste las comparaciones y los operadores lógicos de la unidad anterior, el `while` te va a parecer pan comido. Y si algo de la unidad anterior te quedó flojo, esta es una buena noticia disfrazada: vas a repasar comparaciones y lógica sin darte cuenta, porque las vas a usar en cada ejemplo de `while` y en cada función.

Un último consejo práctico: a partir de esta unidad, tus programas van a empezar a tener varias piezas (funciones, ciclos, condiciones) y va a ser más fácil perderte. Acostúmbrate a probar por partes: primero escribe una función, pruébala sola con valores de ejemplo, y cuando funcione, úsala en el programa grande. A esto le decimos probar de forma incremental, y es la diferencia entre pasar una tarde feliz programando y pasar una noche miserable buscando un bug en cien líneas que escribiste de un jalón. Los profesionales no escriben todo de golpe; construyen por piezas y prueban cada pieza. Hazlo tú también desde ahora.

¿Listo? Vamos a darle a tu programa un ayudante incansable (el `while`) y una caja de herramientas con nombre (las funciones). Empezamos con la repetición sin contador fijo.

Y antes de arrancar, un consejo que te va a acompañar toda tu carrera: cuando algo no funcione, reduce el problema. Si tu `while` se porta raro, pruébalo con tres vueltas a mano en papel. Si tu función devuelve algo extraño, llámala sola con valores simples antes de usarla en el programa grande. Los bugs se esconden en lo complejo y se revelan en lo simple. Los programadores novatos agregan código para arreglar; los experimentados quitan código hasta encontrar al culpable. Aprende a quitar desde ahora. El que sabe simplificar, sabe programar. Y el que programa todos los días, avanza rapidísimo.

Explicación detallada

El ciclo while: repetir hasta que pase algo

El while es el ciclo de la insistencia. Su sintaxis es más simple que la del for: solo una condición entre paréntesis y un bloque entre llaves. Sin inicialización obligatoria, sin actualización obligatoria en el encabezado. Se lee así: "mientras esta condición sea verdadera, repite este bloque".

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Esto imprime 0, 1, 2, 3, 4, igual que el for equivalente. Vamos a seguir el flujo: antes de cada vuelta, JavaScript evalúa $i < 5$. Primera vuelta: i vale 0, $0 < 5$ es true, imprime 0, luego $i++$ lo sube a 1. Segunda: $1 < 5$ true, imprime 1, i sube a 2... Quinta vuelta: i vale 4, imprime 4, i sube a 5. Sexta revisión: $5 < 5$ es false, el ciclo termina sin ejecutar el bloque. ¿Ves la diferencia con el for? Aquí TÚ te encargas de crear la variable antes ($\text{let } i = 0$) y de actualizarla dentro del bloque ($i++$). El while no te ayuda con eso; te da libertad total, y con la libertad viene la responsabilidad.

Esa responsabilidad es exactamente el tema central del while: la condición tiene que poder volverse false en algún momento, y eso depende de que algo dentro del bloque la acerque a su fin. Si olvidas el $i++$ en el ejemplo anterior, i siempre vale 0, $i < 5$ siempre es true, y tienes un ciclo infinito: el programa repite eternamente, imprimiendo ceros hasta que lo detengas a la fuerza. En el navegador, la pestaña se congela; en Node, tienes que presionar Ctrl+C. No se rompe nada permanentemente, pero es desagradable y, si el ciclo infinito está en un servidor, puede ser grave. Por eso la ley fundamental del while, repítela conmigo: el ciclo solo termina si algo dentro de él acerca la condición a su fin.

Vamos a ver los usos naturales del while, esos donde brilla y el for se sentiría forzado. El primero: repetir hasta lograr algo, sin saber cuántos intentos tomará. Ejemplo clásico: pedir la contraseña hasta que sea correcta.


```

let password = "";
// En un programa real, esto vendría de un prompt o un formulario.
// Aquí simulamos intentos con una lista.
let intentos = ["gato", "perro123", "secreto42"];
let indice = 0;

while (password !== "secreto42") {
  password = intentos[indice];
  console.log("Probando: " + password);
  indice++;
}
console.log("¡Acceso concedido!");

```

El flujo: password empieza vacío, "" !== "secreto42" es true, entramos. Tomamos intentos[0] ("gato"), lo probamos, subimos el índice. "gato" !== "secreto42" true, seguimos. "perro123" tampoco, seguimos. "secreto42" === "secreto42", la condición password !== "secreto42" se vuelve false, salimos. "¡Acceso concedido!". ¿Cuántas vueltas dio? Tres, pero el código no lo sabía de antemano: el while se adaptó a los datos. Con un for habrías tenido que saber cuántos intentos hay; con while, el ciclo sigue hasta que se cumple la meta. Esa es su esencia: orientado a la meta, no al conteo.

Ojo con un detalle importante de este ejemplo: si la lista de intentos se acabara sin contener la contraseña correcta, el índice seguiría creciendo, intentos[indice] sería undefined, y el ciclo sería infinito (undefined nunca es "secreto42"). En un programa real, pondrías un límite de intentos o una condición de salida extra. Este es un patrón profesional: los while que dependen de datos externos casi siempre llevan una segunda condición de seguridad, como while (password !== correcta && intentos < 5). El && que aprendiste en la unidad anterior te protege aquí.

Segundo uso natural: procesar datos hasta encontrar una marca de fin. Imagina que lees mensajes y paras cuando aparezca "FIN". Tercero: esperar a que algo cambie, como en juegos ("mientras el jugador tenga vidas, sigue el juego"). Cuarto: contadores con pasos irregulares, donde el avance no es constante. En todos estos casos, la pregunta no es "¿cuántas veces?" sino "¿hasta cuándo?".

Comparación directa for vs while, porque elegir bien es una habilidad. Usa for cuando: sabes cuántas vueltas (o puedes calcularlo), recorres un rango de números, o visitas cada elemento de una colección. Usa while cuando: repites hasta que se cumpla una condición, no sabes cuántas vueltas serán, o la condición de salida depende de algo que cambia dentro del ciclo de forma no predecible. ¿Se pueden intercambiar? Casi siempre sí, pero el código queda más

claro si usas el que expresa tu intención. Un for que simula un while se ve raro; un while que simula un for se ve innecesariamente manual. La legibilidad es una característica del programa, no un lujo.

¿Sabías que...? El while es más antiguo como concepto que el for con contador. Los primeros lenguajes tenían "repite mientras" antes de tener la sintaxis compacta de tres partes. El for nació después como un atajo para el caso común de contar. Cuando usas while, estás usando la forma más pura y original de la repetición en programación.

Hablemos del peligro real: el ciclo infinito. Ya vimos cómo se provoca (condición que nunca se vuelve false), pero vamos a catalogar las causas comunes para que las reconozcas. Causa uno: olvidar actualizar la variable de control (el `i++` que falta). Causa dos: actualizarla en la dirección equivocada (`i--` cuando la condición necesita que `i` crezca). Causa tres: condición mal escrita, como `while (i = 5)` con un solo `=` (asignación en lugar de comparación: `i` siempre vale 5, que es truthy, ciclo eterno). Causa cuatro: depender de algo externo que nunca cambia, como esperar a que el usuario haga algo que tu programa no le permite hacer. Causa cinco: errores de lógica donde la condición de salida es inalcanzable, como `while (x > 0)` donde `x` solo aumenta.

¿Cómo te das cuenta de que estás en un ciclo infinito? Síntomas: la página se congela y no responde, el ventilador se acelera, la consola escupe miles de líneas iguales, el programa nunca termina. ¿Qué hacer? En el navegador: cierra la pestaña (a veces hay que esperar unos segundos a que el navegador te ofrezca "detener la página"). En la consola del navegador, un ciclo infinito con `console.log` puede congelar las herramientas de desarrollo; ciérralas y recarga. En Node.js: Ctrl+C. Y lo más importante: no entres en pánico. Un ciclo infinito no daña tu computadora; solo desperdicia unos segundos de CPU. Todos los programadores provocan decenas en su carrera. Es un rito de iniciación, no una tragedia.

¿Cómo prevenirlos? Tres hábitos. Uno: antes de escribir el bloque, pregúntate "¿qué hace que esto termine?" y asegúrate de que ese algo esté dentro del bloque. Dos: en ciclos que dependen de datos externos, agrega una condición de seguridad con `&&` (límite de vueltas o de tiempo). Tres: durante el desarrollo, pon un `console.log` de la variable de control en las primeras pruebas para ver que avanza como esperas; cuando funcione, lo quitas. Estos hábitos te van a ahorrar más tiempo que cualquier otra cosa en esta sección.

Cuidado con... el while con la condición ya false desde el inicio: el bloque nunca se ejecuta, ni una sola vez. A veces es lo que quieres (si no hay nada que procesar, no haces nada), pero a veces es un bug: esperabas al menos una vuelta. Si tu while "no hace nada", revisa el valor inicial de la condición antes de la primera vuelta. Para los casos donde necesitas garantizar al menos una ejecución existe do...while, que viene ahora.

do...while: primero actúa, luego pregunta

El do...while es el primo del while con una diferencia crucial: el bloque se ejecuta AL MENOS UNA VEZ, porque la condición se revisa AL FINAL de cada vuelta, no al inicio. Su sintaxis:

```
let numero;
do {
  numero = 7; // en un programa real, aquí pedirías el dato al usuario
  console.log("Procesando: " + numero);
} while (numero < 0);
```

Flujo: entra al bloque sin preguntar nada, procesa el 7, y AL FINAL revisa `numero < 0`: `7 < 0` es false, termina. El bloque se ejecutó una vez aunque la condición ya era false desde el principio. Con un while normal, el bloque no se habría ejecutado nunca.

¿Cuándo sirve esto? Cuando la primera ejecución no depende de la condición porque es la que GENERA el valor que la condición va a revisar. El ejemplo perfecto es pedir datos al usuario: necesitas pedir el dato al menos una vez para tener algo que validar. Con while tendrías que pedirlo una vez antes del ciclo y luego otra vez dentro (código duplicado); con do...while lo pides dentro y la condición al final decide si repites.

```
let edad;
do {
  edad = 25; // simulamos que el usuario escribe 25
  console.log("Edad capturada: " + edad);
} while (edad < 0 || edad > 120);
console.log("Edad válida: " + edad);
```

Aquí la validación (edad entre 0 y 120) solo tiene sentido después de tener un valor. El do...while expresa exactamente eso: "hazlo, y si no quedó bien, repite". Es el ciclo de "inténtalo al menos una vez".

Comparación rápida: while revisa antes (cero o más vueltas), do...while revisa después (una o más vueltas). Si no estás seguro de cuál usar, pregúntate: "¿tiene sentido ejecutar el bloque al

menos una vez aunque la condición ya sea falsa?" Si sí, `do...while`. Si no, `while`. En la práctica, el `while` es mucho más común; el `do...while` aparece en validaciones de entrada y menús interactivos. Conócelo, úsalo cuando toque, no lo fuerces donde no va.

Un detalle de sintaxis que confunde: el `do...while` termina con punto y coma después del `while` (condición); Es el único ciclo que lleva punto y coma al final. Si lo olvidas, JavaScript a veces lo perdona y a veces no; mejor escríbelo siempre.

¿Sabías que...? Algunos programadores veteranos casi nunca usan `do...while` y prefieren simularlo con `while (true) + break`. Por ejemplo: `while (true) {pedirDato(); if (datoValido) break;}`. Es válido y funciona, pero el `do...while` expresa la intención más claramente. En este curso preferimos la forma que se lee mejor, no la más astuta.

Funciones: el arte de no repetirse

Entramos al plato fuerte. Una función es un bloque de código con nombre que realiza una tarea. La defines una vez y la usas (la "llamas" o "invocas") cuantas veces quieras, desde donde quieras. Piensa en las funciones como recetas: la receta de los hot cakes está escrita una vez en tu recetario, pero la "ejecutas" cada domingo. La definición es la receta; la invocación es cocinar.

La sintaxis básica tiene tres partes: la palabra `function`, el nombre que tú eliges, paréntesis (que pueden llevar parámetros) y el bloque con el trabajo.

```
function saludar() {  
  console.log("¡Hola! ¿Cómo estás?");  
}
```

Esto DEFINE la función, pero ojo: definir no es ejecutar. Si corres solo esto, no se imprime nada. La función es como una receta guardada: existe, pero nadie está cocinando. Para usarla, la invocas escribiendo su nombre seguido de paréntesis:

```
saludar(); // ¡Hola! ¿Cómo estás?  
saludar(); // ¡Hola! ¿Cómo estás? (otra vez, sin reescribir nada)
```

Cada vez que escribes `saludar()`, JavaScript salta al bloque de la función, ejecuta sus líneas y regresa donde estaba. Puedes llamarla cero veces (entonces fue inútil definirla), una vez o mil veces. El poder está en que el código vive en UN solo lugar: si mañana quieres que el saludo diga "¡Hola! ¿Qué onda?", lo cambias una vez en la función y todos los saludos del programa cambian. Eso es reutilización, y es la razón de ser de las funciones.

Analogía del mundo real que me gusta mucho: las funciones son como los botones de una calculadora. El botón de raíz cuadrada tiene un mecanismo interno complicado, pero tú solo lo presionas y te da el resultado. No necesitas saber cómo calcula la raíz; solo necesitas saber qué botón presionar y qué número darle. Las funciones bien hechas son así: por fuera simples (nombre claro, datos de entrada, resultado), por dentro pueden ser tan complejas como necesiten. A esto le decimos abstracción: usar algo por lo que hace, no por cómo lo hace.

Vamos a ver por qué las funciones cambian tu forma de programar con un ejemplo del pecado original: código repetido sin funciones vs con funciones. Tarea: mostrar el área de tres terrenos rectangulares.

Sin funciones (código malo, repetido):

```
let base1 = 10, altura1 = 5;
console.log("Área: " + base1 * altura1);
let base2 = 7, altura2 = 3;
console.log("Área: " + base2 * altura2);
let base3 = 12, altura3 = 8;
console.log("Área: " + base3 * altura3);
```

Funciona, pero la fórmula $\text{base} * \text{altura}$ está escrita tres veces y el mensaje otras tres. Si hay que cambiar el mensaje o la fórmula, son seis ediciones. Con función:

```
function areaRectangulo(base, altura) {
  return base * altura;
}

console.log("Área: " + areaRectangulo(10, 5));
console.log("Área: " + areaRectangulo(7, 3));
console.log("Área: " + areaRectangulo(12, 8));
```

La lógica vive en un solo lugar. Tres usos, cero repetición. Y si mañana el área necesita redondeo, lo agregas en la función y listo. Esta es la diferencia entre un programa de juguete y un programa mantenible.

Hablemos de nombres de funciones, porque un buen nombre vale oro. Reglas: usa verbos o frases verbales (calcular, obtener, validar, mostrar, convertir), porque las funciones HACEN cosas. Sé específico: calcularIVA es mejor que calcular, que es mejor que hacer. Usa camelCase como en variables (la convención de JavaScript). Evita nombres genéricos como funcion1 o proceso: dentro de dos semanas no vas a saber qué hacen. Un buen nombre hace que el código se lea como español: si (esMayorDeEdad(edad)) se entiende sin comentarios.

Nombrar bien es una habilidad que se entrena; cada vez que dudes entre dos nombres, elige el más descriptivo.

Parámetros y argumentos: los datos que entran

Ya viste que `areaRectangulo(base, altura)` recibe dos valores. Vamos a precisar la terminología porque aquí es donde todos se confunden al inicio. Los **PARÁMETROS** son los nombres que aparecen en la **DEFINICIÓN** de la función: son como los espacios vacíos, los "moldes" que dicen qué datos espera la función. Los **ARGUMENTOS** son los valores reales que pasas en la **INVOCACIÓN**: lo que llena esos espacios. Receta: "harina, huevos" en la receta son parámetros; los 500 gramos y los 3 huevos que usas al cocinar son argumentos.

```
function presentar(nombre, edad) { // nombre y edad son PARÁMETROS
  console.log("Me llamo " + nombre + " y tengo " + edad + " años");
}

presentar("Ana", 25); // "Ana" y 25 son ARGUMENTOS
presentar("Luis", 30); // otros argumentos, misma función
```

En la llamada `presentar("Ana", 25)`, JavaScript asigna "Ana" al parámetro `nombre` y 25 al parámetro `edad`, como si hiciera `let nombre = "Ana"` y `let edad = 25` al entrar a la función. Luego ejecuta el bloque con esos valores. En la segunda llamada, los mismos parámetros reciben valores distintos. Los parámetros son variables temporales: nacen cuando la función es invocada y mueren cuando termina. Esto conecta con el alcance, que veremos en detalle más adelante.

Los parámetros hacen a las funciones flexibles: sin ellos, `saludar()` siempre dice lo mismo; con un parámetro `nombre`, `saludar("Ana")` y `saludar("Luis")` personalizan el mensaje. Una función sin parámetros es como una máquina con un solo botón que siempre hace lo mismo; una con parámetros es una máquina con perillas que se adapta. Casi todas las funciones útiles tienen parámetros.

¿Qué pasa si llamas a la función con menos argumentos de los que espera? Los parámetros faltantes valen `undefined`. Ejemplo: `presentar("Ana")` sin edad imprime "Me llamo Ana y tengo undefined años". No da error, pero el resultado es raro. ¿Y si pasas MÁS argumentos de los esperados? JavaScript los ignora silenciosamente (en realidad los guarda en un objeto especial llamado `arguments`, pero eso es tema avanzado). Moraleja: cuenta tus argumentos; que coincidan con los parámetros es tu responsabilidad.

Valores por defecto: puedes darle a un parámetro un valor que se usa cuando no le pasan argumento. Es como decir "si no me dices lo contrario, asumo esto".

```
function saludar(nombre = "amigo") {  
  console.log("¡Hola, " + nombre + "!");  
}  
  
saludar("Ana"); // ¡Hola, Ana!  
saludar();      // ¡Hola, amigo! (se usó el valor por defecto)
```

Esto es comodísimo para opciones: `function conectar(url, reintentos = 3)` le da un comportamiento estándar a quien no especifica, pero permite personalizar a quien sí. Los valores por defecto van al final de la lista de parámetros (un parámetro con defecto seguido de uno sin defecto es legal pero confuso; evita hacerlo). Y ojo: el valor por defecto solo se usa cuando el argumento es `undefined`; si pasas `null` a propósito, se queda `null`. Detalle fino, pero bueno saberlo.

Pregunta frecuente: "¿cuántos parámetros debe tener una función?" Los que necesite, pero hay una guía práctica: si una función tiene más de 3 o 4 parámetros, probablemente está haciendo demasiadas cosas o conviene agrupar los datos en un objeto (tema de la unidad de objetos). Las funciones con muchos parámetros son difíciles de llamar correctamente porque hay que recordar el orden. Cuando veas `function registrar(nombre, correo, clave, edad, ciudad, pais, telefono)`, sospecha: esa función quiere ser varias funciones o recibir un objeto.

return: los datos que salen

Si los parámetros son los datos que entran, `return` es el dato que sale. Una función puede devolver un valor a quien la llamó con la palabra `return`, y ese valor se puede guardar en una variable, imprimir o usar en otra operación.

```
function sumar(a, b) {  
  let resultado = a + b;  
  return resultado; // devolvemos el resultado a quien llamó  
}  
  
let total = sumar(5, 3); // total vale 8  
console.log(total);      // 8  
console.log(sumar(10, 20) * 2); // 60: el return se usa directo en la operación
```

Piensa en return como la "respuesta" de la función. Le preguntas "¿cuánto es 5 + 3?", te responde 8. Sin return, la función hace su trabajo interno pero no te entrega nada (hablaremos de eso en un momento). Con return, la función se convierte en una pieza que produce valores, y esos valores alimentan el resto del programa: total = sumar(5,3); promedio = sumar(a,b) / 2; mensajes con "El total es"+ sumar(x, y).

Detalles cruciales del return. Uno: cuando JavaScript encuentra un return, la función TERMINA inmediatamente; lo que esté después no se ejecuta. Esto permite "salidas tempranas": revisar un caso especial al inicio y devolver el resultado sin anidar todo lo demás.

```
function dividir(a, b) {  
  if (b === 0) {  
    return "Error: no se puede dividir entre cero";  
  }  
  return a / b; // solo llegamos aquí si b no es cero  
}
```

Dos: una función puede tener varios return en distintos caminos, pero solo se ejecuta uno por invocación (el primero que se alcance). Tres: return sin valor (solo return;) termina la función y devuelve undefined; se usa para salir temprano cuando no hay nada que devolver. Cuatro: si la función termina sin ningún return, devuelve undefined automáticamente. Ejemplo:

```
function mostrarMensaje(texto) {  
  console.log(texto);  
  // no hay return  
}  
  
let r = mostrarMensaje("hola"); // imprime "hola"  
console.log(r); // undefined: la función no devolvió nada
```

Esto distingue dos tipos de funciones: las que CALCULAN algo (devuelven un valor con return: sumar, calcularIVA, esMayorDeEdad) y las que HACEN algo (efecto visible como imprimir o modificar la página, sin return: mostrarMensaje, guardarDatos). Ambas son válidas y necesarias. El error típico es confundirlas: llamar a una función que imprime esperando que devuelva el valor, o viceversa. Cuando uses una función, pregúntate: "¿esta devuelve algo o solo hace algo?" Y cuando escribas una, decide conscientemente cuál de las dos es.

¿Sabías que...? En matemáticas, las funciones siempre devuelven un valor ($f(x) = \dots$). En programación, las funciones que no devuelven nada también son funciones; la diferencia es que las usas por su efecto, no por su resultado. Esta flexibilidad es lo que las hace tan útiles

para programas interactivos: la función que se ejecuta cuando haces clic en un botón normalmente no devuelve nada, solo hace cosas (cambia la página, guarda datos).

Cuidado con... poner código después del return esperando que se ejecute. No se ejecuta nunca; es "código muerto". Algunos editores te lo marcan en gris. Si ves código gris después de un return, muévelo antes o elimínalo.

Cuidado con... el return en la misma línea vs separado. En JavaScript, return seguido de un salto de línea puede comportarse raro por la inserción automática de punto y coma (el famoso ASI): `return \n valor` puede interpretarse como `return;` (devuelve undefined). Regla simple: escribe return y el valor en la misma línea, siempre. Es uno de esos detalles del lenguaje que muerden una vez y no se olvidan.

Alcance (scope): dónde vive cada variable

Llegamos al tema que más confunde a los principiantes y que, una vez entendido, te hace sentir que subiste de nivel. El alcance (scope) es la zona del programa donde una variable existe y puede usarse. Hay dos alcances básicos: global y local.

Una variable GLOBAL se declara fuera de cualquier función y existe en todo el programa: cualquier función puede leerla y modificarla.

```
let impuesto = 0.16; // global: vive fuera de todo

function precioConImpuesto(precio) {
  return precio * (1 + impuesto); // la función puede usar la global
}

console.log(precioConImpuesto(100)); // 116
```

Una variable LOCAL se declara dentro de una función (incluyendo los parámetros) y solo existe dentro de esa función. Fuera de ella, es como si no existiera: intentar usarla da error.

```
function calcular() {
  let temporal = 42; // local: nace y muere dentro de calcular
  return temporal * 2;
}

console.log(calcular()); // 84
console.log(temporal); // ERROR: temporal no existe aquí afuera
```

¿Por qué existe esta restricción? Por protección. Imagina un programa con cincuenta funciones; si todas las variables fueran globales, cualquier función podría cambiar accidentalmente la variable de otra. Sería un caos: cambiarías algo en un lugar y se rompería algo en otro sin relación aparente. El alcance local es como tener cuartos separados en una casa: lo que pasa en la cocina no afecta lo de la recámara. Cada función es un cuarto con sus propias variables, y eso te permite usar nombres simples como `total`, `i` o `resultado` en muchas funciones sin que choquen entre sí.

De hecho, puedes tener dos variables con el MISMO nombre en funciones distintas y no pasa nada: son variables diferentes que viven en cuartos diferentes.

```
function funcionA() {  
  let x = 10;  
  return x;  
}  
function funcionB() {  
  let x = 99; // otra x, en otro cuarto, sin conflicto  
  return x;  
}
```

La `x` de `funcionA` y la `x` de `funcionB` no se conocen ni se afectan. Esto es liberador: no tienes que inventar nombres únicos para todo el programa.

Ahora, la regla de oro del scope que te prometí: declara tus variables lo más CERCA posible de donde las usas, y prefiere locales sobre globales. Las globales úsalas solo para valores verdaderamente compartidos por todo el programa (como configuración: la tasa de impuesto, la URL de la API). Todo lo demás, local dentro de su función. Si sigues esta regla, el 90% de los problemas de alcance desaparecen antes de nacer.

Hay un caso tramposo que debes conocer: si dentro de una función asignas un valor a una variable SIN declararla con `let` o `const`, JavaScript (en modo no estricto) la crea como global automáticamente. Es un bug clásico y silencioso:

```
function truco() {  
  valor = 100; // ¡SIN let! Se crea una global por accidente  
}  
truco();  
console.log(valor); // 100: la "local" se fugó a global
```

Esto se llama "fuga de variable" y es de las peores: el programa funciona "bien" pero contamina el alcance global, y meses después otra función usa valor sin saber que `truco()` la

creó. La solución es simple y absoluta: declara SIEMPRE tus variables con `let` o `const`. Sin excepciones. Si tu editor tiene modo estricto ("use strict"), este error se convierte en error visible en lugar de fuga silenciosa; úsalo cuando puedas.

Otro concepto que sale del alcance: el sombreado (shadowing). Si declaras una local con el mismo nombre que una global, dentro de la función la local "tapa" a la global: la función ve su propia variable, no la de afuera.

```
let nombre = "Global";

function fiesta() {
  let nombre = "Local"; // tapa a la global dentro de fiesta
  console.log(nombre); // "Local"
}

fiesta();
console.log(nombre); // "Global": afuera no cambió nada
```

El sombreado no es un error, pero puede confundir: lees `nombre` dentro de la función y crees que es la global. Por eso la buena práctica es evitar reutilizar nombres de globales como locales; si necesitas la global, úsala directo sin redeclararla.

Pregunta frecuente: "¿los parámetros son locales?" Sí. Los parámetros son variables locales que nacen con los valores de los argumentos. Por eso puedes llamarlos como quieras sin miedo a choques.

Pregunta frecuente: "¿puede una función usar variables globales?" Sí, puede leerlas y modificarlas. Pero modificar globales desde funciones es una mala práctica (efectos secundarios sorpresa): si `precioConImpuesto` cambiara la tasa de impuesto global, otras funciones se verían afectadas sin saber por qué. Lee globales cuando sean configuración; evita modificarlas desde funciones. Si una función necesita cambiar algo, que lo devuelva con `return` y que quien la llama decida qué hacer con el resultado.

Anécdota de scope para que se te grabe: una estudiante tenía un `for` con `let i` dentro de una función y, fuera de la función, otro `for` también con `let i`. Le preocupaba que chocaran. No chocan: cada `i` vive en su cuarto. Pero cuando cambió uno a `var` por un tutorial viejo, la `i` se fugó al alcance de la función y empezó a interferir. Moraleja doble: `let` respeta los cuartos, y no mezcles `var` en tu código moderno. En este curso, `let` y `const` siempre.

Funciones como herramienta de reutilización: partir problemas grandes

Ya sabes la mecánica; ahora la filosofía. La reutilización no es solo "no copiar y pegar": es una forma de pensar los problemas. Cuando te enfrentas a algo grande ("hacer un sistema de calificaciones"), no lo escribes de un jalón: lo PARTES en piezas pequeñas, cada pieza es una función, y luego las combinas. A esto le decimos descomposición, y es la habilidad más importante del programador profesional, más que memorizar sintaxis.

Ejemplo: programa que procesa calificaciones de un grupo. En lugar de una masa de código, lo partimos:

```
function promedio(lista) { /* ... */ }
function maxima(lista) { /* ... */ }
function aprobados(lista) { /* ... */ }
function reporte(lista) {
  // reporte USA a las otras tres: compone piezas
  console.log("Promedio: " + promedio(lista));
  console.log("Máxima: " + maxima(lista));
  console.log("Aprobados: " + aprobados(lista));
}
```

Cada función hace UNA cosa y la hace bien. reporte no calcula nada: orquesta. Si el cálculo del promedio tiene un bug, lo arreglas en promedio y todo lo que la usa se beneficia. Si quieres un reporte nuevo, reutilizas las piezas. Esta es la arquitectura de los programas reales: pirámides de funciones pequeñas donde las de arriba usan a las de abajo.

Principios para partir bien. Uno: una función, una tarea. Si tu función se llama procesarDatosYEnviarCorreoYGuardarEnBase, son tres funciones disfrazadas de una. Dos: nombres que revelan la intención; si necesitas un comentario para explicar qué hace la función, probablemente el nombre es malo. Tres: las funciones pequeñas se prueban fácil; una función de 10 líneas se verifica en un minuto, una de 100 líneas es un misterio. Cuatro: no te obsesiones con el tamaño perfecto; "pequeña y con un solo propósito" es la guía, no una ley.

¿Sabías que...? Hay un chiste entre programadores: las dos cosas más difíciles en computación son invalidar cachés, nombrar cosas y los errores off-by-one. Nombrar funciones es genuinamente difícil porque un buen nombre condensa el propósito. No te frustres si tardas en nombrar; es señal de que estás pensando bien.

Cuidado con... las funciones que hacen "magia" con globales: la función que lee cinco globales y modifica otras tres es imposible de probar sola, porque depende de todo el programa. Las mejores funciones son las "puras": reciben todo por parámetros y devuelven el resultado por

return, sin tocar nada externo. sumar(a, b) es pura y predecible; una función que suma usando globales es frágil. Tiende a lo puro siempre que puedas.

Mini-reto de diseño: piensa en el programa del juego de adivinanza del ejercicio 6 de la unidad anterior. ¿Cómo lo partirías en funciones? Una propuesta: generarSecreto() crea el número, pedirJugada() obtiene el intento, evaluarJugada(secreto, jugada) devuelve "muy bajo"/"muy alto"/"ganaste", y jugar() orquesta el ciclo. Cuatro funciones pequeñas en lugar de una masa. Si lo ves claro, ya estás pensando en funciones.

Para cerrar la explicación, una idea que une los dos bloques de la unidad: el while y las funciones se usan juntos constantemente. Una función que busca algo en datos usa un while por dentro; un while que procesa elementos llama funciones en cada vuelta. Y ambos usan comparaciones y lógica de la unidad anterior en sus condiciones. Tu caja de herramientas ya tiene: guardar datos (variables), calcular (operadores), decidir (comparaciones y lógica), repetir de forma contada (for), repetir hasta lograrlo (while) y empaquetar lógica reutilizable (funciones). Con esto ya puedes escribir programas que resuelven problemas reales. Lo que sigue (arrays, objetos, DOM) es más material para meter en estas herramientas, no herramientas nuevas. Vas muy bien.

Más profundidad: while con centinelas, contadores y decisiones

Vamos a profundizar en patrones típicos del while que te vas a encontrar en programas reales. Uno de los más comunes es el valor centinela: un valor especial que marca "aquí se acaba". Por ejemplo, procesar edades hasta que aparezca un -1 (que significa "ya no hay más datos").

```
let edades = [20, 25, -1, 30]; // el -1 es el centinela: ahí paramos
let i = 0;
let suma = 0;
let cantidad = 0;

while (edades[i] !== -1) {
  suma = suma + edades[i];
  cantidad++;
  i++;
}

console.log("Promedio: " + suma / cantidad); // (20+25)/2 = 22.5
```

El ciclo avanza mientras el valor actual NO sea el centinela. En cuanto aparece -1, la condición `edades[i] !== -1` se vuelve false y paramos, sin procesar el -1 ni lo que venga después (el 30 se

ignora, porque los datos "reales" terminaron en el centinela). Los centinelas aparecen en archivos, en protocolos de red y en entradas de usuario ("escribe FIN para terminar"). La alternativa sería saber de antemano cuántos datos hay, pero muchas veces no lo sabes: el centinela resuelve eso.

Otro patrón: while como contador con condición compuesta. Ya lo mencionamos como medida de seguridad, pero vamos a verlo como técnica principal: "repite mientras no lo logres Y no te pases del límite".

```
let secreta = 7;
let intento = 0;
let rondas = 0;

while (intento !== secreta && rondas < 10) {
  intento = intento + 1; // simulamos intentos crecientes: 1, 2, 3...
  rondas++;
  console.log("Ronda " + rondas + ": pruebo " + intento);
}

if (intento === secreta) {
  console.log("¡La encontré en " + rondas + " rondas!");
} else {
  console.log("Me rendí después de 10 rondas");
}
```

La condición tiene dos partes unidas con &&: la meta (intento !== secreta) y el freno de seguridad (rondas < 10). El ciclo termina por la primera que se vuelva false. En este ejemplo, intento llega a 7 en la ronda 7: intento !== secreta se vuelve false y salimos por éxito. Si la secreta fuera 99, saldríamos por el límite en la ronda 10. Este doble control es estándar en la industria: búsquedas con límite, reintentos de conexión con tope, lecturas con timeout. Acostúmbrate a escribirlo así desde ahora.

Hablemos de while anidados, que existen aunque son menos comunes que los for anidados. Funcionan igual: el interior completo por cada vuelta del exterior. El riesgo es el mismo pero amplificado: si olvidas actualizar la variable del ciclo interior, el infinito es interior y el exterior ni siquiera avanza a su segunda vuelta. Con while anidados, la disciplina de "¿qué hace que esto termine?" se aplica a CADA nivel por separado. Úsalos cuando el problema lo pida (dos dimensiones con condición de salida), no por deporte.

Un error sutil del while: actualizar la variable de control en el lugar equivocado dentro del bloque, especialmente combinado con continue. Mira:

```
// CÓDIGO MALO: continue se salta el i++
let i = 0;
while (i < 5) {
  if (i === 2) {
    continue; // ¡se salta el i++ de abajo! i se queda en 2 para siempre
  }
  console.log(i);
  i++;
}
```

Cuando *i* vale 2, el `continue` salta directamente a revisar la condición, sin pasar por `i++`. La condición `2 < 5` sigue `true`, el bloque se ejecuta otra vez, *i* sigue 2, `continue` otra vez... infinito. La lección: en un `while`, la actualización debe ser imposible de saltar. O la pones al mero inicio de la vuelta, o evitas `continue`, o reestructuras con `for`. Este bug es famoso porque el código "se ve bien" y el programa simplemente se congela en el 2.

Comparación final `for` vs `while` con ejemplos del mismo problema resuelto de las dos formas, para que sientas la diferencia de intención. Problema: sumar los números del 1 al 100.

```
// Con for: la intención es "recorrer del 1 al 100"
let sumaFor = 0;
for (let j = 1; j <= 100; j++) {
  sumaFor = sumaFor + j;
}

// Con while: la intención es "acumular hasta pasar de 100"
let sumaWhile = 0;
let n = 1;
while (n <= 100) {
  sumaWhile = sumaWhile + n;
  n++;
}
```

Ambos dan 5050. ¿Cuál es mejor? El `for`, porque el problema es un recorrido conocido y el `for` lo dice de un vistazo: inicio, fin y paso están en el encabezado. El `while` funciona, pero reparte la lógica en tres lugares (inicialización antes, condición en el paréntesis, actualización al final del bloque) y te obliga a verificar que no falte ninguna. Regla práctica final: si puedes escribirlo con `for` de forma natural, hazlo; reserva el `while` para cuando la condición de salida mande.

Más profundidad: funciones que se usan entre sí y errores típicos

Vamos a ver cómo las funciones se combinan en programas reales, porque una función sola es un truco; un conjunto de funciones que colaboran es un programa. La idea clave es la composición: funciones pequeñas que se llaman entre sí para lograr algo grande.

```
function esPar(n) {
  return n % 2 === 0;
}

function filtrarPares(hasta) {
  let pares = [];
  for (let i = 1; i <= hasta; i++) {
    if (esPar(i)) {
      pares.push(i);
    }
  }
  return pares;
}

function sumarLista(lista) {
  let total = 0;
  let i = 0;
  while (i < lista.length) {
    total = total + lista[i];
    i++;
  }
  return total;
}

// El programa principal compone las piezas:
let pares = filtrarPares(10);      // [2, 4, 6, 8, 10]
let suma = sumarLista(pares);      // 30
console.log("Suma de pares hasta 10: " + suma);
```

Fíjate en la arquitectura: `esPar` es una pieza mínima y pura; `filtrarPares` la usa dentro de un `for`; `sumarLista` usa un `while` por dentro (para que veas `while` y funciones juntos); y el "programa principal" no tiene lógica complicada: solo combina piezas con nombres claros. Si hay un bug en la detección de pares, lo arreglas en `esPar` y todo se corrige. Si quieres sumar impares, escribes `esImpar` y reutilizas el resto. Así se construye software que crece sin volverse un monstruo.

Hablemos de un error que TODOS cometen al inicio: olvidar los paréntesis al invocar. Mira la diferencia:

```
function obtenerNombre() {  
    return "Ana";  
}  
  
console.log(obtenerNombre); // imprime el CÓDIGO de la función (raro)  
console.log(obtenerNombre()); // imprime "Ana" (correcto)
```

Sin paréntesis, no estás llamando a la función: estás refiriéndote a la función misma como valor (las funciones son valores en JavaScript, un concepto poderoso que aprovecharás después). Con paréntesis, la ejecutas. Si tu programa imprime algo como "function obtenerNombre() {return...}" en lugar del resultado, ya sabes: te faltaron los paréntesis. Es el equivalente en funciones del = vs ===: un par de caracteres que lo cambian todo.

Otro error típico: usar el valor de retorno sin capturarlo. Si calcularIVA(100) devuelve 116 pero escribes solo calcularIVA(100); sin guardar ni imprimir el resultado, el cálculo se hace y se tira a la basura. JavaScript no se queja; simplemente el valor se pierde. Cuando una función devuelve algo, HAZ algo con ese algo: guárdalo, imprímelo, úsalo. Si no necesitas el valor, probablemente la función debería ser del tipo "hace algo" en lugar de "calcula algo".

Otro: parámetros en el orden equivocado. Si defines function restar(a, b) {return a - b;} y la llamas como restar(3, 10), obtienes -7, no 7. JavaScript asigna por posición, no por nombre: el primer argumento va al primer parámetro. Con funciones de dos o tres parámetros numéricos es facilísimo invertirlos sin darte cuenta. La defensa: nombres claros en la llamada mediante variables (restar(minuendo, sustraendo) se lee mejor que restar(3, 10)) y, cuando el orden sea confuso, reconsiderar el diseño.

¿Sabías que...? Las declaraciones de función tienen una característica llamada hoisting (elevación): puedes invocar a una función ANTES de la línea donde la defines, y funciona. JavaScript "sube" las definiciones al inicio antes de ejecutar.

```
saludar(); // funciona aunque la definición está abajo  
function saludar() {  
    console.log("¡Hola!");  
}
```

Esto es útil para organizar tu código con el programa principal arriba y las funciones de apoyo abajo, que a muchos les parece más legible. Pero ojo: el hoisting aplica a declaraciones

function, no a todas las formas de crear funciones (las funciones flecha asignadas a variables, que verás después, no se elevan igual). No abuses del hoisting como truco; úsalo como organización consciente.

Pregunta frecuente: "¿una función puede llamar a otra función?" Sí, y es lo normal: es la composición que vimos arriba. Pregunta frecuente: "¿una función se puede llamar a sí misma?" Sí, se llama recursión, y es un tema fascinante pero avanzado; lo verás más adelante en tu camino. Por ahora, quédate con que las funciones colaboran entre sí como un equipo.

Pregunta frecuente: "¿qué pasa si dos funciones necesitan compartir un dato?" Opciones en orden de preferencia: uno, pasarlo como parámetro y devolver el resultado (lo más limpio); dos, que una función llame a la otra pasándole el dato; tres, usar una variable global compartida (evítalo salvo configuración). Si te encuentras pasando el mismo dato por cinco funciones en cadena, es señal de que esas funciones quizá deberían ser una sola, o de que el dato debería viajar en un objeto (próximas unidades).

Cuidado con... las funciones larguísimas. Si tu función tiene más de 30 o 40 líneas, casi seguro está haciendo varias cosas: pártela. Una señal de alerta es cuando describes lo que hace usando la palabra "y" muchas veces: "valida el correo Y calcula el descuento Y guarda en base de datos" son tres funciones. Partir no es complicar: es ordenar.

Cuidado con... el perfeccionismo prematuro. Al inicio, tus funciones serán torpes: nombres regulares, tamaños irregulares, alguna global de más. Está bien. La habilidad de diseñar buenas funciones se entrena escribiendo muchas funciones malas primero. Lo importante es escribirlas, usarlas y notar qué duele: si te duele reutilizarla, el diseño necesita trabajo. El dolor es el maestro.

Para cerrar esta sección de explicación, volvamos a la idea central con un ejemplo que resume la unidad entera: un programa que adivina con while y funciones partiendo el problema.

```

function comparar(secreto, intento) {
  if (intento === secreto) return "igual";
  if (intento < secreto) return "bajo";
  return "alto";
}

function jugar(secreto, jugadas) {
  let i = 0;
  let resultado = "";
  while (resultado !== "igual" && i < jugadas.length) {
    resultado = comparar(secreto, jugadas[i]);
    console.log("Jugada " + jugadas[i] + ": " + resultado);
    i++;
  }
  return resultado === "igual";
}

let gane = jugar(13, [5, 18, 10, 13]);
console.log(gane ? "¡Ganaste!" : "Perdiste");

```

Todo lo de la unidad en doce líneas: comparar es una función pura con parámetros y returns tempranos; jugar usa un while con doble condición (meta + límite de seguridad) y compone a comparar; el programa principal solo orquesta e imprime. Si entiendes este ejemplo línea por línea, la unidad está dominada. Si algo no te cuadra, relee la sección correspondiente: está todo ahí.

Ejemplos paso a paso

Vamos a programar juntos, ejemplo por ejemplo, narrando el razonamiento como si estuviera sentado a tu lado. Cada ejemplo construye sobre lo que ya sabes y te deja un patrón reutilizable.

Ejemplo 1: Cuenta regresiva con while (anatomía del ciclo)

Vamos a hacer una cuenta regresiva del 5 al 1 con while, diseccionando cada parte para que veas dónde vive la inicialización, la condición y la actualización (que en el for estaban juntas y aquí están separadas).

```
// 1. INICIALIZACIÓN: antes del ciclo, creamos la variable de control
let cuenta = 5;

// 2. CONDICIÓN: se revisa ANTES de cada vuelta
while (cuenta > 0) {
  console.log(cuenta); // 3. BLOQUE: el trabajo de cada vuelta

  // 4. ACTUALIZACIÓN: dentro del bloque, acercamos el fin
  cuenta--;
}

// Cuando cuenta llega a 0, la condición es false y salimos
console.log("¡Tiempo!");
```

Sigue el flujo vuelta por vuelta. cuenta = 5. ¿5 > 0? Sí: imprime 5, cuenta baja a 4. ¿4 > 0? Sí: imprime 4, baja a 3. ¿3 > 0? Sí: 3, baja a 2. ¿2 > 0? Sí: 2, baja a 1. ¿1 > 0? Sí: 1, baja a 0. ¿0 > 0? No: salimos. Se imprime "¡Tiempo!". Seis revisiones de condición, cinco vueltas de trabajo.

Ahora, el experimento educativo: quita la línea cuenta-- y piensa qué pasaría (no lo corras con un número grande de vueltas... bueno, córrelo, pero prepárate para cerrar la pestaña). cuenta siempre vale 5, 5 > 0 siempre true, infinito imprimiendo cincos. Ese es el ciclo infinito en su forma más pura: la condición nunca se acerca a su fin porque nada dentro del bloque la mueve. Cada vez que escribas un while, señala con el dedo la línea que actualiza la variable de control y pregúntate: "¿esta línea se ejecuta SIEMPRE en cada vuelta?" Si hay un continue o un if que pueda saltársela, tienes un infinito en potencia.

Variante útil: cuenta regresiva que avisa en cada número par.

```
let cuenta = 10;
while (cuenta > 0) {
  if (cuenta % 2 === 0) {
    console.log(cuenta + " (par)");
  } else {
    console.log(cuenta);
  }
  cuenta--;
}
```

Aquí el bloque combina condición y ciclo: en cada vuelta decides qué imprimir según el valor actual. El while maneja la repetición, el if maneja la decisión. Así se ven los programas reales: capas de herramientas trabajando juntas.

Ejemplo 2: Adivinar con intentos limitados (while con doble condición)

Vamos a construir el juego de adivinar un número entre 1 y 10, con máximo 3 intentos. La condición del while tiene dos partes: no haber adivinado Y que queden intentos.

```
let secreto = 7;
let jugadas = [3, 9, 7]; // simulamos lo que "escribe" el jugador
let indice = 0;
let adivinado = false;
let intentos = 0;

// Seguimos mientras NO haya adivinado Y queden intentos (máximo 3)
while (!adivinado && intentos < 3) {
  let jugada = jugadas[indice];
  intentos++;
  console.log("Intento " + intentos + ": " + jugada);

  if (jugada === secreto) {
    adivinado = true; // ¡lo logró! esto terminará el ciclo
    console.log("¡Correcto!");
  } else if (jugada < secreto) {
    console.log("Muy bajo, intenta más alto");
  } else {
    console.log("Muy alto, intenta más bajo");
  }
  indice++;
}

if (!adivinado) {
  console.log("Se acabaron los intentos. Era el " + secreto);
}
```

Flujo detallado: intentos=0, adivinado=false. ¿!false && 0 < 3? -> ¿true && true? Sí. jugada = 3, intentos=1. 3 === 7? No. 3 < 7? Sí: "Muy bajo". Segunda vuelta: ¿!false && 1 < 3? Sí. jugada = 9, intentos=2. 9 < 7? No, else: "Muy alto". Tercera vuelta: jugada = 7, intentos=3. 7 === 7: adivinado = true, "¡Correcto!". Cuarta revisión: ¿!true &&...? -> false &&... = false: salimos. Como adivinado es true, no se imprime el mensaje de derrota.

Prueba el caso de derrota cambiando jugadas a [1, 2, 3]: tres vueltas, ninguna adivina, en la cuarta revisión intentos < 3 es false, salimos, y se imprime "Se acabaron los intentos. Era el 7". Y prueba el caso donde adivina al primer intento: [7, 1, 2]: una sola vuelta y sale. La doble condición maneja todos los casos elegantemente. Este es el patrón profesional para "intentar hasta lograrlo con límite".

Ejemplo 3: Menú con do...while (al menos una vez)

Vamos a simular un menú que se muestra al menos una vez y se repite hasta que el usuario elige salir (opción 3).

```
let opcion;
let elecciones = [1, 2, 3]; // simulamos: el usuario elige 1, luego 2, luego 3
let turno = 0;

do {
  // Mostramos el menú (en un programa real, esto sería un prompt)
  console.log("--- MENÚ ---");
  console.log("1. Saludar");
  console.log("2. Despedir");
  console.log("3. Salir");

  opcion = elecciones[turno];
  turno++;
  console.log("Elegiste: " + opcion);

  if (opcion === 1) {
    console.log("¡Hola!");
  } else if (opcion === 2) {
    console.log("¡Adiós!");
  }
  // si es 3, no hacemos nada especial: el while de abajo terminará
} while (opcion !== 3);

console.log("Programa terminado");
```

¿Por qué do...while y no while? Porque el menú DEBE mostrarse al menos una vez para que el usuario pueda elegir; con while tendríamos que inicializar opcion con un valor falso antes del ciclo (opcion = 0) solo para entrar la primera vez, lo cual es un truco feo. El do...while dice exactamente lo que queremos: "muestra el menú, y repite mientras no elija salir". Flujo: primera vuelta sin preguntar nada, opcion = 1, saluda. ¿1!== 3? Sí, repetimos. opcion = 2,

despide. ¿2!== 3? Sí. opcion = 3. ¿3!== 3? No: salimos. "Programa terminado". Los menús interactivos son EL caso de uso del do...while; cuando veas uno en código real, casi seguro usa esta estructura.

Ejemplo 4: Tu primera función útil (declarar, invocar, reutilizar)

Vamos a crear una función que calcula el precio con IVA (16%) y la vamos a usar varias veces. Paso a paso, como debes hacerlo siempre: definir, probar sola, reutilizar.

```
// PASO 1: DEFINIR. Nombre con verbo, parámetros claros.
function precioConIVA(precio) {
  let iva = precio * 0.16;
  let total = precio + iva;
  return total;
}

// PASO 2: PROBAR SOLA con un valor conocido.
// 100 + 16% = 116. Si da 116, la función está bien.
console.log(precioConIVA(100)); // 116

// PASO 3: REUTILIZAR en el programa real, varias veces, sin repetir lógica.
let precioPlayera = precioConIVA(250); // 290
let precioTenis = precioConIVA(1200); // 1392
let precioGorra = precioConIVA(180); // 208.8

console.log("Playera: " + precioPlayera);
console.log("Tenis: " + precioTenis);
console.log("Gorra: " + precioGorra);
console.log("Total compra: " + (precioPlayera + precioTenis + precioGorra));
```

Observa el flujo de una llamada, por ejemplo precioConIVA(250): JavaScript asigna 250 al parámetro precio (como let precio = 250), ejecuta el bloque: iva = 40, total = 290, return 290. Ese 290 "regresa" al lugar de la llamada y se guarda en precioPlayera. Luego la función se olvida de todo: sus variables locales (iva, total) desaparecen. En la siguiente llamada con 1200, todo empieza de cero con el nuevo valor. Las funciones no tienen memoria entre llamadas (salvo trucos avanzados): cada invocación es un mundo nuevo. Eso las hace predecibles y fáciles de probar.

Y ahora imagina que el IVA cambia a 18%: editas UNA línea (el 0.16) y los tres precios se corrigen. Con código copiado serían tres ediciones y el riesgo de olvidar una. Esta es la historia de la anécdota de la introducción, convertida en buena práctica.

Ejemplo 5: Parámetros, argumentos y valores por defecto (función de envío)

Vamos a hacer una función que calcula el costo de envío de una tienda: tiene un costo base por kilómetro, una distancia, y un descuento opcional para clientes frecuentes (por defecto, sin descuento).

```
// costoPorKm y distancia son obligatorios; descuento tiene valor por defecto 0
function costoEnvio(costoPorKm, distancia, descuento = 0) {
  let subtotal = costoPorKm * distancia;
  let total = subtotal - descuento;
  return total;
}

// Cliente normal: sin descuento (se usa el valor por defecto)
console.log(costoEnvio(10, 15));          // 150

// Cliente frecuente: con descuento de 20
console.log(costoEnvio(10, 15, 20));      // 130

// Otro caso: distinta distancia y tarifa
console.log(costoEnvio(8, 30, 50));       // 190
```

En la primera llamada, solo pasamos dos argumentos: descuento no recibe nada, así que vale `undefined`, y el valor por defecto 0 entra en acción: $\text{total} = 150 - 0 = 150$. En la segunda, pasamos 20 y se usa ese valor. Fíjate en el orden: los parámetros con defecto van al final, porque los argumentos se asignan por posición y no querrías tener que pasar un valor "vacío" para llegar al tercero.

Ahora un ejemplo de lo que pasa si te equivocas en el orden o la cantidad:

```
console.log(costoEnvio(15)); // subtotal = 15 * undefined = NaN, total = NaN
```

distancia vale `undefined`, $15 * \text{undefined}$ es `NaN`, y `NaN` se propaga: el resultado es `NaN`. JavaScript no te avisa; te devuelve un "no es número" silencioso. Por eso, cuando una función te devuelva `NaN`, lo primero que debes revisar es si le pasaste todos los argumentos en el orden correcto. Es el equivalente a revisar que enchufaste todo antes de decir que el aparato no sirve.

Ejemplo 6: return temprano (validador de edad elegante)

Vamos a validar una edad para un registro: debe ser un número, entre 0 y 120. Usaremos returns tempranos para manejar cada caso inválido al inicio, dejando el camino feliz al final, sin anidación profunda.

```
function validarEdad(edad) {
  // Caso 1: ni siquiera es un número
  if (typeof edad !== "number") {
    return "Error: la edad debe ser un número";
  }
  // Caso 2: número imposible (negativo o absurdo)
  if (edad < 0 || edad > 120) {
    return "Error: edad fuera de rango (0-120)";
  }
  // Caso 3: menor de edad (ejemplo: registro solo adultos)
  if (edad < 18) {
    return "Rechazado: debes tener 18 o más";
  }
  // Camino feliz: todo bien
  return "Aceptado: registro exitoso";
}

console.log(validarEdad("veinte")); // Error: debe ser un número
console.log(validarEdad(-5));       // Error: fuera de rango
console.log(validarEdad(16));       // Rechazado: debes tener 18 o más
console.log(validarEdad(25));       // Aceptado
```

Sigue el flujo con edad = 16: ¿typeof 16 !== "number"? No (sí es número), seguimos. ¿16 < 0 || 16 > 120? No, seguimos. ¿16 < 18? Sí: return "Rechazado..." y la función TERMINA ahí; la última línea nunca se ejecuta para este caso. Con edad = 25: pasa las tres revisiones sin entrar a ningún if y llega al return final: "Aceptado". Este estilo de "guardias al inicio" hace que cada validación sea independiente y legible; compáralo con la versión anidada (if dentro de if) y verás por qué los profesionales prefieren los returns tempranos. Cada guardia es una frase simple: "si esto está mal, avisa y sal".

Nota el uso de typeof: es un operador que te dice el tipo de un valor ("number", "string", "boolean"...). Lo usamos aquí para validar el tipo antes de comparar, una buena práctica defensiva cuando los datos vienen de fuera (formularios, usuarios).

Ejemplo 7: Alcance en acción (el experimento definitivo)

Este ejemplo está diseñado para que lo corras y veas el scope con tus propios ojos. Tiene globales, locales, parámetros y un sombreado, todo marcado.

```
let tasa = 0.16;           // GLOBAL: existe en todo el programa
let tienda = "MiTienda";  // GLOBAL

function calcularTotal(precio, cantidad) { // precio y cantidad: PARÁMETROS
  (locales)
  let subtotal = precio * cantidad; // LOCAL: solo vive aquí dentro
  let total = subtotal * (1 + tasa); // usa la global tasa: permitido
  return total;
}

function mostrarTienda() {
  let tienda = "OtraTienda"; // LOCAL que sombrea a la global dentro de esta
  función
  console.log("Dentro: " + tienda); // "OtraTienda" (la local tapa a la global)
}

console.log(calcularTotal(100, 2)); // 232
mostrarTienda();                  // Dentro: OtraTienda
console.log("Fuera: " + tienda);  // Fuera: MiTienda (la global no cambió)

// console.log(subtotal); // Si descomentas esto: ERROR. subtotal no existe aquí.
```

Análisis línea por línea del scope: tasa y tienda globales son visibles desde cualquier función (calcularTotal lee tasa sin problema). precio, cantidad y subtotal son locales de calcularTotal: nacen con la llamada y mueren al terminar; fuera de la función no existen. En mostrarTienda, la local tienda sombrea a la global: dentro se ve "OtraTienda", pero la global sigue valiendo "MiTienda" afuera, porque la local es otra variable en otro cuarto. La última línea comentada daría error si la activas: subtotal no existe en el alcance global. Este ejemplo es tu laboratorio: descomenta la última línea, corre, lee el error (algo como "subtotal is not defined"), y habrás visto el scope en acción real, no solo en teoría.

Ejemplo 8: Descomposición completa (sistema de calificaciones)

Vamos a cerrar los ejemplos con un programa completo que usa todo: funciones que colaboran, while, for, comparaciones y lógica. Un sistema que procesa las calificaciones de un grupo.

```

// Pieza 1: calcula el promedio de una lista
function promedio(lista) {
  let suma = 0;
  for (let i = 0; i < lista.length; i++) {
    suma = suma + lista[i];
  }
  return suma / lista.length;
}

// Pieza 2: cuenta cuántos aprobaron (calificación >= 6)
function contarAprobados(lista) {
  let cuenta = 0;
  let i = 0;
  while (i < lista.length) {
    if (lista[i] >= 6) {
      cuenta++;
    }
    i++;
  }
  return cuenta;
}

// Pieza 3: encuentra la calificación más alta
function maxima(lista) {
  let max = lista[0];
  for (let i = 1; i < lista.length; i++) {
    if (lista[i] > max) {
      max = lista[i];
    }
  }
  return max;
}

// Pieza 4: orquesta las piezas y muestra el reporte
function reporte(lista) {
  console.log("=== REPORTE DEL GRUPO ===");
  console.log("Alumnos: " + lista.length);
  console.log("Promedio: " + promedio(lista));
  console.log("Aprobados: " + contarAprobados(lista) + " de " + lista.length);
  console.log("Calificación máxima: " + maxima(lista));
}

// Programa principal: solo datos y una llamada
let calificaciones = [8, 5, 9, 7, 4, 10, 6];

```

```
reporte(calificaciones);
```

Observa la arquitectura: tres funciones "trabajadoras" (promedio, contarAprobados, maxima), cada una con UNA tarea y un return; una función "orquestadora" (reporte) que las combina; y el programa principal de dos líneas. A propósito usé for en dos piezas y while en una, para que veas que la elección depende del contexto y ambas conviven. Si el criterio de aprobación cambia de 6 a 7, editas UNA línea en contarAprobados. Si quieres agregar "reprobados", escribes una función nueva reutilizando el patrón. Este programa es pequeño, pero su estructura es la de los programas grandes: piezas pequeñas, nombres claros, composición. Cuando escribas tus propios proyectos, imita esta forma.

Práctica

La misma regla de siempre: intenta primero sin ver la solución. Con funciones es aún más importante, porque el músculo que estás entrenando es el de partir problemas, y ese músculo solo crece cuando TÚ decides cómo partir.

Ejercicio 1: Suma con while (traducción for -> while)

Enunciado: Escribe un programa que sume los números del 1 al 50 usando un ciclo while (no for). Imprime la suma final. La suma del 1 al 50 es 1275; úsalo para verificar.

Solución:

```
let suma = 0;
let n = 1; // inicialización ANTES del ciclo

while (n <= 50) { // condición: mientras no pasemos de 50
  suma = suma + n;
  n++; // actualización DENTRO del bloque: ¡no la olvides!
}

console.log("Suma del 1 al 50: " + suma); // 1275
```

Explicación paso a paso: Como es while, las tres partes del for se reparten: let n = 1 antes del ciclo (inicialización), n <= 50 en el paréntesis (condición) y n++ al final del bloque (actualización). En cada vuelta sumamos n y lo incrementamos. Cuando n llega a 51, la condición es false y salimos con suma = 1275. Verifica el resultado: si te da otro número, algo anda mal con tus límites (¿empezaste en 0? ¿usaste <?). Y haz el experimento mental del infinito: si borras n++, n siempre vale 1, la condición siempre true, ciclo eterno. Señala con el dedo tu línea de actualización en cada while que escribas; ese hábito vale oro.

Ejercicio 2: Contraseña con do...while (al menos un intento)

Enunciado: Simula un login donde la contraseña correcta es "abc123". Tienes una lista de intentos: ["hola", "abc123"]. Usa do...while para procesar los intentos: muestra cada intento, y repite mientras la contraseña sea incorrecta. Al final indica si entró o no. Luego prueba con intentos ["hola", "mundo"] para ver el caso de fracaso.

Solución:

```
let correcta = "abc123";
let intentos = ["hola", "abc123"];
let i = 0;
let acceso = false;
let intentoActual;

do {
  intentoActual = intentos[i];
  console.log("Probando: " + intentoActual);
  if (intentoActual === correcta) {
    acceso = true;
  }
  i++;
} while (!acceso && i < intentos.length);

if (acceso) {
  console.log("¡Bienvenido!");
} else {
  console.log("Acceso denegado");
}
```

Explicación paso a paso: Usamos do...while porque queremos procesar al menos un intento aunque la lista viniera vacía (con while y lista vacía, ni siquiera revisaríamos). En cada vuelta tomamos el intento actual, lo comparamos con === (estricto, como aprendiste en la unidad 3) y

si coincide marcamos acceso = true. La condición de salida es doble: !acceso (seguimos si no ha entrado) && i < intentos.length (pero sin pasarnos de la lista: el freno de seguridad). Con ["hola", "abc123"]: primera vuelta "hola" != "abc123", acceso sigue false, i=1; condición: !false && 1 < 2 -> true, seguimos. Segunda vuelta: "abc123" === "abc123", acceso = true, i=2; condición: !true -> false, salimos. "¡Bienvenido!". Con ["hola", "mundo"]: dos vueltas sin éxito, i llega a 2, i < 2 es false, salimos, "Acceso denegado". Fíjate que el freno i < intentos.length evita leer más allá de la lista (donde habría undefined y un posible infinito). Doble condición: meta + seguridad, siempre.

Ejercicio 3: Función esMayorDeEdad (parámetros y return)

Enunciado: Escribe una función llamada esMayorDeEdad que reciba una edad como parámetro y devuelva true si es mayor o igual a 18, false si no. Luego úsala para revisar las edades 15, 18 y 21, imprimiendo un mensaje personalizado en cada caso.

Solución:

```
function esMayorDeEdad(edad) {
  return edad >= 18;
}

// La probamos sola primero con los tres casos clave
console.log(esMayorDeEdad(15)); // false
console.log(esMayorDeEdad(18)); // true (el límite incluido)
console.log(esMayorDeEdad(21)); // true

// Ahora la usamos en mensajes personalizados
let edades = [15, 18, 21];
for (let i = 0; i < edades.length; i++) {
  if (esMayorDeEdad(edades[i])) {
    console.log(edades[i] + " años: puede entrar");
  } else {
    console.log(edades[i] + " años: no puede entrar");
  }
}
```

Explicación paso a paso: La función es mínima y pura: recibe edad, devuelve el resultado de la comparación edad >= 18 directamente con return (no necesitamos if: la comparación YA es un booleano, devolverla directo es más elegante que if (edad >= 18) return true; else return false;). Probamos los tres casos frontera: 15 (debajo), 18 (exacto, debe dar true por el >=) y 21 (arriba). Luego la reutilizamos dentro de un for para tres personas distintas: la lógica de "qué

es mayor de edad" vive en un solo lugar y el ciclo solo la aplica. Si la edad legal cambiara a 21, editarías una línea. Este ejercicio parece simple, pero contiene el patrón completo: definir, probar sola, reutilizar.

Ejercicio 4: Función con valor por defecto (descuento en tienda)

Enunciado: Escribe una función `aplicarDescuento(precio, porcentaje = 10)` que devuelva el precio con el descuento aplicado. Úsala tres veces: una con el descuento por defecto, una con 25% y una con 0%. Verifica los resultados a mano.

Solución:

```
function aplicarDescuento(precio, porcentaje = 10) {  
  let descuento = precio * (porcentaje / 100);  
  return precio - descuento;  
}  
  
console.log(aplicarDescuento(200));           // 180 (descuento por defecto: 10%)  
console.log(aplicarDescuento(200, 25));       // 150 (25% de 200 = 50)  
console.log(aplicarDescuento(200, 0));        // 200 (0%: sin descuento)
```

Explicación paso a paso: El parámetro `porcentaje` tiene valor por defecto 10: si no le pasas el segundo argumento, JavaScript usa 10. Primera llamada: `aplicarDescuento(200)` -> `porcentaje = 10`, `descuento = 20`, `return 180`. Segunda: `aplicarDescuento(200, 25)` -> el argumento 25 reemplaza al defecto, `descuento = 50`, `return 150`. Tercera: `aplicarDescuento(200, 0)` -> ojo, pasamos 0 EXPLÍCITAMENTE, y 0 reemplaza al valor por defecto (el defecto solo se usa cuando el argumento es `undefined`, no cuando es 0): `descuento = 0`, `return 200`. Este matiz es importante: "no pasar nada" no es lo mismo que "pasar cero". Verifica a mano: 10% de 200 es 20, $200 - 20 = 180$. Correcto. Esta función es del tipo "calcula y devuelve": pura, predecible, reutilizable en cualquier tienda.

Ejercicio 5: Alcance (predice la salida)

Enunciado: Sin correr el código, predice qué imprime cada `console.log`. Luego córrelo y compara. Si fallaste alguna, relea la sección de alcance.

```

let x = 10;

function prueba() {
  let x = 20;
  console.log("A: " + x);
  x = x + 5;
  console.log("B: " + x);
}

prueba();
console.log("C: " + x);

function prueba2() {
  x = x + 100; // ¡SIN let!
}
prueba2();
console.log("D: " + x);

```

Solución (salida):

```

A: 20
B: 25
C: 10
D: 110

```

Explicación paso a paso: A: dentro de prueba, la local $x = 20$ sombrea a la global; se imprime la local: 20. B: $x = 20 + 5 = 25$, sigue siendo la local; se imprime 25. Al terminar prueba, su x local desaparece. C: fuera de la función solo existe la global, que nadie tocó: 10. Hasta aquí, todo protegido por el scope. D: en prueba2, $x = x + 100$ SIN let: JavaScript no crea una local, sino que modifica la GLOBAL (x vale 10, $+100 = 110$). La global queda contaminada. Esta es la fuga de variable en acción: una línea sin let cambió el estado de todo el programa. Si predijiste D: 10, caíste en la trampa clásica; ahora ya sabes por qué la regla es "declara SIEMPRE con let o const". Este ejercicio es de los que más enseñan con menos código: cuatro letras (A, B, C, D) y tres conceptos (sombreado, locales que mueren, fuga a global).

Ejercicio 6: Reto integrador (while + funciones + descomposición)

Enunciado: Un programa debe encontrar el primer número mayor que 100 que sea divisible entre 7 y 13 a la vez (o sea, entre 91). Parte el problema en funciones: una función `esDivisible(n, divisor)` que diga si n es divisible entre divisor, y un programa principal que use

un while para buscar desde 101 hacia arriba. El programa debe imprimir el número encontrado y cuántas vueltas tardó. (Pista: 182 es divisible entre 91... verifícalo con tu programa.)

Solución:

```
function esDivisible(n, divisor) {  
  return n % divisor === 0;  
}  
  
let candidato = 101;  
let vueltas = 0;  
  
// Seguimos mientras NO sea divisible entre 7 y 13 a la vez  
while (!(esDivisible(candidato, 7) && esDivisible(candidato, 13))) {  
  candidato++;  
  vueltas++;  
}  
  
console.log("Encontrado: " + candidato); // 182  
console.log("Vueltas: " + vueltas);      // 81
```

Explicación paso a paso: Primero partimos: esDivisible es una función pura y mínima ($n \% \text{divisor} === 0$), reutilizable para cualquier divisor. El programa principal no sabe de residuos: solo orquesta. El while tiene como condición la NEGACIÓN de la meta: "mientras no sea divisible entre 7 y 13, sigue buscando". En cada vuelta probamos al candidato: si no cumple, lo incrementamos y contamos la vuelta. Vamos a verificar: 101 entre 7... 98 es 7×14 , 101 no es divisible. El ciclo avanza: 105 sí es divisible entre 7 (15×7) pero no entre 13. Seguimos... 182 = $2 \times 91 = 2 \times 7 \times 13$: divisible entre ambos. La condición $!(\text{true} \ \&\& \ \text{true}) = !\text{true} = \text{false}$: salimos. candidato = 182, vueltas = $182 - 101 = 81$. Fíjate en la condición negada con paréntesis: $!(A \ \&\& \ B)$ se lee "mientras no se cumpla (A y B)". Sin los paréntesis sería $!A \ \&\& \ B$, que significa otra cosa (precedencia del !, ¿te acuerdas de la unidad 3?). Todo conectado: funciones para partir, while para buscar, lógica para la condición. Si resolviste este sin ayuda, tu nivel ya es otro.

Resumen de la unidad

Cerramos con los puntos clave. Si estos te quedan claros, la unidad cumplió.

Primero: el while repite mientras su condición sea true, revisándola ANTES de cada vuelta. Es el ciclo de la meta desconocida: úsalo cuando no sabes cuántas vueltas necesitas ("repetir hasta lograrlo"), no cuando recorres algo conocido (ahí manda el for).

Segundo: la ley fundamental del while: el ciclo solo termina si algo dentro de él acerca la condición a su fin. Señala con el dedo la línea de actualización en cada while que escribas. Si puede saltarse (por un continue o un if), tienes un ciclo infinito en potencia.

Tercero: el ciclo infinito no rompe nada: congela la pestaña o el programa y se arregla cerrando o con Ctrl+C. Sus causas típicas son olvidar actualizar la variable, actualizarla al revés, usar = en lugar de === en la condición, o depender de algo externo que nunca cambia. Para prevenir: pregúntate qué lo termina antes de escribir el bloque, agrega frenos de seguridad con && cuando dependas de datos externos, e imprime la variable de control en tus pruebas.

Cuarto: do...while ejecuta el bloque AL MENOS UNA VEZ porque revisa la condición al final. Úsalo para menús y validaciones de entrada, donde primero necesitas un valor para poder revisarlo. No olvides el punto y coma final.

Quinto: una función es un bloque de código con nombre que defines una vez e invocas cuantas veces quieras. Definir no es ejecutar: sin los paréntesis de la llamada, nada pasa. Su razón de ser es la reutilización y el principio DRY: no te repitas; la lógica vive en un solo lugar y se corrige en un solo lugar.

Sexto: los parámetros son los nombres en la definición (los espacios que la función espera) y los argumentos son los valores reales en la llamada (lo que llena esos espacios). Se asignan por posición. Puedes dar valores por defecto a los parámetros (van al final) para hacerlos opcionales.

Séptimo: return devuelve un valor a quien llamó y termina la función inmediatamente. Las funciones que calculan devuelven valores; las que hacen cosas producen efectos. No confundas unas con otras, y escribe return en la misma línea que su valor.

Octavo: el alcance (scope) dice dónde existe cada variable. Las locales viven dentro de su función y mueren al terminar; las globales viven en todo el programa. Prefiere locales, declara cerca de donde usas, y declara SIEMPRE con let o const para evitar fugas silenciosas a global. Una local puede sombrear a una global del mismo nombre dentro de su función.

Noveno: parte los problemas grandes en funciones pequeñas con una sola tarea y nombres con verbo. Las funciones se llaman entre sí (composición): unas calculan, otras orquestan. Si describes una función usando muchos "y", probablemente son varias funciones.

Décimo: todo se conecta. El while usa comparaciones y lógica en su condición; las funciones contienen ciclos, condiciones y comparaciones por dentro; y los programas reales son funciones que colaboran usando ciclos para repetir. Con variables, operadores, decisiones, for,

while y funciones, ya tienes la caja de herramientas completa de la programación estructurada. Lo que sigue (arrays, objetos, DOM) es nuevo material para estas herramientas. Vas muy bien: sigue escribiendo código todos los días.

UNIDAD 5: Funciones modernas y listas

Bienvenido a la unidad donde JavaScript deja de sentirse como "el lenguaje del tutorial" y empieza a sentirse como una herramienta de verdad en tus manos. Si has llegado hasta aquí, ya sabes qué es JavaScript, cómo meterlo en una página, cómo guardar valores en variables, cómo comparar cosas, cómo tomar decisiones con `if` y `else`, y cómo repetir tareas con `for` y `while`. Con eso ya puedes hacer programas funcionales. Pero hay una diferencia enorme entre un programa que "funciona" y un programa que está bien escrito, que se lee rápido, que otros programadores entienden sin esfuerzo y que tú mismo entiendes tres meses después cuando regresas al código. Esa diferencia vive exactamente en los temas de esta unidad: las funciones modernas y las listas de datos.

Piensa en lo siguiente. Cuando aprendes a cocinar, al principio sigues recetas al pie de la letra: "pica la cebolla, fríela diez minutos, agrega el jitomate". Con el tiempo aprendes atajos: en vez de decirle a alguien "agarra el cuchillo, ponlo sobre la cebolla, baja el cuchillo, levanta el cuchillo, repite", simplemente dices "pica la cebolla". Las funciones son eso: un nombre corto para una serie de pasos. Y las funciones flecha, que es el primer gran tema de esta unidad, son la versión todavía más corta y más elegante de decirlo. Son la taquigrafía del programador moderno.

En esta unidad vas a aprender las funciones flecha, esa sintaxis con `=>` que ves en todo el código JavaScript moderno y que al principio parece un jeroglífico. Vas a entender por qué existen, cuándo te conviene usarlas y, algo igual de importante, cuándo NO te conviene usarlas. Vas a dominar el `return` implícito, ese truco donde la función devuelve un valor sin que tengas que escribir la palabra `return`. Vas a ver cómo los parámetros trabajan con las funciones flecha. Y después vamos a entrar de lleno al mundo de los arrays, las listas de datos, que son la estructura más usada en todo JavaScript: cómo crearlos, cómo funcionan los índices que empiezan en cero (y por qué empiezan en cero, que tiene su historia), cómo acceder a los elementos, cómo modificarlos, y los métodos más importantes: `push`, `pop`, `shift`, `unshift` y la propiedad `length`. Vas a aprender a recorrer arrays elemento por elemento, vas a trabajar con arrays que mezclan tipos de datos, y vas a entender una de las trampas más clásicas del lenguaje: la diferencia entre copiar un array y simplemente crear otra referencia al mismo array.

¿Por qué importa todo esto? Porque en el mundo real, los programas casi nunca trabajan con un solo dato. Trabajan con listas: la lista de productos de una tienda, la lista de usuarios registrados, la lista de mensajes de un chat, la lista de calificaciones de un alumno, la lista de fotos de una galería. Si no dominas los arrays, te vas a atorar en cada proyecto. Y las

funciones flecha están en todas partes del JavaScript moderno: en los tutoriales, en el código de las empresas, en las librerías como React, en los ejemplos de Stack Overflow. Si no las entiendes, vas a sentir que lees otro idioma cada vez que abras un proyecto ajeno. Esta unidad te quita esa barrera.

El panorama de la unidad es el siguiente. Primero, en la explicación detallada, vamos a desmenuzar las funciones flecha desde cero: la sintaxis completa, las versiones abreviadas, las reglas para quitar paréntesis y llaves, el return implícito y sus límites, y la gran advertencia sobre this que todo principiante debe conocer. Luego pasamos a los arrays: qué son conceptualmente, cómo se crean de varias formas, el sistema de índices base cero con su lógica, cómo leer y escribir elementos, y cada método esencial explicado con ejemplos de la vida real. Veremos cómo recorrer un array con un ciclo for clásico, qué pasa con los arrays mixtos, y el tema de copia contra referencia, que es donde más programadores nuevos se tropiezan. Después, en los ejemplos paso a paso, vamos a construir código real comentado línea por línea: desde una función flecha sencilla hasta un mini programa que gestiona una lista de tareas. En la práctica tendrás ejercicios graduados, cada uno con su solución explicada paso a paso, para que no solo copies sino que entiendas. Y cerraremos con un resumen de los puntos clave.

Un consejo antes de empezar: no te brinques las analogías ni las secciones de errores comunes. Sé que dan ganas de ir directo al código, pero te lo digo por experiencia de haber visto a cientos de estudiantes atorarse: los que entienden el "por qué" avanzan el triple de rápido que los que solo memorizan sintaxis. Las funciones flecha y los arrays son temas que vas a usar todos los días de tu vida como programador. Vale la pena invertirle tiempo de calidad aquí. Vamos a darle.

Explicación detallada

Empecemos con las funciones flecha, porque son la puerta de entrada al JavaScript moderno. Para entenderlas bien, primero recordemos qué es una función tradicional. Una función es un bloque de código con nombre que puedes ejecutar cuantas veces quieras. La sintaxis clásica que ya conoces se ve así: escribes la palabra `function`, luego el nombre, luego paréntesis con los parámetros, luego llaves con el cuerpo. Eso funciona perfecto y nadie te va a regañar por usarlo. Pero con el tiempo, la comunidad de JavaScript se dio cuenta de algo: muchísimas funciones son cortitas. Son funciones de una o dos líneas que solo transforman un dato, que solo calculan algo rápido, que solo sirven como ayudantes dentro de otra operación más grande. Escribir `function`, el nombre, los paréntesis y las llaves para una función de una línea se siente como usar un camión de mudanza para llevar una mochila. Las funciones flecha nacieron para esos casos: son la versión compacta, la taquigrafía.

La sintaxis básica de una función flecha se ve así en su forma más completa: `const sumar = (a, b) => {return a + b;};`. Vamos a desarmarla pieza por pieza porque cada pieza tiene su razón de ser. Primero, `const sumar =`: casi siempre guardamos la función flecha en una constante, porque las funciones flecha son anónimas por naturaleza, no tienen nombre propio, así que les damos un nombre guardándolas en una variable. Usamos `const` y no `let` porque normalmente no vas a reasignar una función; una vez que `sumar` suma, quieres que siempre sume. Luego vienen los parámetros entre paréntesis: `(a, b)`, igual que en las funciones normales. Después viene la flecha `=>`, que es literalmente un signo igual seguido de un signo mayor que, y que se lee como "toma estos parámetros y produce esto". Y al final el cuerpo entre llaves, igual que una función normal.

Ahora viene lo interesante: las abreviaciones. JavaScript te permite ir recortando la sintaxis según el caso, y cada recorte tiene su regla. Primera regla: si la función tiene un solo parámetro, puedes quitar los paréntesis. En vez de `(nombre) => {return "Hola"+ nombre;}` puedes escribir `nombre => {return "Hola"+ nombre;}`. Ojo: esto solo funciona con exactamente un parámetro. Con cero parámetros necesitas los paréntesis vacíos: `() => {...}`. Con dos o más parámetros también necesitas los paréntesis: `(a, b) => {...}`. Es una regla sencilla pero la gente la olvida y luego se pregunta por qué el código truena. Segunda regla: si el cuerpo de la función es una sola expresión, puedes quitar las llaves y la palabra `return`, y el valor de esa expresión se devuelve automáticamente. Eso es el famoso `return` implícito. En vez de `(a, b) => {return a + b;}` escribes `(a, b) => a + b`. Se lee: "toma a y b, produce a + b". Es elegante, es corto, y cuando te acostumbras, lo lees rapidísimo.

Pero cuidado, porque el `return` implícito tiene sus matices y aquí es donde los principiantes se confunden. El `return` implícito solo funciona cuando quitas las llaves. En el momento en que pones llaves, ya no hay `return` implícito: tienes que escribir `return` explícitamente o la función

devuelve undefined. Esto causa un error clasicísimo: alguien escribe `const duplicar = (n) => {n * 2};` y se sorprende de que `duplicar(5)` devuelva undefined en vez de 10. ¿Por qué? Porque puso llaves, entonces JavaScript ya no devuelve nada automáticamente; la expresión `n * 2` se calcula y se tira a la basura. La regla de oro es: llaves puestas, return escrito; llaves quitadas, return automático. Grábatelo como mantra.

Hay otro detalle fino con el return implícito: ¿qué pasa si quieres devolver un objeto literal? Un objeto se escribe con llaves: `{nombre: "Ana"}`. Pero si escribes `const crear = () => {nombre: "Ana"};`, JavaScript no ve un objeto: ve las llaves como el cuerpo de la función, y dentro del cuerpo `nombre: "Ana"` se interpreta como una etiqueta, no como un objeto. El resultado es undefined y mucha confusión. La solución es envolver el objeto en paréntesis: `const crear = () => ({nombre: "Ana"});`. Los paréntesis le dicen a JavaScript "esto es una expresión, no el cuerpo de la función". Este es uno de esos trucos que parecen magia negra hasta que entiendes la lógica, y después lo usas sin pensar. Cada programador de JavaScript se ha topado con este problema al menos una vez; ahora tú ya lo sabes de antemano.

Hablemos de cuándo usar funciones flecha y cuándo no, porque no son un reemplazo universal de las funciones normales. Úsalas cuando la función sea corta y sencilla: transformaciones de datos, cálculos de una línea, callbacks que le pasas a otros métodos. Son perfectas para eso. El código se vuelve más legible porque lo importante, la transformación, queda al frente sin ceremonia. Por ejemplo, cuando más adelante uses `map` para transformar un array, `escribir.map(n => n * 2)` es mucho más claro que `escribir.map(function(n) {return n * 2;})`. La versión flecha deja ver de inmediato que cada número se multiplica por dos.

Cuándo NO usarlas: primero, cuando necesites la palabra `this` con su comportamiento tradicional de función. Este es el tema más delicado de las funciones flecha y merece su propia explicación. En una función normal, `this` cambia según cómo se llame la función: si la llamas como método de un objeto, `this` es ese objeto. En una función flecha, `this` NO cambia: la flecha no tiene su propio `this`, sino que hereda el `this` del lugar donde fue creada. Esto es una ventaja enorme en muchos casos, por ejemplo dentro de callbacks donde quieres conservar el `this` original, pero es un problema si defines un método de un objeto con sintaxis de flecha esperando que `this` sea el objeto, porque no lo será. En la unidad 6 vamos a profundizar en `this` con ejemplos concretos; por ahora quédate con la regla práctica: para métodos de objetos que usen `this`, usa funciones normales; para todo lo demás corto y funcional, las flechas son tus amigas.

Segundo caso donde no convienen: cuando la función es larga o compleja. Si tu función flecha tiene veinte líneas, varios ifs anidados y tres returns, la sintaxis compacta ya no ayuda: estorba. Una función larga merece una función con nombre tradicional, bien documentada,

donde la estructura sea clara. La brevedad es una virtud solo cuando el contenido es breve. Tercer caso: cuando necesites el objeto arguments, que es una variable especial disponible solo en funciones normales que contiene todos los argumentos recibidos. Las flechas no tienen arguments propio. En la práctica moderna esto importa poco porque puedes usar parámetros rest, pero es bueno saberlo para no llevarte sorpresas leyendo código viejo.

¿Sabías que las funciones flecha siempre son anónimas? Cuando escribes `const saludar = () => "Hola";`, la función en sí no tiene nombre; el nombre saludar es el de la constante que la guarda. JavaScript es lo bastante listo para inferir el nombre en los mensajes de error, así que si algo falla verás "saludar" en el rastreo, pero técnicamente la función es anónima. Esto importa cuando pasas funciones flecha directamente como argumentos sin guardarlas en variables: existen, hacen su trabajo y desaparecen, como un mensajero que entrega el paquete y se va.

Otra anécdota de error típico: los principiantes a veces escriben la flecha al revés, `=<` en vez de `=>`, o se les olvida el signo igual y escriben solo `>`. El error que lanza el navegador puede ser críptico, algo sobre "unexpected token", y se quedan mirando la pantalla sin entender. Si ves un error raro justo donde definiste una flecha, revisa primero que la flecha esté bien escrita: igual seguido de mayor que, sin espacios raros en medio aunque los espacios sí se permiten. También es común olvidar que una función flecha asignada a const no puede ser redeclarada; si intentas `const sumar = ...` dos veces en el mismo ámbito, truena. Usa nombres distintos o reestructura.

Pasemos ahora a los parámetros en las funciones flecha, porque aquí hay jugo. Los parámetros funcionan igual que en las funciones normales: son los nombres que la función espera recibir. Puedes tener valores por defecto: `const saludar = (nombre = "visitante") => "Hola" + nombre;` Si llamas `saludar()` sin argumentos, nombre vale "visitante". Esto es utilísimo para hacer funciones amables que no truenen cuando alguien olvida un argumento. También puedes usar parámetros rest para recibir un número variable de argumentos: `const sumarTodo = (...numeros) => numeros.length;`. Los tres puntitos... juntan todos los argumentos en un array. Y puedes desestructurar parámetros, aunque eso lo veremos más a fondo después; por ahora basta saber que la flecha acepta todo lo que acepta una función normal en cuanto a parámetros.

Un mini-reto mental: ¿qué devuelve `const f = x => x * 2;` cuando la llamas con `f(5)`? Devuelve 10, con return implícito y sin paréntesis porque hay un solo parámetro. ¿Y `const g = x => {x * 2};`? Devuelve undefined, porque las llaves matan el return implícito. Si entendiste la diferencia entre esas dos, ya dominas el ochenta por ciento de los errores de flechas que verás en la vida real.

Cambiamos de tercio y entremos al mundo de los arrays. Un array es una lista ordenada de valores. Eso es todo, pero esa simple idea sostiene casi todo el software que usas. Tu lista de contactos es un array de personas. Tu historial de videos es un array de videos. Los resultados de una búsqueda son un array de resultados. En JavaScript, un array se crea con corchetes: `const frutas = ["manzana", "pera", "uva"];`. Dentro de los corchetes pones los elementos separados por comas. Puedes crear un array vacío y llenarlo después: `const lista = [];`. Puedes mezclar tipos: `const mixto = ["texto", 42, true];`, aunque ya hablaremos de si conviene.

La característica más famosa y más confusa de los arrays es que los índices empiezan en cero. El primer elemento está en la posición 0, el segundo en la 1, el tercero en la 2. Para alguien nuevo esto se siente antinatural: ¿por qué el primero es el cero? La razón es histórica y técnica: en los lenguajes de bajo nivel, el índice representa el desplazamiento desde el inicio de la memoria, y el primer elemento está a desplazamiento cero. JavaScript heredó esa convención de C y Java. No es la única opción posible, algunos lenguajes empiezan en uno, pero en JavaScript es cero y punto. La consecuencia práctica es el famoso error "off by one": querer el tercer elemento y pedir el índice 3, que en realidad es el cuarto. La regla mnemotécnica: el índice del último elemento siempre es `length - 1`. Si un array tiene 5 elementos, van de 0 a 4. Si intentas acceder al índice 5, obtienes `undefined`, no un error. JavaScript no te regaña; simplemente te da `undefined` y sigue, lo cual puede generar errores silenciosos difíciles de rastrear. Cuidado con eso.

Acceder a un elemento se hace con corchetes y el índice: `frutas[0]` es "manzana", `frutas[2]` es "uva". Modificar es igual de directo: `frutas[1] = "durazno";` cambia "pera" por "durazno". También puedes agregar elementos asignando a un índice que no existe: `frutas[3] = "mango";` agrega "mango" al final. Pero ojo: si asignas a un índice muy lejano, como `frutas[10] = "kiwi";` en un array de 4 elementos, JavaScript crea huecos vacíos en medio. El array tendrá `length 11` con varios espacios vacíos. Esos huecos son una fuente de comportamientos raros; mejor usa los métodos apropiados para agregar, que veremos ahora.

La propiedad `length` te dice cuántos elementos tiene el array: `frutas.length`. Es una propiedad, no un método, así que no lleva paréntesis. Un error común de principiante es escribir `frutas.length()` con paréntesis y obtener un error de "not a function". La propiedad `length` es tu mejor amiga para recorrer arrays: el último índice válido es `length - 1`, y un ciclo `for` que vaya de 0 hasta `length - 1` visita todos los elementos. También puedes usar `length` para vaciar un array asignándole cero: `frutas.length = 0;` deja el array vacío. Es un truco poco conocido pero válido.

Ahora los métodos esenciales. Piensa en un array como una fila de personas. `push` agrega una persona al final de la fila: `frutas.push("sandía");`. Es el método más usado para agregar elementos, rapidísimo y claro. `pop` saca a la última persona de la fila y te la devuelve: `const ultima = frutas.pop();`. Fíjate que `pop` hace dos cosas: modifica el array quitando el último elemento, y devuelve ese elemento para que lo uses. Esta dualidad es importante: muchos métodos de array modifican el original y devuelven algo útil. `shift` hace lo mismo que `pop` pero al inicio: quita el primer elemento y recorre todos los demás un lugar. `unshift` agrega al inicio: `frutas.unshift("fresa");` pone "fresa" en la posición 0 y recorre todo lo demás. Una analogía útil: `push` y `pop` trabajan con el final de la fila, que es barato y rápido; `shift` y `unshift` trabajan con el inicio, lo que obliga a recorrer a todos los demás, así que son un poco más costosos. En arrays pequeños no notarás la diferencia, pero es bueno saberlo para cuando manejes listas enormes.

¿Cómo decides cuál usar? Si estás construyendo una lista en orden, `push` es tu método: vas agregando al final. Si implementas una pila, como el historial de "deshacer", usas `push` para agregar y `pop` para quitar: el último en entrar es el primero en salir. Si implementas una cola, como la fila de un banco, usas `push` para agregar al final y `shift` para atender al primero: el primero en entrar es el primero en salir. Estas dos estructuras, pila y cola, aparecen en entrevistas de trabajo y en problemas reales, y con `push`, `pop` y `shift` ya las tienes dominadas.

Recorrer un array significa visitar cada elemento, típicamente para hacer algo con él: mostrarlo, transformarlo, buscarlo, sumarlo. La forma clásica es el ciclo `for` con índice: `for (let i = 0; i < frutas.length; i++) {console.log(frutas[i]);}`. Memoriza este patrón porque lo vas a escribir cientos de veces. El índice `i` empieza en 0, el ciclo sigue mientras `i` sea menor que `length`, y en cada vuelta `i` aumenta en 1. Dentro del cuerpo, `frutas[i]` es el elemento actual. Errores típicos aquí: usar `<=` en vez de `<` en la condición, lo que hace que el ciclo intente leer un índice inexistente y trabaje con `undefined` en la última vuelta; olvidar el `i++` y crear un ciclo infinito; o declarar `i` con `var` en vez de `let` y tener problemas de ámbito en código más complejo. En la próxima unidad veremos `forEach` y `map`, que son formas más modernas de recorrer, pero el `for` clásico sigue siendo fundamental y a veces necesario.

Hablemos de los arrays mixtos: JavaScript permite `["texto", 42, true, null]` sin quejarse. ¿Es buena idea? Depende. Para datos estructurados donde cada posición tiene un significado fijo, como `["Ana", 25, true]` donde 0 es nombre, 1 es edad y 2 es activa, funciona pero es frágil: si alguien cambia el orden, todo se rompe silenciosamente. Para esos casos, los objetos que veremos en la unidad 6 son mejores, porque nombran cada dato. Los arrays mixtos tienen sentido cuando la lista es genuinamente heterogénea, como los argumentos de una función o una fila de datos tabulares. La buena práctica: si los elementos representan lo mismo, como

una lista de nombres o de precios, que sean todos del mismo tipo; si cada posición significa algo distinto, considera usar un objeto.

Y llegamos a la trampa clásica: copiar contra referenciar. Cuando haces `const a = [1, 2, 3]; const b = a;`, NO estás copiando el array. Estás creando una segunda etiqueta que apunta al mismo array en memoria. Si haces `b.push(4);`, entonces `a` también "cambia", porque `a` y `b` son el mismo array con dos nombres. Esto confunde muchísimo al principio porque con valores simples no pasa: `const x = 5; const y = x; y = 10;` no cambia `x`. La diferencia es que los números y textos son valores primitivos que se copian, mientras que los arrays son objetos que se comparten por referencia. La analogía: un primitivo es como fotocopiar un documento, cada quien tiene su copia; un array es como compartir el enlace a un documento en la nube, todos editan el mismo.

¿Cómo se copia un array de verdad? La forma moderna y más legible es el operador de propagación: `const copia = [...original];`. Los tres puntitos "desparraman" los elementos en un array nuevo. También puedes usar `original.slice();` sin argumentos, que devuelve una copia superficial. Ojo con la palabra "superficial": si el array contiene objetos, la copia comparte esos objetos internos. Para arrays de números y textos, `[...original]` es suficiente y es lo que usarás el noventa por ciento de las veces. ¿Cuándo importa esto? Cuando pasas un array a una función que lo modifica, o cuando quieres guardar el estado anterior antes de un cambio, o cuando trabajas con listas en interfaces donde cada cambio debe ser intencional. Un bug clásico: una función que ordena o filtra "en su copia" pero en realidad muta el original porque nunca lo copió, y de pronto datos en otra parte de la app cambian misteriosamente. Ahora ya sabes por qué pasa y cómo evitarlo.

Preguntas frecuentes que vale la pena responder. ¿Un array puede crecer sin límite? En la práctica sí, hasta los límites de memoria; no necesitas declararle un tamaño. ¿Qué pasa si comparo dos arrays con `===`? `const a = [1,2]; const b = [1,2]; a === b` es `false`, porque `===` compara referencias, no contenido. Dos arrays distintos nunca son `===` aunque tengan lo mismo dentro. Para comparar contenido hay que hacerlo elemento por elemento o convertir a texto. ¿Los arrays tienen métodos para buscar? Sí, pero los veremos más adelante; por ahora con recorrer y comparar tienes suficiente. ¿Puedo guardar funciones dentro de un array? Sí, las funciones son valores como cualquier otro: `() => 1, () => 2` es válido, aunque es un patrón avanzado.

¿Sabías que `length` no solo se lee sino que se puede escribir para recortar un array? Si tienes `const n = [1,2,3,4,5];` y haces `n.length = 3;`, el array queda `[1,2,3]`. Es un truco de veterano para recortar listas. Y al revés, si aumentas `length`, creas huecos vacíos, cosa que casi nunca quieres hacer a propósito.

Cuidado con modificar un array mientras lo recorres. Si dentro de un `for` haces `push` o `splice`, los índices se mueven y puedes saltarte elementos o crear un ciclo infinito. La regla: si necesitas transformar la lista mientras la recorres, trabaja sobre una copia o construye un array nuevo con los resultados. Este consejo te va a ahorrar horas de depuración.

Para cerrar la explicación, conectemos los dos temas de la unidad. Las funciones flecha y los arrays son la pareja dinámica del JavaScript moderno: las flechas cortas son perfectas para operar sobre listas. Cuando en la próxima unidad veas `forEach`, `map` y `filter`, vas a pasarles funciones flecha como argumentos, y todo va a encajar: la sintaxis compacta de la flecha brilla justo cuando la aplicas a cada elemento de una lista. Dominar estos dos temas por separado es el prerequisite; combinarlos es el superpoder. Y ese superpoder es exactamente lo que usan los programadores profesionales todos los días para procesar datos: tomar una lista, transformarla con funciones pequeñas y producir resultados. Estás aprendiendo el corazón del oficio.

Profundicemos ahora en algunos casos borde de las funciones flecha que separan al que "más o menos las usa" del que realmente las domina. Primer caso borde: las flechas no pueden usarse como constructoras. Si intentas `const P = () => {}; new P();`, JavaScript lanza un error. Las funciones flecha no tienen `prototype` propio ni pueden crear objetos con `new`. En la práctica moderna casi nadie crea objetos con `new` para casos simples, pero si lees código viejo o trabajas con clases, es bueno saberlo. Segundo caso: las flechas no tienen su propio `arguments`. Si dentro de una flecha escribes `arguments` esperando los argumentos de la flecha, obtendrás los `arguments` de la función contenedora, o un error si no hay ninguna. La alternativa moderna es el parámetro `rest`: `(...args) => ...` te da un array real con todos los argumentos, que además es más cómodo que el objeto `arguments`.

Tercer caso borde: la precedencia y los paréntesis alrededor de la flecha completa. Cuando pasas una flecha como argumento, no necesitas paréntesis extra: `hacerAlgo(x => x * 2)` está bien. Pero cuando la flecha es parte de una expresión más grande, a veces necesitas agruparla: `const resultado = (x => x * 2)(5)`; llama a la flecha inmediatamente con 5 y guarda 10. Sin los paréntesis externos, la sintaxis se vuelve ambigua y truena. Este patrón de "función flecha ejecutada inmediatamente" es raro en código cotidiano, pero aparece en ejemplos y pruebas; ahora sabes leerlo.

Cuarto caso: flechas multilínea con `return` implícito. Puedes escribir `const sumar = (a, b) => a + b`; en una línea, pero también puedes partir la expresión en varias líneas siempre que no pongas llaves ni punto y coma prematuro: `const total = (precios) => precios.reduce((acc, p) => acc + p, 0)`; JavaScript permite saltos de línea dentro de la expresión. Pero cuidado con el punto y coma automático: si escribes `return` en una línea y el valor en la siguiente dentro de

una función con llaves, JavaScript inserta un punto y coma invisible y la función devuelve undefined. Con el return implícito sin llaves no hay ese problema porque no hay palabra return. Es otro punto a favor de la sintaxis compacta cuando el caso lo permite.

Hablemos de estilo y buenas prácticas con flechas, porque el código también se lee por humanos. Buena práctica uno: si la flecha cabe en una línea y se entiende, déjala en una línea. `const doble = n => n * 2;` es perfecto. Buena práctica dos: si la flecha necesita más de dos o tres operaciones, usa llaves y return explícito, o mejor aún, conviértela en función con nombre tradicional. Una flecha de quince líneas es una señal de que algo se puede dividir en funciones más pequeñas. Buena práctica tres: dale nombres descriptivos a la constante que guarda la flecha. `const f = x => x * 2;` no dice nada; `const duplicar = x => x * 2;` se explica solo. Buena práctica cuatro: no abuses de anidar flechas dentro de flechas dentro de flechas. Dos niveles se leen bien; cuatro niveles son un acertijo. Si ves que anidas mucho, extrae funciones con nombre.

Una anécdota típica de la vida real: un estudiante me mostró su código para calcular el promedio de calificaciones y era una sola línea larguísima con tres flechas anidadas, un ternario dentro de otro ternario y cero espacios. Funcionaba, pero ni él entendía qué hacía dos días después. Lo reescribimos en cinco líneas con nombres claros y de pronto el código "hablaba". La moraleja: la brevedad de las flechas es para eliminar ruido, no para comprimir la lógica hasta hacerla ilegible. El mejor código es el que un colega entiende a la primera leída.

Comparemos código malo y código bueno con flechas para que veas la diferencia en acción. Código malo: `const p = (a,b) => {let r = a + b; return r};`. Funciona, pero declara una variable temporal innecesaria y usa llaves pudiendo no usarlas. Código bueno: `const sumar = (a, b) => a + b;`. Mismo resultado, la mitad de ruido. Otro ejemplo. Código malo: `const esMayor = (edad) => {if (edad >= 18) {return true;} else {return false;}};`. Código bueno: `const esMayorDeEdad = edad => edad >= 18;`. La comparación ya devuelve true o false, no necesitas el if. Este patrón de "devolver la comparación directamente" es uno de los hábitos que más limpian el código de un principiante.

Ahora profundicemos en los métodos de array con más detalle del que dimos en la introducción, porque cada uno tiene sus sutilezas. `push` puede agregar varios elementos a la vez: `arr.push("a", "b", "c");` agrega los tres al final. Y devuelve la nueva longitud del array, no el array. `const nuevaLongitud = arr.push("x");` guarda un número. Mucha gente espera que devuelva el array modificado y se confunde. `pop` no recibe argumentos: siempre quita el último. Si el array está vacío, `pop` devuelve undefined y no truena; es amable pero silencioso, así que si tu lógica depende de que haya elementos, verifica `length` antes.

shift y unshift son los primos del inicio. `const primero = arr.shift()`; quita y devuelve el primero. `arr.unshift("nuevo")`; agrega al inicio y devuelve la nueva longitud, igual que push. Un detalle de rendimiento: en arrays gigantes, shift y unshift son más lentos porque tienen que reindexar todos los elementos. Para la mayoría de tus programas no importará, pero si algún día procesas listas de cientos de miles de elementos en un ciclo caliente, prefiere trabajar por el final con push y pop. Es el tipo de conocimiento que no necesitas hoy pero que te hace ver profesional cuando lo mencionas.

¿Sabías que puedes usar push con el operador de propagación para agregar todos los elementos de un array a otro? `const a = [1, 2]; const b = [3, 4]; a.push(...b)`; deja a como [1, 2, 3, 4]. Los puntitos desparraman b dentro de los argumentos de push. Es un truco elegante para concatenar sin crear un array nuevo. Y hablando de concatenar, existe el método concat que no modifica los originales sino que devuelve un array nuevo: `const c = a.concat(b)`; Si quieres preservar los originales intactos, concat o `[...a,...b]` son tus herramientas.

El método slice merece mención especial porque es la navaja suiza para copiar porciones. `arr.slice(1, 3)` devuelve un array nuevo con los elementos de los índices 1 y 2; el índice final no se incluye. `arr.slice(2)` devuelve desde el índice 2 hasta el final. `arr.slice(-2)` devuelve los últimos dos elementos; los índices negativos cuentan desde el final. Y `arr.slice()` sin argumentos devuelve una copia completa del array. Memoriza slice como "el que copia sin modificar": es la herramienta segura cuando quieres un pedazo sin tocar el original. Su primo peligroso es splice, que sí modifica el original; no lo necesitas para esta unidad, pero cuando lo veas en código ajeno, recuerda: splice muta, slice no.

Otro error clásico con índices: confundir el índice con el valor. Si tienes `const edades = [20, 25, 30]`; y quieres imprimir cada edad, el ciclo usa i como índice y `edades[i]` como valor. El principiante a veces imprime i pensando que imprime la edad, y ve 0, 1, 2 en vez de 20, 25, 30. Parece obvio cuando lo lees, pero en un programa de cien líneas con variables llamadas de forma parecida, pasa más de lo que crees. Un hábito que ayuda: nombra la variable del índice de forma que recuerde su rol, o mejor, usa nombres descriptivos para el array para que `edades[i]` se lea natural.

Hablemos de recorrer arrays con más profundidad, porque "recorrer" no es una sola operación: hay varios patrones según lo que quieras lograr. Patrón uno: hacer algo con cada elemento, como imprimirlo. for clásico o, más adelante, `forEach`. Patrón dos: transformar cada elemento y guardar los resultados en un array nuevo, como duplicar cada número. Se hace creando un array vacío, recorriendo el original con for, aplicando la transformación y haciendo push del resultado. En la próxima unidad verás que map hace exactamente esto en una línea. Patrón tres: filtrar, quedarte solo con los elementos que cumplen una condición. Igual: array nuevo,

for, if adentro, push solo si cumple. La próxima unidad te presentará filter. Patrón cuatro: acumular, como sumar todos los números o concatenar textos. Se usa una variable acumuladora fuera del ciclo que se va actualizando adentro. Patrón cinco: buscar, como encontrar el primer elemento que cumple algo o su índice. Se recorre y se rompe el ciclo con break cuando se encuentra. Estos cinco patrones cubren el noventa por ciento de lo que harás con listas en tu carrera. Si los practicas hasta que salgan solos, la unidad 6 te va a parecer un paseo.

El patrón de acumulación merece un ejemplo mental detallado porque es donde más se atoran. Quieres sumar [10, 20, 30]. Declaras let suma = 0; ANTES del ciclo. Dentro del ciclo haces suma = suma + numeros[i]; o suma += numeros[i];. Al terminar, suma vale 60. Los errores típicos: declarar suma dentro del ciclo, con lo que se reinicia a cero en cada vuelta y al final solo tienes el último valor; o inicializarla en un valor incorrecto, como 1 en vez de 0 para una suma, o "" mal manejado para concatenar; o olvidar actualizarla. La regla: el acumulador nace antes del ciclo, vive actualizándose dentro, y se usa después. Para multiplicación el valor inicial es 1, no 0, porque todo multiplicado por 0 da 0; ese detalle ha arruinado más de una tarea.

Veamos ahora los arrays mixtos con ojos críticos. JavaScript no te impide mezclar, y hay casos legítimos: una fila de una tabla donde cada columna es de un tipo distinto, o una lista de argumentos variados. Pero la mayoría de las veces que un principiante crea un array mixto, lo que realmente quiere es un objeto. Ejemplo: const persona = ["Ana", 25, "México"];. Funciona, pero ¿qué significa persona[1]? Tienes que recordar que el 1 es la edad. Si mañana agregas el apellido al inicio, todos los índices se recorren y el código que usaba persona[1] para la edad ahora lee el apellido. En cambio, con un objeto persona.edad siempre es la edad. La lección: los arrays son para listas de cosas del mismo tipo donde el orden es lo que importa; los objetos son para paquetes de datos distintos donde el nombre es lo que importa. Esta distinción te va a guiar en cientos de decisiones de diseño.

Cuidado con los valores "falsy" dentro de arrays cuando filtres o busques. El 0, el texto vacío "", null, undefined, NaN y false son falsy: se comportan como false en un if. Si tienes const datos = [0, 5, "", "hola"]; y filtras con if (datos[i]), eliminas el 0 y el "" aunque sean datos válidos. Si el cero es una calificación legítima o el texto vacío es una respuesta válida, tu filtro los borra por accidente. La lección: cuando la ausencia de valor importa, compara explícitamente, como datos[i] !== undefined, en vez de confiar en la verdad implícita. Este es un bug sutil que aparece en sistemas reales, por ejemplo al filtrar formularios donde el usuario dejó un campo en cero a propósito.

La copia contra referencia merece una segunda vuelta con más profundidad porque es un concepto que reaparece en objetos, funciones y todo el lenguaje. En JavaScript hay dos familias de valores: los primitivos, que son números, textos, booleanos, null, undefined y símbolos, y que se copian por valor; y los objetos, donde entran los arrays, las funciones y los objetos literales, que se comparten por referencia. Cuando asignas un primitivo, la variable guarda el valor. Cuando asignas un array, la variable guarda una referencia, como la dirección de una casa, no la casa misma. Por eso `const b = a` no copia: le da a b la misma dirección.

Esto tiene consecuencias en funciones. Si pasas un array a una función y la función hace push, el array original cambia, porque la función recibió la referencia al mismo array. A veces eso es lo que quieres: una función que agrega un elemento a una lista compartida. A veces no: una función que "calcula algo con la lista" no debería modificarla como efecto secundario. La buena práctica es ser explícito: si tu función va a modificar el array, que se note en el nombre, como `agregarAlFinal(lista, elemento)`; si solo va a leerlo, no lo modifiques, y si necesitas una versión modificada, crea y devuelve una copia. Los efectos secundarios sorpresa son la fuente número uno de bugs "fantasma" donde los datos cambian sin que nadie sepa quién los cambió.

¿Cómo copiar bien? Para arrays planos de primitivos, `[...original]` es perfecto y es el estándar moderno. Para copiar una porción, `slice`. Para copiar y transformar a la vez, `map`, que verás en la unidad 6. Existe también `JSON.parse(JSON.stringify(array))` como truco para copia profunda, que duplica hasta los objetos internos, pero tiene limitaciones: pierde funciones, convierte undefined y maneja mal fechas y valores especiales. Menciónalo como curiosidad, no como herramienta diaria. En esta etapa de tu aprendizaje, `[...original]` cubre tus necesidades; la copia profunda es un tema avanzado para cuando trabajes con estructuras anidadas complejas.

Un mini-reto para verificar que entendiste referencias: `const a = [1, 2]; const b = a; b[0] = 99;` ¿Cuánto vale `a[0]`? Vale 99, porque a y b son el mismo array. Segundo reto: `const a = [1, 2]; const b = [...a]; b[0] = 99;` ¿Cuánto vale `a[0]`? Vale 1, porque b es una copia independiente. Si respondiste bien las dos sin dudar, el concepto ya es tuyo. Si dudaste, relee esta sección y haz los experimentos en la consola: no hay mejor maestro que probar y ver.

Para cerrar esta explicación larga, hablemos de cómo estos temas se conectan con tu futuro como programador. Las funciones flecha son la sintaxis que verás en React, en Node.js, en ejemplos de APIs, en código de entrevistas técnicas. Los arrays son la estructura con la que vas a modelar casi cualquier colección de datos. Juntos forman el vocabulario básico del procesamiento de datos en JavaScript: listas que entran, funciones pequeñas que las transforman, resultados que salen. Cada vez que veas código moderno y te parezca críptico,

casi seguro es una combinación de flechas y métodos de array que aún no conoces. Después de esta unidad y la siguiente, ese código va a empezar a leerse solo. Es como aprender a leer partituras: al principio son puntitos, después es música.

Ejemplos paso a paso

Vamos a construir ejemplos de menor a mayor complejidad, cada uno comentado para que puedas seguir el razonamiento. Te recomiendo que no solo los leas: escríbelos en tu editor o en la consola del navegador y ejecútalos. La diferencia entre leer código y escribir código es como la diferencia entre ver a alguien nadar y meterte a la alberca.

Ejemplo 1: tu primera función flecha. Vamos a convertir una función tradicional en flecha paso a paso para que veas la mecánica del recorte.

```
const saludar = (nombre) => {  
  return "Hola, " + nombre;  
};  
  
console.log(saludar("Ana"));
```

Esta es la forma completa de la flecha: parámetros entre paréntesis, llaves y return explícito. Funciona igual que la función tradicional. Ahora apliquemos la primera abreviación: como hay un solo parámetro, quitamos los paréntesis.

```
const saludar = nombre => {  
  return "Hola, " + nombre;  
};  
  
console.log(saludar("Ana"));
```

Sigue funcionando igual. Ahora la segunda abreviación: el cuerpo es una sola expresión, así que quitamos llaves y return, y el valor se devuelve automáticamente.

```
const saludar = nombre => "Hola, " + nombre;  
  
console.log(saludar("Ana"));
```

Tres líneas se volvieron una, y se lee perfecto: "saludar toma un nombre y produce Hola, más el nombre". Este proceso de recorte es el que vas a aplicar mentalmente cada vez que escribas una flecha: empieza completa si te da seguridad y recorta cuando veas que se puede.

Ejemplo 2: flechas con varios parámetros y valores por defecto.

```
const presentar = (nombre = "visitante", edad = 0) => {  
  return nombre + " tiene " + edad + " años";  
};  
  
console.log(presentar("Luis", 30));  
console.log(presentar("Marta"));  
console.log(presentar());
```

Observa cómo los valores por defecto hacen la función amable: si llamas `presentar("Marta")` sin edad, edad vale 0 en vez de `undefined`. Si la llamas sin nada, usa ambos valores por defecto. Este patrón es oro en funciones de utilidad que otros van a usar, porque evita que un olvido produzca `NaN` o `undefined` en la pantalla del usuario.

Ejemplo 3: el error clásico del `return` implícito, para que lo veas fallar a propósito y lo reconozcas cuando te pase de verdad.

```
const duplicarBien = n => n * 2;  
const duplicarMal = n => {  
  n * 2;  
};  
  
console.log(duplicarBien(5));  
console.log(duplicarMal(5));
```

La primera imprime 10. La segunda imprime `undefined`, porque las llaves desactivan el `return` implícito y la expresión `n * 2` se calcula y se desecha. Cuando veas una función que "debería devolver algo pero devuelve `undefined`", lo primero que revisas es: ¿tiene llaves sin `return`? Nueve de cada diez veces, ahí está el problema.

Ejemplo 4: devolver un objeto con una flecha, el truco de los paréntesis.

```
const crearUsuarioMal = nombre => {  
  nombre: nombre;  
};  
  
const crearUsuarioBien = nombre => ({  
  nombre: nombre,  
  activo: true  
});  
  
console.log(crearUsuarioMal("Ana"));  
console.log(crearUsuarioBien("Ana"));
```

La versión mal devuelve undefined porque JavaScript interpreta las llaves como el cuerpo de la función. La versión bien envuelve el objeto en paréntesis y devuelve el objeto correctamente. Este ejemplo vale oro: guárdalo en tu memoria porque te va a salir en pruebas, entrevistas y código real.

Ejemplo 5: crear y acceder a un array, lo fundamental.

```
const colores = ["rojo", "verde", "azul"];  
  
console.log(colores[0]);  
console.log(colores[1]);  
console.log(colores[2]);  
console.log("Total de colores:", colores.length);
```

El índice 0 es el primero, el 1 el segundo, el 2 el tercero. length vale 3 aunque el último índice sea 2. Repite en voz alta: "el último índice es length menos uno". Esa frase te va a salvar de incontables errores.

Ejemplo 6: modificar elementos y agregar con índices.

```
const tareas = ["lavar", "cocinar"];  
tareas[1] = "cocinar pasta";  
tareas[2] = "barrer";  
  
console.log(tareas);  
console.log(tareas.length);
```

Cambiamos "cocinar" por "cocinar pasta" con el índice 1, y agregamos "barrer" en el índice 2 que no existía. El array crece solo. Funciona, pero recuerda: para agregar al final, push es más claro y más seguro que calcular el índice a mano.

Ejemplo 7: push y pop en acción, la pila de platos.

```
const platos = [];  
  
platos.push("plato hondo");  
platos.push("plato plano");  
platos.push("taza");  
  
console.log(platos);  
  
const ultimo = platos.pop();  
console.log("Quité:", ultimo);  
console.log(platos);  
console.log("Quedan:", platos.length);
```

Imagina una pila de platos en la cocina: push pone cada plato encima, pop quita el de hasta arriba y te lo entrega en la mano. El console.log final muestra que quedan dos platos. Este patrón de pila aparece en el botón "deshacer" de los editores, en el historial del navegador y en muchos algoritmos.

Ejemplo 8: shift y unshift, la fila del banco.

```
const fila = ["Ana", "Luis"];  
  
fila.unshift("Marta");  
console.log(fila);  
  
const atendido = fila.shift();  
console.log("Atendiendo a:", atendido);  
console.log(fila);
```

unshift mete a Marta al inicio de la fila, y shift atiende al primero, que ahora es Marta, sacándola de la fila. Así funcionan las colas: entra por atrás con push, sale por adelante con shift. Los sistemas de turnos, las listas de reproducción y las colas de impresión usan esta lógica.

Ejemplo 9: recorrer un array con for, el patrón que debes memorizar.

```
const calificaciones = [8, 9, 7, 10, 6];

for (let i = 0; i < calificaciones.length; i++) {
  console.log("Calificación " + (i + 1) + ": " + calificaciones[i]);
}
```

El ciclo va de 0 a length - 1, y en cada vuelta calificaciones[i] es el elemento actual. Nota el (i + 1) para mostrarle al usuario números "humanos" que empiezan en 1, mientras por dentro trabajamos con índices desde 0. Separar el índice interno de la numeración visible es una buena práctica en interfaces.

Ejemplo 10: el patrón de acumulación, sumar todos los elementos.

```
const precios = [120, 80, 200, 50];
let total = 0;

for (let i = 0; i < precios.length; i++) {
  total = total + precios[i];
}

console.log("El total es: " + total);
```

El acumulador total nace en 0 antes del ciclo, se actualiza en cada vuelta sumando el precio actual, y al final vale 450. Si declararas let total = 0 dentro del ciclo, se reiniciaría cada vez. Si lo inicializaras en 1, el total saldría mal. El acumulador fuera, la actualización dentro, el uso después: ese es el ritmo.

Ejemplo 11: construir un array nuevo transformando, el antecesor manual de map.

```
const numeros = [1, 2, 3, 4];
const dobles = [];

for (let i = 0; i < numeros.length; i++) {
  dobles.push(numeros[i] * 2);
}

console.log(numeros);
console.log(dobles);
```

Creamos dobles vacío, recorremos numeros, y por cada elemento hacemos push de su doble. El original queda intacto y el nuevo tiene [2, 4, 6, 8]. Este es exactamente el patrón que map

automatiza en la próxima unidad; entenderlo a mano primero hace que map se sienta obvio después.

Ejemplo 12: filtrar manualmente, el antecesor de filter.

```
const edades = [15, 22, 17, 30, 12, 25];
const mayores = [];

for (let i = 0; i < edades.length; i++) {
  if (edades[i] >= 18) {
    mayores.push(edades[i]);
  }
}

console.log(mayores);
```

Recorremos, preguntamos con if si cada edad es mayor o igual a 18, y solo las que pasan la prueba entran al array nuevo con push. El resultado es [22, 30, 25]. Nota cómo la flecha podría limpiar la condición, pero aquí lo importante es el patrón: nuevo array, ciclo, condición, push selectivo.

Ejemplo 13: buscar un elemento y su índice.

```
const invitados = ["Ana", "Luis", "Marta", "Pedro"];
const buscado = "Marta";
let posicion = -1;

for (let i = 0; i < invitados.length; i++) {
  if (invitados[i] === buscado) {
    posicion = i;
    break;
  }
}

if (posicion === -1) {
  console.log("No está en la lista");
} else {
  console.log(buscado + " está en la posición " + posicion);
}
```

Inicializamos posicion en -1 como señal de "no encontrado", porque -1 nunca es un índice válido. Recorremos comparando cada elemento con ===, y al encontrarlo guardamos el índice

y rompemos el ciclo con `break` para no seguir buscando en vano. Después decidimos el mensaje según si `posicion` sigue en `-1`. Este patrón de "bandera inicial más `break`" es estándar para búsquedas.

Ejemplo 14: referencia contra copia, el experimento que debes hacer.

```
const original = [1, 2, 3];
const referencia = original;
const copia = [...original];

referencia.push(4);
copia.push(5);

console.log("original:", original);
console.log("referencia:", referencia);
console.log("copia:", copia);
```

`referencia.push(4)` modifica también a `original`, porque son el mismo array: verás `[1, 2, 3, 4]` en ambos. En cambio `copia` es independiente gracias a `[...original]`: queda `[1, 2, 3, 5]` sin afectar al `original`. Ejecuta este ejemplo y mira los tres `console.log`: entenderlo con los ojos es cien veces mejor que solo leerlo.

Ejemplo 15: mini programa completo, gestor de lista de compras que combina todo.

```

const listaCompras = [];

const agregarProducto = producto => {
  listaCompras.push(producto);
  console.log(producto + " agregado. Total: " + listaCompras.length);
};

const quitarUltimo = () => {
  const quitado = listaCompras.pop();
  console.log(quitado + " eliminado. Total: " + listaCompras.length);
};

const mostrarLista = () => {
  console.log("=== Mi lista ===");
  for (let i = 0; i < listaCompras.length; i++) {
    console.log((i + 1) + ". " + listaCompras[i]);
  }
};

agregarProducto("leche");
agregarProducto("pan");
agregarProducto("huevos");
mostrarLista();
quitarUltimo();
mostrarLista();

```

Tres funciones flecha trabajan sobre el mismo array: `agregarProducto` usa `push`, `quitarUltimo` usa `pop`, `mostrarLista` recorre con `for`. Es un programa pequeño pero completo y real: tiene estado, tiene operaciones y tiene salida legible. Si entiendes cada línea de este ejemplo, ya sabes usar flechas y arrays juntos, que es la meta de la unidad. Como reto extra, agrégale una función `quitarPrimero` que use `shift`, y una función `buscarProducto` que use el patrón de búsqueda del ejemplo 13.

Práctica

Es hora de ensuciarte las manos. Estos ejercicios están ordenados de menor a mayor dificultad. Intenta cada uno por tu cuenta antes de mirar la solución: la lucha es donde ocurre el aprendizaje. Si te atorras más de quince minutos, mira la solución, entiéndela línea por línea, ciérrala y vuelve a escribirla sin ver. Ese ciclo de intentar, revisar y reescribir es la forma más rápida de que el conocimiento se quede.

Ejercicio 1: conversor con flecha. Enunciado: crea una función flecha llamada `aFahrenheit` que reciba una temperatura en grados Celsius y devuelva su equivalente en Fahrenheit. La fórmula es: $Fahrenheit = Celsius * 9/5 + 32$. Usa la sintaxis más corta posible. Pruébala con 0, 25 y 100 grados.

```
const aFahrenheit = c => c * 9 / 5 + 32;

console.log(aFahrenheit(0));
console.log(aFahrenheit(25));
console.log(aFahrenheit(100));
```

Explicación paso a paso: primero definimos la constante `aFahrenheit` y le asignamos la flecha. Como hay un solo parámetro, quitamos los paréntesis: `c =>`. El cuerpo es una sola expresión, la fórmula, así que quitamos llaves y `return`: el resultado se devuelve automáticamente. La fórmula `c * 9 / 5 + 32` respeta el orden de operaciones: primero multiplica y divide de izquierda a derecha, luego suma 32. Los resultados son 32, 77 y 212, que son los puntos de congelación, un día caluroso y la ebullición del agua. Si tu resultado con 100 no es 212, revisa el orden de las operaciones.

Ejercicio 2: el último de la lista. Enunciado: escribe una función flecha llamada `ultimoElemento` que reciba un array y devuelva su último elemento sin modificar el array. Debe funcionar con arrays de cualquier tamaño y no debe tronar con un array vacío: en ese caso debe devolver `undefined` de forma natural.

```
const ultimoElemento = arr => arr[arr.length - 1];

console.log(ultimoElemento(["a", "b", "c"]));
console.log(ultimoElemento([10]));
console.log(ultimoElemento([]));
```

Explicación paso a paso: la clave es que el último índice siempre es `length - 1`. Para `["a","b","c"]`, `length` es 3, así que pedimos el índice 2, que es "c". Para `[10]`, `length` es 1, pedimos el índice 0, que es 10. Para el array vacío, `length` es 0, pedimos el índice -1, que no existe, y JavaScript devuelve `undefined` sin lanzar error. La función usa `return` implícito en una sola línea. Nota que no modificamos el array: solo leemos. Este es un buen ejemplo de función "pura": recibe datos, devuelve un resultado y no cambia nada.

Ejercicio 3: agregar solo si no existe. Enunciado: crea una función flecha `agregarUnico` que reciba un array y un elemento, y agregue el elemento al final solo si todavía no está en el array. Debe devolver el array modificado. Usa `push` y el patrón de búsqueda con un ciclo `for`.

```
const agregarUnico = (lista, elemento) => {
  let existe = false;
  for (let i = 0; i < lista.length; i++) {
    if (lista[i] === elemento) {
      existe = true;
      break;
    }
  }
  if (!existe) {
    lista.push(elemento);
  }
  return lista;
};

const colores = ["rojo", "azul"];
agregarUnico(colores, "verde");
agregarUnico(colores, "rojo");
console.log(colores);
```

Explicación paso a paso: aquí la flecha necesita llaves porque el cuerpo tiene varias instrucciones: no aplica el return implícito. Empezamos con una bandera existe en false. Recorremos la lista comparando cada elemento con ===; si encontramos el elemento, prendemos la bandera y rompemos el ciclo con break porque ya no hay nada que buscar. Después del ciclo, si la bandera sigue en false, hacemos push del elemento. Devolvemos la lista con return explícito. Al probar, "verde" sí se agrega porque no estaba, pero "rojo" no se duplica. El resultado final es ["rojo", "azul", "verde"]. Este patrón de "bandera más break" lo vas a reutilizar muchísimo.

Ejercicio 4: promedio con acumulador y flecha. Enunciado: escribe una función flecha llamada promedio que reciba un array de números y devuelva su promedio. Si el array está vacío, debe devolver 0 en vez de NaN. Usa un ciclo for con acumulador.

```
const promedio = numeros => {
  if (numeros.length === 0) {
    return 0;
  }
  let suma = 0;
  for (let i = 0; i < numeros.length; i++) {
    suma += numeros[i];
  }
  return suma / numeros.length;
};

console.log(promedio([10, 8, 9, 7]));
console.log(promedio([]));
```

Explicación paso a paso: lo primero es defender la función del caso borde: si length es 0, dividir entre cero daría NaN, así que devolvemos 0 de inmediato con un return temprano. Esta técnica del "return temprano" mantiene el código limpio: atiendes los casos especiales al inicio y el camino principal queda despejado. Luego el acumulador suma nace en 0 antes del ciclo, se actualiza con `suma += numeros[i]` en cada vuelta, y al final dividimos entre la cantidad de elementos. Con `[10, 8, 9, 7]` la suma es 34 y el promedio 8.5. Con el array vacío, el if inicial nos salva del NaN. Acostúmbrate a preguntarte siempre "¿qué pasa si el array viene vacío?" cuando escribas funciones que reciben listas.

Ejercicio 5: invertir sin mutar. Enunciado: crea una función flecha llamada `invertirCopia` que reciba un array y devuelva un array NUEVO con los elementos en orden inverso, sin modificar el original. No puedes usar el método `reverse` directamente sobre el original porque lo muta; debes construir el resultado.

```
const invertirCopia = arr => {
  const invertido = [];
  for (let i = arr.length - 1; i >= 0; i--) {
    invertido.push(arr[i]);
  }
  return invertido;
};

const original = [1, 2, 3, 4];
const alReves = invertirCopia(original);
console.log(alReves);
console.log(original);
```

Explicación paso a paso: creamos un array nuevo vacío llamado invertido. El ciclo empieza en el último índice, `arr.length - 1`, y va hacia atrás restando 1 en cada vuelta hasta llegar a 0; la condición `i >= 0` incluye el primer elemento. En cada vuelta hacemos `push` del elemento actual al array nuevo. Así el primer elemento que entra es el último del original, y el resultado queda invertido. Devolvemos el array nuevo. La prueba muestra `[4, 3, 2, 1]` para `alReves` y `[1, 2, 3, 4]` intacto para `original`. Este ejercicio combina tres ideas de la unidad: recorrer con índices, construir arrays nuevos con `push`, y no mutar el original. Si lo resolviste sin ver la solución, tienes muy buen nivel.

Ejercicio 6: mayúsculas selectivas. Enunciado: escribe una función flecha llamada `gritarLargos` que reciba un array de palabras y devuelva un array nuevo con las palabras de más de 4 letras convertidas a mayúsculas, dejando las demás igual. Usa `toUpperCase` para convertir.

```
const gritarLargos = palabras => {
  const resultado = [];
  for (let i = 0; i < palabras.length; i++) {
    if (palabras[i].length > 4) {
      resultado.push(palabras[i].toUpperCase());
    } else {
      resultado.push(palabras[i]);
    }
  }
  return resultado;
};

console.log(gritarLargos(["sol", "montaña", "mar", "elefante", "luz"]));
```

Explicación paso a paso: creamos el array `resultado` vacío. Recorremos `palabras` con el `for` clásico. Para cada palabra, preguntamos si su `length` es mayor que 4; nota que aquí `length` se aplica a un texto, no a un array, y nos dice cuántas letras tiene. Si cumple, hacemos `push` de la palabra en mayúsculas con `toUpperCase`; si no, `push` de la palabra tal cual. Devolvemos el array nuevo. El resultado es `["sol", "MONTAÑA", "mar", "ELEFANTE", "luz"]`. Este ejercicio muestra el patrón de transformación condicional: cada elemento pasa por una decisión antes de entrar al resultado. En la próxima unidad harás esto mismo con `map` y un `ternario` en una sola línea, y vas a apreciar la diferencia.

Ejercicio 7: el reto final, mini agenda. Enunciado: usando solo lo aprendido en esta unidad, crea un pequeño programa de agenda de contactos con estas funciones flecha: agregarContacto(nombre) que agregue el nombre al final si no existe ya; eliminarContacto(nombre) que lo quite de la lista si existe; mostrarContactos() que imprima cada contacto numerado desde 1. Usa un array llamado agenda para guardar los nombres.

```
const agenda = [];  
  
const agregarContacto = nombre => {  
  let existe = false;  
  for (let i = 0; i < agenda.length; i++) {  
    if (agenda[i] === nombre) {  
      existe = true;  
      break;  
    }  
  }  
  if (!existe) {  
    agenda.push(nombre);  
    console.log(nombre + " agregado a la agenda");  
  } else {  
    console.log(nombre + " ya estaba en la agenda");  
  }  
};  
  
const eliminarContacto = nombre => {  
  let posicion = -1;  
  for (let i = 0; i < agenda.length; i++) {  
    if (agenda[i] === nombre) {  
      posicion = i;  
      break;  
    }  
  }  
  if (posicion !== -1) {  
    for (let j = posicion; j < agenda.length - 1; j++) {  
      agenda[j] = agenda[j + 1];  
    }  
    agenda.pop();  
    console.log(nombre + " eliminado de la agenda");  
  } else {  
    console.log(nombre + " no está en la agenda");  
  }  
};  
  
const mostrarContactos = () => {
```

```

console.log("=== Agenda ===");
for (let i = 0; i < agenda.length; i++) {
    console.log((i + 1) + ". " + agenda[i]);
}
if (agenda.length === 0) {
    console.log("(vacía)");
}
};

agregarContacto("Ana");
agregarContacto("Luis");
agregarContacto("Ana");
mostrarContactos();
eliminarContacto("Luis");
mostrarContactos();
eliminarContacto("Pedro");

```

Explicación paso a paso: este programa integra todo. `agregarContacto` reutiliza el patrón de "agregar solo si no existe" del ejercicio 3, con mensajes para cada caso. `eliminarContacto` primero busca la posición con el patrón de búsqueda; si la encuentra, recorre los elementos desde esa posición hacia adelante, copiando cada elemento siguiente un lugar atrás, lo que "pisa" al eliminado; al final hace `pop` para quitar el duplicado que quedó al final. Es una eliminación manual que te enseña qué hace un array por dentro cuando quitas un elemento del medio. Si la posición es `-1`, avisa que no existe. `mostrarContactos` recorre e imprime numerando desde 1 para el usuario, y si la agenda está vacía lo dice explícitamente en vez de no mostrar nada, que es un detalle de buena experiencia. La prueba agrega dos contactos, rechaza el duplicado, muestra la lista, elimina a Luis, vuelve a mostrar y avisa que Pedro no existe. Si lograste escribir algo parecido por tu cuenta, estás listo para la unidad 6, donde herramientas como `filter` y `find` harán este mismo programa en la mitad de líneas.

Preguntas frecuentes y curiosidades

¿Las funciones flecha son más rápidas que las normales? En la práctica moderna, no hay una diferencia de rendimiento que deba guiar tu decisión. Los motores de JavaScript optimizan ambas por igual. Elige flecha o función tradicional por legibilidad y por el comportamiento de `this`, nunca por una supuesta velocidad. Quien te diga que las flechas son "más rápidas" está repitiendo un mito de los primeros años.

¿Puedo mezclar flechas y funciones normales en el mismo programa? Por supuesto. De hecho, es lo normal en código profesional: funciones normales para las piezas grandes con

nombre, como `iniciarSesion` o `calcularTotal`, y flechas para las piezas pequeñas, como los callbacks que transforman datos. El código maduro no es "todo flechas" ni "todo function": es cada sintaxis en el lugar donde brilla.

¿Sabías que el nombre `=>` se lee "flecha gorda" en inglés, "fat arrow"? Viene de CoffeeScript, un lenguaje que compilaba a JavaScript y que popularizó la sintaxis. Cuando el comité de JavaScript diseñó la versión moderna del lenguaje, adoptó la flecha gorda porque la comunidad ya la amaba. Así que cada vez que escribes `=>` estás usando una idea que nació en la comunidad antes de entrar al estándar oficial.

Cuidado con las flechas de una sola línea que crecen sin que te des cuenta. Empiezas con `const total = precios => precios.reduce((a, p) => a + p, 0);` y un mes después alguien le mete un descuento, un impuesto y un redondeo, todo encadenado en la misma línea. La línea termina midiendo doscientos caracteres y nadie la entiende. La regla práctica: si tu flecha de una línea necesita más de dos operaciones encadenadas, dale cuerpo con llaves o divídela en pasos con nombres. El código se escribe una vez y se lee cien veces; optimiza para la lectura.

¿Sabías que puedes usar una flecha sin guardarla en ninguna variable? Es lo que se hace al pasar callbacks: `numeros.map(n => n * 2)`. La flecha nace, transforma cada número y desaparece. No necesita nombre porque su contexto la explica: está justo donde se usa. Esta es la razón profunda de que las flechas sean anónimas por diseño.

Cuidado con el operador de propagación en copias: `[...original]` copia el array, pero si los elementos son objetos, los objetos internos se siguen compartiendo. Es la "copia superficial". Para tus ejercicios con números y textos no hay problema, pero cuando trabajes con listas de objetos, como una lista de usuarios donde cada usuario es un objeto, modificar `usuario.nombre` en la copia también cambia el original. La copia profunda real es un tema avanzado; por ahora basta con que sepas que la superficialidad existe para que no te sorprenda el día que la encuentres.

¿Sabías que `length` se actualiza solo? No necesitas "avisarle" al array cuando haces `push` o `pop`: `length` siempre refleja la realidad. Y al revés: si asignas un valor menor a `length`, el array se recorta de verdad. Es una propiedad "viva", conectada con el contenido. Pocos lenguajes te dan ese poder directo; úsalo con respeto.

Cuidado con comparar arrays con `===` esperando comparar contenido. `[1, 2] === [1, 2]` es `false` porque son dos objetos distintos en memoria. Si necesitas saber si dos listas tienen lo mismo, compara elemento por elemento con un ciclo, o convierte ambas a texto con `join` y compara los textos. Este detalle aparece en exámenes y entrevistas con frecuencia sospechosa.

Una anécdota final de errores típicos: un estudiante tenía un carrito de compras donde cada vez que agregaba un producto, el total se calculaba mal solo a veces. El bug: su función de agregar hacía push sobre el array que también usaba la función de mostrar, y la función de mostrar ordenaba el array "para verse bonito", mutando el orden original. Como el total se calculaba después con índices asumidos, a veces coincidía y a veces no. La solución fue que la función de mostrar trabajara sobre [...carrito] en vez del original. Moraleja: cuando un bug aparece "a veces sí y a veces no", sospecha de una mutación compartida. Los bugs intermitentes casi siempre son referencias compartidas haciendo travesuras.

Mini-reto de cierre: escribe una función flecha llamada estadísticas que reciba un array de números y devuelva un objeto con el total, el promedio, el mayor y el menor. Necesitarás acumuladores, comparaciones dentro del ciclo y el truco de los paréntesis para devolver el objeto. Si lo logras sin mirar atrás, esta unidad ya es tuya por completo.

Resumen de la unidad

Las funciones flecha son la sintaxis compacta para funciones cortas: $(a, b) \Rightarrow a + b$. Nacen para eliminar ceremonia en funciones de una o dos líneas y dominan el código JavaScript moderno.

Con un solo parámetro puedes quitar los paréntesis: $x \Rightarrow x * 2$. Con cero o con dos o más parámetros, los paréntesis son obligatorios.

El return implícito funciona solo sin llaves: $n \Rightarrow n * 2$ devuelve el resultado automáticamente. En cuanto pones llaves, debes escribir return o la función devuelve undefined.

Para devolver un objeto literal con return implícito, envuélvelo en paréntesis: $() \Rightarrow (\{\text{nombre: "Ana"}\})$. Sin los paréntesis, JavaScript confunde las llaves del objeto con el cuerpo de la función.

Usa flechas para funciones cortas, transformaciones y callbacks. No las uses para métodos de objetos que necesiten this tradicional, para funciones largas y complejas, ni como constructoras con new.

Las flechas heredan el this del lugar donde se crean en vez de tener uno propio. Eso es una ventaja en callbacks y un problema en métodos de objetos; en la unidad 6 lo verás a fondo.

Los valores por defecto en parámetros hacen funciones amables: $(\text{nombre} = \text{"visitante"}) \Rightarrow \dots$. El parámetro rest (...args) junta argumentos variables en un array.

Un array es una lista ordenada creada con corchetes: `const frutas = ["manzana", "pera"]`. Los índices empiezan en cero por herencia de los lenguajes de bajo nivel; el último índice válido siempre es `length - 1`.

Accedes con corchetes y el índice: `frutas[0]`. Modificas igual: `frutas[1] = "uva"`. Acceder a un índice inexistente devuelve `undefined` sin lanzar error, así que verifica tus índices.

`length` es una propiedad sin paréntesis que dice cuántos elementos hay. Es la base para recorrer: `for (let i = 0; i < arr.length; i++)`.

`push` agrega al final y devuelve la nueva longitud; `pop` quita el último y lo devuelve. `shift` quita el primero y `unshift` agrega al inicio. `push` y `pop` son la pareja rápida; `shift` y `unshift` reindexan todo.

Con `push` y `pop` implementas una pila: el último en entrar es el primero en salir. Con `push` y `shift` implementas una cola: el primero en entrar es el primero en salir.

Los patrones para recorrer son cinco: hacer algo con cada elemento, transformar a un array nuevo, filtrar con condición, acumular en una variable y buscar con bandera más `break`.

El acumulador nace antes del ciclo, se actualiza dentro y se usa después. Para sumas empieza en 0; para multiplicaciones empieza en 1.

Los arrays pueden mezclar tipos pero usualmente no deberían: si cada posición significa algo distinto, un objeto es mejor herramienta que un array mixto.

Asignar un array a otra variable no lo copia: crea una referencia al mismo array. Para copiar de verdad usa `[...original]` o `slice()`. Con primitivos sí hay copia real.

Las funciones que reciben arrays obtienen la referencia: si hacen `push` o `pop`, el original cambia. Diseña tus funciones para que sea obvio cuándo modifican y cuándo solo leen.

`slice` copia sin modificar; `concat` une sin modificar; `[...a,...b]` une creando un array nuevo. Son tus herramientas seguras cuando el original debe quedar intacto.

Cuidado con modificar un array mientras lo recorres: los índices se mueven y el ciclo se vuelve impredecible. Trabaja sobre una copia o construye un array nuevo.

Las flechas y los arrays son la pareja del JavaScript moderno: funciones pequeñas aplicadas a listas de datos. Dominarlos por separado te prepara para combinarlos con `map`, `filter` y `forEach` en la próxima unidad.

UNIDAD 6: Recorrer datos y objetos

En la unidad anterior aprendiste las dos herramientas básicas del programador moderno de JavaScript: las funciones flecha, compactas y elegantes, y los arrays, las listas ordenadas donde viven los datos. Ahora viene la parte donde esas dos herramientas se combinan y se vuelven un superpoder. Esta unidad trata de recorrer datos y de modelar el mundo con objetos, y te lo digo sin exagerar: lo que vas a aprender aquí es lo que separa a quien "sabe un poco de JavaScript" de quien puede construir aplicaciones de verdad.

Piénsalo así. Hasta ahora tus programas manejan datos sueltos: un número aquí, un texto allá, una lista de calificaciones. Pero el mundo real no está hecho de datos sueltos. Una tienda no vende "un precio": vende productos que tienen nombre, precio, existencias, categoría y foto. Una red social no maneja "un texto": maneja publicaciones con autor, fecha, contenido, likes y comentarios. Cada cosa del mundo real es un paquete de datos relacionados, y en JavaScript esos paquetes se llaman objetos. Y las listas del mundo real no son listas de números sueltos: son listas de objetos, como el catálogo de productos o la lista de usuarios. Para trabajar con esas listas necesitas formas elegantes de recorrerlas, transformarlas y filtrarlas. Ahí entran `forEach`, `map` y `filter`, los tres métodos que, junto con las funciones flecha de la unidad anterior, forman el pan de cada día del desarrollador profesional.

En esta unidad vas a aprender primero la tríada de recorrido: `forEach` para hacer algo con cada elemento, `map` para transformar cada elemento y producir un array nuevo, y `filter` para quedarte solo con los elementos que cumplen una condición. Vas a entender conceptualmente la diferencia entre ellos: cuándo el método transforma y cuándo no, qué devuelve cada uno, y por qué elegir uno u otro no es cuestión de gusto sino de intención. Después vamos a entrar a los objetos literales: qué son, cómo se crean con llaves, cómo se definen propiedades y valores, cómo se accede con punto y con corchetes, cómo se agregan y eliminan propiedades. Luego veremos los métodos de objeto, que son funciones guardadas dentro de objetos, y la famosa palabra `this`: qué es, por qué existe y cómo se comporta. Cerraremos con ejemplos de objetos que modelan cosas reales, porque un objeto cobra sentido cuando representa algo que reconoces: un producto, un usuario, un coche, una cuenta de banco.

¿Por qué importa todo esto? Porque a partir de aquí, tus programas van a parecerse a programas de verdad. Cuando consumes datos de una API en la unidad de fetch, esos datos van a llegar como arrays de objetos. Cuando construyas interfaces, vas a recorrer listas de objetos para mostrarlos en pantalla. Cuando guardes información, la vas a organizar en objetos. Los objetos son la forma en que JavaScript modela la realidad, y `forEach`, `map` y `filter` son la forma en que la recorres. Sin estos temas, te quedas en el nivel de los ejercicios de juguete; con ellos, puedes empezar a construir proyectos que la gente usa.

El panorama de la unidad: en la explicación detallada desmenuzaremos `forEach`, `map` y `filter` uno por uno, con sus firmas, sus valores de retorno y sus casos de uso, y los compararemos entre sí para que nunca los confundas. Luego construiremos los objetos desde cero: sintaxis literal, propiedades, acceso por punto y por corchetes con sus reglas, agregar y eliminar propiedades, y objetos anidados. Después atacaremos los métodos y `this`, el tema que más dolores de cabeza causa a los principiantes pero que, bien explicado, se vuelve intuitivo. Veremos por qué `this` existe, cómo cambia según quién llama al método, y la trampa clásica de usar funciones flecha como métodos. En los ejemplos paso a paso construiremos código real: desde recorrer una lista de precios con cada método hasta modelar una tienda completa con productos que tienen métodos. La práctica te traerá ejercicios graduados con soluciones explicadas, y el resumen cerrará con los puntos clave.

Un consejo de colega: esta unidad tiene dos mitades que se alimentan. Si sientes que los métodos de recorrido te cuestan, practícalos con arrays simples de números antes de combinarlos con objetos. Y si `this` te confunde, no te frustres: confunde a todo el mundo al principio, incluidos programadores con años de experiencia en otros lenguajes. La clave es ver suficientes ejemplos hasta que el patrón "clic" en tu cabeza. Vamos a darle con todo.

Explicación detallada

Empecemos con la tríada: `forEach`, `map` y `filter`. Los tres son métodos de array, los tres reciben una función como argumento, y los tres recorren el array elemento por elemento. La diferencia está en qué hacen con cada elemento y qué devuelven. Entender esa diferencia es el corazón de esta primera mitad.

`forEach` es el más sencillo de entender: ejecuta una función una vez por cada elemento, y no devuelve nada útil, solo `undefined`. Se usa cuando quieres "hacer algo" con cada elemento: imprimirlo, guardarlo en una base de datos, actualizar la pantalla, enviarlo a un servidor. Es el reemplazo moderno del `for` clásico para el patrón de "hacer algo con cada elemento". La sintaxis: `numeros.forEach(n => console.log(n));`. La flecha recibe cada elemento y hace algo

con él. Nota que no hay array nuevo, no hay resultado que guardar: `forEach` es pura acción, como un repartidor que toca cada puerta y se va.

Pero `forEach` tiene una limitación importante que debes conocer: no puedes romperlo a la mitad con `break`. Un `for` clásico te deja salir temprano cuando encuentras lo que buscas; `forEach` no: siempre recorre todos los elementos. Si necesitas salir temprano, usa un `for` clásico o, más adelante en tu camino, métodos como `find` o `some`. Esta limitación no es un defecto, es una decisión de diseño: `forEach` comunica "quiero hacer esto con TODOS", y si no quieres hacerlo con todos, probablemente necesitas otro método. Otra sutileza: `forEach` ignora los huecos en arrays dispersos, esos espacios vacíos que se crean al asignar índices lejanos. Es un caso borde raro, pero existe.

`map` es el transformador: aplica una función a cada elemento y devuelve un array NUEVO con los resultados, sin modificar el original. La sintaxis: `const dobles = numeros.map(n => n * 2);`. Si `numeros` es `[1, 2, 3]`, `dobles` es `[2, 4, 6]` y `numeros` sigue siendo `[1, 2, 3]`. Esta es la diferencia fundamental con `forEach`: `map` produce algo, `forEach` solo actúa. La regla de oro: si necesitas un array nuevo donde cada elemento es la transformación del original, usa `map`. Es el método que más vas a usar en tu carrera, te lo garantizo: transformar listas de datos es la operación más común en el desarrollo real, desde convertir precios de dólares a pesos hasta extraer los nombres de una lista de usuarios.

Un detalle crucial de `map`: el array nuevo siempre tiene la misma longitud que el original. Cada elemento del original produce exactamente un elemento en el resultado, en el mismo orden. Si tu función de transformación devuelve `undefined` para algunos elementos, el array nuevo tendrá `undefined` en esas posiciones; `map` no filtra, no salta, no elimina: transforma uno a uno. Cuando veas un array resultado con huecos de `undefined`, casi seguro alguien usó `map` donde debía usar `filter`, o su función no devuelve lo que cree. Y recuerda la lección de la unidad 5 sobre el `return` implícito: `numeros.map(n => {n * 2;})` produce un array de `undefined`, porque las llaves matan el `return`. El error clásico de `map` es exactamente ese, y ahora ya sabes reconocerlo al instante.

`filter` es el selector: prueba cada elemento con una función que debe devolver `true` o `false`, y devuelve un array NUEVO solo con los elementos que pasaron la prueba, es decir, donde la función devolvió un valor `truthy`. La sintaxis: `const mayores = edades.filter(e => e >= 18);`. Si `edades` es `[15, 22, 17, 30]`, `mayores` es `[22, 30]`. El original no se modifica. La función que le pasas a `filter` es un predicado, una pregunta que se responde sí o no: "¿este elemento cumple la condición?". La regla de oro: si necesitas un subconjunto de la lista, usa `filter`.

La diferencia entre `map` y `filter` confunde al principio porque ambos devuelven arrays nuevos. La forma de no confundirlos: `map` responde "¿en qué convierto cada elemento?" y siempre devuelve la misma cantidad de elementos; `filter` responde "¿cuáles elementos me quedo?" y devuelve una cantidad igual o menor. Un truco mental: `map` cambia la forma de los datos, `filter` cambia la cantidad. Cuando dudes, pregúntate: ¿quiero los mismos elementos pero transformados, o quiero menos elementos pero intactos? La respuesta te dice qué método usar.

¿Sabías que puedes encadenar estos métodos? Como `map` y `filter` devuelven arrays nuevos, puedes llamar otro método sobre el resultado: `usuarios.filter(u => u.activo).map(u => u.nombre)`. Esto se lee como una frase: "de los usuarios, quédate con los activos y dame sus nombres". El encadenamiento es una de las cosas más elegantes de JavaScript y es extremadamente común en código profesional. Cada eslabón de la cadena es una operación clara y el resultado fluye de una a otra. Eso sí, no abuses: una cadena de seis métodos se vuelve difícil de depurar; si la cadena crece mucho, divídela en pasos con nombres descriptivos.

Comparemos código malo y código bueno para que sientas la diferencia. Tarea: de una lista de precios, quedarse con los mayores a 100 y duplicarlos. Código malo a la antigua: un `for` con un `if` adentro, un `push` condicional y luego otro `for` para duplicar, todo con variables temporales. Funciona, pero son quince líneas donde la intención se diluye entre la mecánica. Código bueno: `const resultado = precios.filter(p => p > 100).map(p => p * 2);`. Una línea que se lee como la descripción de la tarea. No es solo estética: el código bueno tiene menos lugares donde equivocarse, porque no manejas índices ni arrays temporales a mano.

Ahora, una advertencia honesta: `forEach`, `map` y `filter` no reemplazan al `for` clásico en todo. El `for` sigue siendo necesario cuando necesitas el índice con control fino, cuando quieres romper el ciclo a la mitad, cuando recorres dos arrays en paralelo, o cuando el rendimiento en un ciclo supercaliente importa. Los métodos son herramientas, no religión. El programador maduro elige según el caso: ¿quiero expresar una transformación? `map`. ¿Quiero control total del recorrido? `for`. Saber cuándo NO usar la herramienta moderna es tan valioso como saber usarla.

Pasemos a los objetos, la segunda gran mitad de la unidad. Un objeto literal es una colección de pares clave-valor entre llaves. La sintaxis: `const persona = {nombre: "Ana", edad: 25, activa: true};`. Cada par tiene una clave, también llamada propiedad, que es un nombre, y un valor, que puede ser de cualquier tipo: texto, número, booleano, array, otro objeto, incluso una función. Piensa en un objeto como una ficha de registro: cada campo tiene su etiqueta y su dato. A diferencia del array, donde accedes por posición numérica, en el objeto accedes por

nombre, y eso hace que el código se explique solo: `persona.edad` se entiende sin comentarios, mientras que `persona[1]` exige que recuerdes qué significa el 1.

Crear objetos es directo, pero hay detalles de sintaxis que debes dominar. Las claves normalmente se escriben sin comillas: `{nombre: "Ana"}`. Si la clave tiene espacios o caracteres especiales, necesita comillas: `{"nombre completo": "Ana Paz"}`. Los valores pueden ser cualquier expresión, incluidas variables: `const edad = 25; const persona = {nombre: "Ana", edad: edad};`. Y hay un atajo elegante: si la variable se llama igual que la clave, puedes escribirla una sola vez: `const persona = {nombre, edad};`. Esto se llama sintaxis abreviada de propiedades y es muy común en código moderno. Parece un detalle menor, pero cuando construyes objetos con diez propiedades, ahorra mucho ruido.

Acceder a propiedades tiene dos formas: punto y corchetes. `persona.nombre` usa punto: es la forma habitual, corta y legible. `persona["nombre"]` usa corchetes con un texto: hace lo mismo pero permite claves dinámicas. ¿Cuándo necesitas corchetes? Cuando la clave está en una variable: `const campo = "nombre"; persona[campo]` vale "Ana". Con punto no puedes hacer eso: `persona.campo` buscaría una propiedad literalmente llamada "campo". También necesitas corchetes para claves con espacios o que empiezan con número. La regla práctica: usa punto siempre que puedas, porque se lee mejor; usa corchetes cuando la clave es dinámica o "rara". Un error clásico es usar punto con una variable y obtener `undefined` sin entender por qué; si ves `undefined` al acceder a una propiedad que "debería existir", revisa si necesitabas corchetes.

Agregar propiedades a un objeto existente es tan simple como asignar: `persona.ciudad = "México";`. Si la propiedad no existía, se crea; si existía, se actualiza. No hay método especial, no hay ceremonia: la asignación crea. Esto hace a los objetos muy flexibles, pero también peligrosos: un error de dedo al escribir el nombre crea una propiedad nueva en vez de actualizar la existente, y el bug es silencioso. `persona.edad = 26;` actualiza; `persona.edda = 26;` crea una propiedad basura "edda" y la edad real sigue igual. Los editores modernos ayudan con autocompletado, pero el hábito de revisar los nombres te va a salvar más de una vez.

Eliminar propiedades se hace con `delete`: `delete persona.activa;`. Después de eso, `persona.activa` es `undefined`. Ojo: `delete` solo quita la propiedad del objeto, no hace nada más; y en código moderno se usa poco, porque muchas veces es mejor marcar algo como inactivo que borrarlo. Pero debes conocerlo porque aparece en código ajeno y en entrevistas. Una sutileza: `delete` devuelve `true` si tuvo éxito, aunque casi nadie usa ese valor de retorno.

Los objetos pueden anidarse: un objeto dentro de otro, un array dentro de un objeto, un objeto dentro de un array. Ejemplo: `const pedido = {cliente: "Ana", productos: ["pan", "leche"], direccion: {calle: "Reforma", numero: 222}}`; Para llegar al número de la calle escribes `pedido.direccion.numero`: navegas por puntos como quien abre carpetas. Para el segundo producto: `pedido.productos[1]`. Combinar punto y corchetes según corresponda es una habilidad que se vuelve automática con práctica. Los datos reales siempre vienen anidados: la respuesta de una API es un objeto con arrays de objetos que tienen más objetos dentro. Acostúmbrate a leer estas estructuras de afuera hacia adentro.

¿Sabías que en JavaScript casi todo es un objeto? Los arrays son objetos con superpoderes para listas, las funciones son objetos que se pueden llamar, y hasta los textos y números tienen métodos porque JavaScript los envuelve temporalmente en objetos cuando los necesitas. Esta es la razón profunda de que `"hola".length` funcione aunque los textos sean primitivos. No necesitas memorizar la mecánica interna ahora, pero sí la idea: el objeto es el ciudadano de primera clase del lenguaje, y entenderlo a fondo es entender JavaScript.

Ahora el plato fuerte: los métodos y `this`. Un método es simplemente una función guardada como propiedad de un objeto. En vez de tener una función suelta que recibe los datos, el objeto lleva sus propias funciones que operan sobre sus datos. Ejemplo: `const perro = {nombre: "Firulais", ladrar: function() {return this.nombre + "dice guau";}}`; Aquí `ladrar` es un método. Cuando lo llamas como `perro.ladrar()`, dentro de la función la palabra `this` se refiere a `perro`, así que `this.nombre` es `"Firulais"`. El método produce `"Firulais dice guau"`.

¿Por qué existe `this`? Imagina que tienes cien objetos `perro` y una sola función `ladrar` suelta. La función necesita saber de QUÉ `perro` hablan: ¿el nombre de cuál uso? Podrías pasarle el `perro` como parámetro: `ladrar(perro1)`, `ladrar(perro2)`. Funciona, pero es torpe. La solución elegante: la función vive dentro del objeto y usa `this` para referirse "al objeto dueño", al que la llamó. `this` es como la palabra "yo" en el lenguaje: "yo" significa algo distinto según quién lo diga, pero siempre se refiere al hablante. Cuando `perro.ladrar()` se ejecuta, `this` es `perro`; si tuvieras `gato.maullar()` con `this.nombre` adentro, `this` sería `gato`. El valor de `this` se decide en el momento de la llamada según quién está antes del punto. Esa es la regla de oro número uno de `this`: mira qué está a la izquierda del punto en la llamada, ese es `this`.

La sintaxis moderna para métodos es aún más corta: puedes escribir `ladrar() {return this.nombre + "dice guau";}` sin la palabra `function` ni los dos puntos. Es azúcar sintáctico para lo mismo y es lo que verás en código actual. Úsala con confianza: `const perro = {nombre: "Firulais", ladrar() {return this.nombre + "dice guau";}}`;

Ahora la trampa clásica, la que ha hecho llorar a más principiantes: usar una función flecha como método. `const perro = {nombre: "Firulais", ladrar: () => this.nombre + "dice guau"}; . ¿Qué pasa al llamar perro.ladrar()? Las flechas NO tienen su propio this: heredan el this del lugar donde fueron creadas. Como el objeto literal no crea un ámbito nuevo para this, la flecha hereda el this de afuera, que normalmente no tiene nombre, así que obtienes undefined dice guau o un error. La regla de oro número dos: para métodos que usen this, usa funciones normales o la sintaxis corta de método, NUNCA flechas. Esta regla es innegociable y te va a ahorrar horas de depuración. En la unidad 5 te advertí de esto; ahora ves el porqué con tus propios ojos.`

Pero ojo, que las flechas sí tienen un rol estelar DENTRO de los métodos. Imagina un método que usa `forEach` o `setTimeout` con un callback y dentro del callback quiere usar `this`. Con una función normal como callback, el `this` se pierde porque el callback no se llama como método del objeto. Con una flecha como callback, el `this` se conserva porque la flecha hereda el `this` del método. Ejemplo: `const equipo = {nombre: "Tigres", jugadores: ["Ana", "Luis"], presentar() { this.jugadores.forEach(j => console.log(j + "juega en" + this.nombre));}}; . La flecha dentro de forEach hereda el this de presentar, que es equipo. Este patrón, método normal afuera y flecha adentro, es extremadamente común en código profesional. Si lo entiendes, entiendes el ochenta por ciento de los usos reales de this.`

Hay más formas de controlar `this`, como `call`, `apply` y `bind`, que te permiten invocar una función diciéndole explícitamente qué será `this`. Son herramientas avanzadas que existen para casos especiales, como pedir prestado un método de otro objeto. No las necesitas dominar ahora, pero debes saber que existen para que no te asusten cuando las veas: `funcion.bind(objeto)` crea una versión de la función con `this` fijado a ese objeto. Las menciono para que tu mapa del territorio esté completo.

Veamos objetos que modelan cosas reales, porque ahí es donde todo cobra sentido. Un producto de tienda: `{nombre: "Laptop", precio: 15000, existencias: 5, aplicarDescuento(porcentaje) {this.precio = this.precio * (1 - porcentaje / 100);}}`. El método modifica el propio objeto usando `this`. Una cuenta de banco: `{titular: "Ana", saldo: 1000, depositar(cantidad) {this.saldo += cantidad;}, retirar(cantidad) {if (cantidad <= this.saldo) { this.saldo -= cantidad; return true;} return false;}}`. Nota cómo retirar valida antes de modificar: los métodos son el lugar ideal para poner las reglas del negocio, porque los datos y sus reglas viajan juntos. Un usuario con lista de tareas: `{nombre: "Luis", tareas: [], agregar(tarea) { this.tareas.push(tarea);}, pendientes() {return this.tareas.filter(t => !t.hecha).length;}}`. Aquí un método usa `filter` sobre un array interno: la unidad 5 y la 6 trabajando juntas en un objeto real.

La gran lección de diseño: los objetos agrupan datos con las operaciones que les corresponden. En vez de tener funciones sueltas que reciben arrays y hacen magia, el objeto ofrece métodos con nombres claros que encapsulan la lógica. Esto se llama encapsulación y es la base de la programación orientada a objetos. No necesitas clases ni herencia para aprovecharla: con objetos literales y métodos ya estás diseñando como profesional.

Preguntas frecuentes para cerrar. ¿Un objeto puede tener un array de objetos? Sí, y es la estructura más común del mundo real: `const catalogo = [{nombre: "Pan", precio: 20}, {nombre: "Leche", precio: 25}];`. Para obtener los nombres: `catalogo.map(p => p.nombre)`. Para los baratos: `catalogo.filter(p => p.precio < 22)`. ¿Ves cómo todo se conecta? ¿Las claves de un objeto tienen orden? En la práctica, las claves de texto mantienen el orden de inserción, pero no debes depender de eso para lógica importante; si el orden importa, usa un array. ¿Cómo sé si una propiedad existe? Con el operador `in`: `"nombre" in persona` devuelve `true` o `false`; o comparando `persona.nombre !== undefined`, aunque esto falla si la propiedad existe pero vale `undefined` a propósito. Para tu nivel, `in` es claro y suficiente.

¿Sabías que puedes copiar objetos con el operador de propagación igual que con arrays? `const copia = {...original};` crea un objeto nuevo con las mismas propiedades. Y tiene la misma advertencia de copia superficial: los objetos anidados se comparten. La simetría entre `[...array]` y `{...objeto}` no es casualidad: es el lenguaje diciéndote que aprendiste un patrón general, no dos trucos aislados.

Cuidado con acceder a propiedades de `undefined` o `null`: si haces `usuario.direccion.calle` pero `usuario.direccion` es `undefined`, el programa truena con un error de `"cannot read properties of undefined"`. Es uno de los errores más comunes en aplicaciones reales, porque los datos de APIs a veces vienen incompletos. La defensa moderna es el encadenamiento opcional: `usuario.direccion?.calle` devuelve `undefined` en vez de tronar si `direccion` no existe. Es un tema un poco avanzado para esta unidad, pero te lo dejo sembrado porque lo vas a necesitar muy pronto.

Para terminar la explicación, mira el arco completo de lo que llevas: empezaste con valores sueltos, aprendiste a listarlos en arrays, a recorrerlos con ciclos, a escribir funciones compactas con flechas, y ahora sabes transformar listas con `map` y `filter`, ejecutar acciones con `forEach`, y modelar entidades con objetos que tienen métodos y `this`. Con estas piezas puedes representar un catálogo de productos, filtrar los disponibles, transformar sus precios y mostrarlos, todo con código claro y moderno. Eso ya no es un ejercicio: es el esqueleto de una aplicación real. La próxima unidad te dará el DOM para mostrarlo en pantalla, y entonces el círculo se cierra.

Ejemplos paso a paso

Igual que en la unidad anterior: escribe cada ejemplo y ejecútalo. Esta vez los ejemplos combinan los métodos de recorrido con objetos, así que verás las dos mitades de la unidad trabajando juntas.

Ejemplo 1: `forEach` para saludar a cada elemento.

```
const nombres = ["Ana", "Luis", "Marta"];

nombres.forEach(nombre => {
  console.log("Hola, " + nombre);
});
```

`forEach` llama a la flecha una vez por cada nombre. La flecha recibe el elemento actual y lo saluda. No hay valor de retorno que guardar: el propósito es la acción, imprimir. Si intentaras `const x = nombres.forEach(...)`, `x` sería `undefined`. Úsalo para efectos: mostrar, enviar, guardar.

Ejemplo 2: `forEach` con índice, el segundo parámetro del callback.

```
const tareas = ["lavar", "cocinar", "barrer"];

tareas.forEach((tarea, indice) => {
  console.log((indice + 1) + ". " + tarea);
});
```

El callback de `forEach` puede recibir hasta tres parámetros: el elemento, su índice y el array completo. Aquí usamos los dos primeros para numerar desde 1. Este patrón reemplaza al for clásico cuando solo quieres mostrar cada elemento con su número.

Ejemplo 3: `map` para transformar, el pan de cada día.

```
const precios = [100, 200, 300];
const conIVA = precios.map(precio => precio * 1.16);

console.log(precios);
console.log(conIVA);
```

Cada precio se multiplica por 1.16 y el resultado va al array nuevo. El original queda intacto: `[100, 200, 300]` contra `[116, 232, 348]`. Memoriza esta forma: `map` + flecha de una línea + `return` implícito. La vas a escribir miles de veces.

Ejemplo 4: el error clásico de map, para reconocerlo al instante.

```
const numeros = [1, 2, 3];

const mal = numeros.map(n => {
  n * 2;
});

const bien = numeros.map(n => n * 2);

console.log(mal);
console.log(bien);
```

mal es [undefined, undefined, undefined] porque las llaves desactivan el return implícito. bien es [2, 4, 6]. Cuando un map te devuelva un array de undefined, ya sabes exactamente dónde mirar: las llaves del callback.

Ejemplo 5: map para extraer una propiedad de objetos.

```
const usuarios = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "Marta", edad: 22 }
];

const nombres = usuarios.map(u => u.nombre);
console.log(nombres);
```

De cada objeto extraemos solo el nombre con u.nombre. El resultado es ["Ana", "Luis", "Marta"]. Este es quizás el uso más común de map en aplicaciones reales: tienes una lista de objetos completa y necesitas solo un campo de cada uno, por ejemplo para llenar un menú desplegable.

Ejemplo 6: filter para seleccionar.

```
const edades = [15, 22, 17, 30, 12, 25];
const mayores = edades.filter(edad => edad >= 18);

console.log(mayores);
```

El predicado `edad => edad >= 18` pregunta por cada elemento "¿eres mayor de edad?". Solo los que responden `true` entran al array nuevo: `[22, 30, 25]`. El original no cambia. Nota que `filter` no transforma: los elementos que pasan quedan intactos.

Ejemplo 7: `filter` con objetos, el caso real.

```
const productos = [
  { nombre: "Pan", precio: 20, disponible: true },
  { nombre: "Leche", precio: 25, disponible: false },
  { nombre: "Queso", precio: 80, disponible: true }
];

const disponibles = productos.filter(p => p.disponible);
console.log(disponibles);
```

Nos quedamos solo con los productos disponibles. El predicado accede a la propiedad booleana de cada objeto. En una tienda real, así se construye la lista de "productos a la venta" a partir del catálogo completo.

Ejemplo 8: encadenar `filter` y `map`, la frase elegante.

```
const productos = [
  { nombre: "Pan", precio: 20 },
  { nombre: "Leche", precio: 25 },
  { nombre: "Queso", precio: 80 }
];

const nombresCaros = productos
  .filter(p => p.precio > 22)
  .map(p => p.nombre);

console.log(nombresCaros);
```

Primero `filter` deja `[Leche, Queso]`, luego `map` extrae los nombres: `["Leche", "Queso"]`. Se lee como la petición: "dame los nombres de los productos que cuestan más de 22". El encadenamiento con saltos de línea y un punto al inicio de cada línea es el estilo estándar para que se lea bien.

Ejemplo 9: crear un objeto y acceder por punto y corchetes.

```
const alumno = {
  nombre: "Ana",
  edad: 20,
  carrera: "Sistemas"
};

console.log(alumno.nombre);
console.log(alumno["edad"]);

const campo = "carrera";
console.log(alumno[campo]);
```

Punto para lo directo, corchetes con texto para lo mismo, corchetes con variable para lo dinámico. Las tres líneas imprimen "Ana", 20 y "Sistemas". Practica la tercera forma hasta que salga natural: es la que usarás cuando el nombre del campo venga de un formulario o de una respuesta de servidor.

Ejemplo 10: agregar, actualizar y eliminar propiedades.

```
const coche = { marca: "Nissan", modelo: "Versa" };

coche.anio = 2022;
console.log(coche);

coche.modelo = "Sentra";
console.log(coche);

delete coche.anio;
console.log(coche);
```

La asignación crea la propiedad anio, luego actualiza modelo de "Versa" a "Sentra", y delete elimina anio. Los console.log muestran cómo el objeto cambia de forma en cada paso. Los objetos en JavaScript son flexibles: crecen y se encogen según los necesitas.

Ejemplo 11: objetos anidados y cómo navegarlos.

```
const pedido = {
  cliente: "Luis",
  direccion: {
    calle: "Reforma",
    numero: 222,
    ciudad: "CDMX"
  },
  productos: ["pan", "leche"]
};

console.log(pedido.direccion.ciudad);
console.log(pedido.productos[0]);
console.log(pedido.productos.length);
```

Navegamos con puntos por los objetos anidados y con corchetes por el array interno. `pedido.direccion.ciudad` es "CDMX", `pedido.productos[0]` es "pan". Lee estas expresiones de izquierda a derecha como una ruta: pedido, dentro su dirección, dentro la ciudad.

Ejemplo 12: método con `this`, el "yo" del objeto.

```
const perro = {
  nombre: "Firulais",
  ladrar() {
    return this.nombre + " dice guau";
  }
};

console.log(perro.ladrar());
```

Al llamar `perro.ladrar()`, `this` es `perro`, así que `this.nombre` es "Firulais" y el método devuelve "Firulais dice guau". Si crearas otro perro con otro nombre y el mismo método, `this` se adaptaría solo. Esa es la magia: el método es genérico, `this` lo personaliza.

Ejemplo 13: la trampa de la flecha como método, para verla fallar a propósito.

```
const gato = {
  nombre: "Michu",
  maullar: () => this.nombre + " dice miau"
};

console.log(gato.maullar());
```

Esto imprime "undefined dice miau" o truena, porque la flecha no tiene this propio y hereda el de afuera, donde no hay nombre. La lección grabada a fuego: métodos que usan this se escriben con sintaxis de método o function, nunca con flecha. Guarda este ejemplo como tu detector de humo.

Ejemplo 14: flecha DENTRO de un método, el patrón profesional.

```
const equipo = {
  nombre: "Tigres",
  jugadores: ["Ana", "Luis", "Marta"],
  presentar() {
    this.jugadores.forEach(jugador => {
      console.log(jugador + " juega en " + this.nombre);
    });
  }
};

equipo.presentar();
```

El método presentar es sintaxis normal, así que this es equipo. La flecha dentro de forEach hereda ese this, por eso this.nombre sigue siendo "Tigres" dentro del callback. Si el callback fuera una función normal, this se perdería. Método normal afuera, flecha adentro: el patrón que verás en todo el código profesional.

Ejemplo 15: objeto que modela una cuenta de banco, con reglas de negocio.

```

const cuenta = {
  titular: "Ana",
  saldo: 1000,
  depositar(cantidad) {
    this.saldo += cantidad;
    return this.saldo;
  },
  retirar(cantidad) {
    if (cantidad <= this.saldo) {
      this.saldo -= cantidad;
      return "Retiro exitoso. Saldo: " + this.saldo;
    }
    return "Fondos insuficientes";
  }
};

console.log(cuenta.depositar(500));
console.log(cuenta.retirar(300));
console.log(cuenta.retirar(5000));

```

depositar suma y devuelve el nuevo saldo: 1500. retirar valida primero: si hay fondos, resta y confirma; si no, rechaza sin modificar nada. El tercer retiro falla con "Fondos insuficientes" y el saldo queda intacto. Los métodos protegen los datos con reglas: nadie puede dejar la cuenta en negativo porque el único camino al saldo pasa por la validación.

Ejemplo 16: programa completo, tienda con catálogo de objetos y métodos de recorrido.


```

const tienda = {
  nombre: "La Esquina",
  productos: [
    { nombre: "Pan", precio: 20, existencias: 10 },
    { nombre: "Leche", precio: 25, existencias: 0 },
    { nombre: "Queso", precio: 80, existencias: 5 }
  ],
  disponibles() {
    return this.productos.filter(p => p.existencias > 0);
  },
  nombresDisponibles() {
    return this.disponibles().map(p => p.nombre);
  },
  valorInventario() {
    let total = 0;
    this.productos.forEach(p => {
      total += p.precio * p.existencias;
    });
    return total;
  }
};

console.log(tienda.nombresDisponibles());
console.log("Valor del inventario: " + tienda.valorInventario());

```

Este ejemplo junta toda la unidad: un objeto con un array de objetos adentro, métodos que usan this, filter para los disponibles, map para sus nombres, forEach con acumulador para el valor total. disponibles devuelve los productos con existencias; nombresDisponibles encadena sobre ese resultado; valorInventario multiplica precio por existencias de cada uno y suma. Es pequeño pero es arquitectura real: datos, métodos que los consultan y métodos que los transforman. Si entiendes este ejemplo línea por línea, la unidad está dominada.

Práctica

Misma dinámica que siempre: intenta primero, sufre un poco, luego mira la solución, entiéndela y reescríbela sin ver. Los ejercicios mezclan recorrido y objetos porque en la vida real nunca vienen separados.

Ejercicio 1: con forEach, ticket de compra. Enunciado: tienes un array de precios. Usa forEach para imprimir cada precio con el formato "Producto N: \$precio", numerando desde 1. No debes crear ningún array nuevo.

```
const precios = [50, 120, 80];

precios.forEach((precio, i) => {
  console.log("Producto " + (i + 1) + ": $" + precio);
});
```

Explicación paso a paso: `forEach` es la elección correcta porque la tarea es una acción, imprimir, no una transformación: no necesitamos ningún resultado. El callback recibe el precio y el índice `i`; sumamos 1 al índice para la numeración humana. La salida muestra los tres productos numerados. Si hubieras usado `map` aquí, funcionaría pero sería semánticamente incorrecto: `map` promete un array nuevo que no necesitas, y crear basura confunde a quien lee tu código. Elegir el método por su intención es parte de escribir buen código.

Ejercicio 2: con `map`, precios con descuento. Enunciado: escribe una función flecha `aplicarDescuento` que reciba un array de precios y un porcentaje, y devuelva un array NUEVO con cada precio ya descontado. El array original no debe modificarse.

```
const aplicarDescuento = (precios, porcentaje) => {
  return precios.map(p => p * (1 - porcentaje / 100));
};

const originales = [100, 200, 300];
const rebajados = aplicarDescuento(originales, 10);

console.log(rebajados);
console.log(originales);
```

Explicación paso a paso: `map` es ideal porque cada precio se transforma en otro precio, uno a uno, y necesitamos el array nuevo. La fórmula $p * (1 - \text{porcentaje} / 100)$ con 10% de descuento equivale a $p * 0.9$. La flecha usa `return` implícito en una línea. La prueba muestra `[90, 180, 270]` en `rebajados` y `[100, 200, 300]` intactos en `originales`, confirmando que no mutamos nada. Este es el patrón de "función pura con `map`": entra una lista, sale una lista nueva, nada cambia afuera.

Ejercicio 3: con `filter`, aprobados. Enunciado: tienes un array de objetos con nombre y calificación. Usa `filter` para quedarte con los aprobados, calificación mayor o igual a 6, y luego `map` para obtener solo sus nombres. Hazlo en una sola cadena.

```
const alumnos = [
  { nombre: "Ana", calificacion: 9 },
  { nombre: "Luis", calificacion: 5 },
  { nombre: "Marta", calificacion: 7 },
  { nombre: "Pedro", calificacion: 4 }
];

const aprobados = alumnos
  .filter(a => a.calificacion >= 6)
  .map(a => a.nombre);

console.log(aprobados);
```

Explicación paso a paso: primero filter evalúa el predicado `a.calificacion >= 6` en cada objeto y conserva solo a Ana y Marta. Sobre ese array intermedio, map extrae `a.nombre` de cada uno. El resultado es `["Ana", "Marta"]`. La cadena se lee como la petición en español: "filtra los de calificación mayor o igual a 6 y dame sus nombres". Nota el orden: filtrar primero reduce el trabajo de map, aunque con listas pequeñas no importa el rendimiento; importa que la intención se lea en orden lógico.

Ejercicio 4: objeto con método que usa this. Enunciado: crea un objeto rectangulo con propiedades base y altura, y un método area que devuelva base por altura usando this. Luego agrega un método perimetro que devuelva el perímetro, también con this. Pruébalos.

```
const rectangulo = {
  base: 10,
  altura: 5,
  area() {
    return this.base * this.altura;
  },
  perimetro() {
    return 2 * (this.base + this.altura);
  }
};

console.log(rectangulo.area());
console.log(rectangulo.perimetro());
```

Explicación paso a paso: usamos la sintaxis corta de método, sin function y sin flecha, para que this funcione. Dentro de area, this es rectangulo porque el método se llama como `rectangulo.area()`; `this.base` es 10 y `this.altura` es 5, así que devuelve 50. En perimetro, `2 * (10`

+ 5) = 30. Si hubiéramos escrito los métodos con flecha, this no sería el rectángulo y obtendríamos NaN. Este ejercicio fija la regla: métodos con this van con sintaxis de método.

Ejercicio 5: acceso dinámico con corchetes. Enunciado: tienes un objeto con datos de un usuario y un array con los nombres de los campos que quieres mostrar. Usa un ciclo for y acceso con corchetes para imprimir cada campo con su valor en formato "campo: valor".

```
const usuario = {
  nombre: "Ana",
  edad: 28,
  ciudad: "Guadalajara"
};

const campos = ["nombre", "ciudad"];

for (let i = 0; i < campos.length; i++) {
  const campo = campos[i];
  console.log(campo + ": " + usuario[campo]);
}
```

Explicación paso a paso: el nombre del campo está en la variable campo, así que el punto no sirve: usuario.campo buscaría una propiedad llamada "campo" que no existe. Con corchetes, usuario[campo] evalúa la variable y accede a usuario["nombre"] y usuario["ciudad"]. La salida muestra "nombre: Ana" y "ciudad: Guadalajara". Este patrón es ubicuo cuando los campos vienen de un formulario, de una configuración o de una API: nunca sabes de antemano qué campo te van a pedir, así que el acceso dinámico es obligatorio.

Ejercicio 6: inventario con métodos encadenados. Enunciado: crea un objeto inventario con un array de productos, cada uno con nombre, precio y existencias. Agrega tres métodos: totalProductos() que devuelva cuántos productos hay; sinExistencias() que devuelva un array con los nombres de los productos agotados; y valorTotal() que devuelva el valor de todo el inventario, precio por existencias sumado. Usa length, filter, map y forEach donde corresponda.

```

const inventario = {
  productos: [
    { nombre: "Pan", precio: 20, existencias: 10 },
    { nombre: "Leche", precio: 25, existencias: 0 },
    { nombre: "Queso", precio: 80, existencias: 3 }
  ],
  totalProductos() {
    return this.productos.length;
  },
  sinExistencias() {
    return this.productos
      .filter(p => p.existencias === 0)
      .map(p => p.nombre);
  },
  valorTotal() {
    let total = 0;
    this.productos.forEach(p => {
      total += p.precio * p.existencias;
    });
    return total;
  }
};

console.log(inventario.totalProductos());
console.log(inventario.sinExistencias());
console.log(inventario.valorTotal());

```

Explicación paso a paso: `totalProductos` usa `length` sobre el array interno: 3. `sinExistencias` filtra los de existencias igual a 0 y mapea a nombres: ["Leche"]. `valorTotal` usa `forEach` con un acumulador: $20 \cdot 10 + 25 \cdot 0 + 80 \cdot 3 = 200 + 0 + 240 = 440$. Cada método elige la herramienta por intención: contar con `length`, seleccionar con `filter`, transformar con `map`, acumular con `forEach`. Los tres usan `this.productos` para llegar al array del propio objeto. Este ejercicio es un mini sistema de inventario real: si lo entiendes completo, sabes modelar datos de negocio.

Ejercicio 7: el reto final, biblioteca personal. Enunciado: crea un objeto biblioteca con un array `libros`, inicialmente vacío, donde cada libro es un objeto con `titulo`, `autor` y `leido` en booleano. Implementa: `agregar(titulo, autor)` que agregue el libro con `leido` en `false`; `marcarLeido(titulo)` que ponga `leido` en `true` si existe; `pendientes()` que devuelva los títulos de los no leídos; y `progreso()` que devuelva el porcentaje de libros leídos, de 0 a 100. Si no hay libros, el progreso es 0.

```

const biblioteca = {
  libros: [],
  agregar(titulo, autor) {
    this.libros.push({ titulo: titulo, autor: autor, leído: false });
    console.log(titulo + " agregado");
  },
  marcarLeido(titulo) {
    for (let i = 0; i < this.libros.length; i++) {
      if (this.libros[i].titulo === titulo) {
        this.libros[i].leído = true;
        console.log(titulo + " marcado como leído");
        return;
      }
    }
    console.log("No se encontró: " + titulo);
  },
  pendientes() {
    return this.libros
      .filter(l => !l.leído)
      .map(l => l.titulo);
  },
  progreso() {
    if (this.libros.length === 0) {
      return 0;
    }
    const leídos = this.libros.filter(l => l.leído).length;
    return (leídos / this.libros.length) * 100;
  }
};

biblioteca.agregar("Cien años de soledad", "García Márquez");
biblioteca.agregar("El principito", "Saint-Exupéry");
biblioteca.marcarLeido("El principito");
console.log(biblioteca.pendientes());
console.log("Progreso: " + biblioteca.progreso() + "%");

```

Explicación paso a paso: agregar hace push de un objeto literal construido con los parámetros; nota la sintaxis abreviada posible, aunque aquí escribimos titulo: titulo explícito para claridad. marcarLeido busca con for y bandera implícita: al encontrar el título, actualiza la propiedad leído del objeto dentro del array, recuerda que los objetos dentro del array se modifican por referencia, y sale con return temprano; si el ciclo termina sin encontrar, avisa. pendientes encadena filter de no leídos con map de títulos. progreso se defiende del array vacío

devolviendo 0 para evitar división entre cero, luego filtra los leídos, cuenta con length y calcula el porcentaje. La prueba agrega dos libros, marca uno como leído, muestra ["Cien años de soledad"] como pendiente y 50% de progreso. Este programa integra objetos, arrays de objetos, métodos, this, filter, map, push, búsqueda y defensa contra casos borde: es todo el curso hasta aquí en un solo objeto. Si lo escribiste por tu cuenta, estás programando como profesional junior sin darte cuenta.

Preguntas frecuentes, curiosidades y profundizaciones

Vamos a cerrar la explicación con una sección de preguntas reales que hacen los estudiantes, curiosidades del lenguaje y profundizaciones que te van a dar ventaja. Léela completa: aquí hay varios "ajá" que te van a ahorrar horas en el futuro.

¿forEach, map o filter? La pregunta del millón, respondida con un método de decisión. Hazte tres preguntas en orden. Uno: ¿necesito un resultado en forma de array nuevo? Si la respuesta es no, porque solo quiero hacer algo con cada elemento, usa forEach. Si la respuesta es sí, pasa a la dos. Dos: ¿el array nuevo tiene la misma cantidad de elementos que el original? Si sí, porque transformo cada uno, usa map. Si no, porque me quedo solo con algunos, usa filter. Tres, la de desempate: ¿estoy cambiando la forma de los datos o la cantidad de datos? Forma es map, cantidad es filter. Con estas tres preguntas no vuelves a confundirlos. Y cuando alguien en una entrevista te pida explicar la diferencia, esta es exactamente la respuesta que demuestra que lo entiendes de verdad y no que lo memorizaste.

¿Sabías que forEach, map y filter reciben más parámetros en el callback de los que normalmente usas? La firma completa es (elemento, indice, array). El tercer parámetro es el array completo que se está recorriendo. Casi nunca lo necesitas, pero existe y alguna vez lo verás en código ajeno: arr.map((elemento, i, arr) =>...). Saber que existe evita que te saque de onda cuando lo encuentres. El índice, en cambio, sí lo usarás seguido: para numerar, para comparar con el vecino, para construir identificadores.

Cuidado con el mito de "los métodos funcionales son lentos". En benchmarks artificiales, un for clásico puede ser marginalmente más rápido que map o filter. En aplicaciones reales, la diferencia es insignificante frente al costo de pintar la pantalla, esperar la red o leer un archivo. La legibilidad gana por goleada: un map bien escrito se entiende en dos segundos, un for manual con push exige leer cinco líneas para entender la intención. Optimiza tu tiempo de lectura, no los nanosegundos del motor. Solo en bucles verdaderamente calientes, como procesar millones de elementos en un videojuego, vale la pena el for manual. Y ese día, ya sabrás suficiente para decidirlo tú mismo.

¿Sabías que `map` y `filter` no modifican el original pero tampoco lo "protegen" por dentro? Si el array contiene objetos, el array nuevo de `map` comparte esos mismos objetos. Ejemplo: `const lista = [{p: 10}]; const copia = lista.map(o => o); copia[0].p = 99; también cambia lista[0].p`. El array es nuevo, los objetos son los mismos. Es la copia superficial otra vez, el fantasma que recorre todo JavaScript. Cuando transformes listas de objetos y necesites independencia total, crea objetos nuevos en el `map`: `lista.map(o => ({...o, p: o.p * 2}))`. Ese patrón, mapear creando objetos nuevos con propagación, es estándar en aplicaciones reales y te lo vas a encontrar en cada proyecto serio.

Hablemos de código malo contra código bueno con objetos, porque aquí también hay niveles. Código malo: `const u = {}; u["nombre"] = "Ana"; u["edad"] = 25;`. Funciona, pero construye el objeto a cucharadas cuando pudo nacer completo. Código bueno: `const u = {nombre: "Ana", edad: 25};`. El objeto literal expresa "esto es una cosa con estos datos" de un vistazo. Otro caso. Código malo: funciones sueltas que reciben diez parámetros: `registrarVenta(nombre, precio, cantidad, cliente, fecha, descuento)`. Cada llamada es un acertijo de orden. Código bueno: `registrarVenta({nombre, precio, cantidad, cliente, fecha, descuento})`. Pasar un objeto como parámetro nombra cada dato y hace las llamadas legibles y resistentes a cambios de orden. Este patrón de "objeto como parámetro" es oro en funciones con muchos argumentos.

Una anécdota de error típico con `this` que veo cada semestre: un estudiante crea un objeto calculadora con métodos que usan `this`, todo funciona perfecto. Luego, para "ahorrar líneas", guarda una referencia al método: `const sumar = calc.sumar;` y la llama como `sumar(2, 3)`. De pronto `this` es `undefined` y todo truena. ¿Por qué? Porque `this` se decide por cómo se LLAMA la función, no por dónde se definió. Al llamarla como `sumar()`, no hay objeto a la izquierda del punto, así que `this` ya no es `calc`. La regla de oro número uno cobra su cuota: el `this` lo pone la llamada. La lección: los métodos deben llamarse como métodos, `objeto.metodo()`, o `this` se pierde. Existen técnicas para "amarrar" el `this`, como `bind`, pero la solución simple es no despegar el método de su objeto.

¿Sabías que puedes listar las claves de un objeto? `Object.keys(persona)` devuelve un array con los nombres de las propiedades: `["nombre", "edad", "ciudad"]`. `Object.values(persona)` devuelve los valores: `["Ana", 28, "Guadalajara"]`. Y `Object.entries(persona)` devuelve pares: `[["nombre", "Ana"], ...]`. Estas tres funciones convierten un objeto en arrays, y una vez que tienes arrays, puedes usar todo lo de esta unidad: `Object.keys(config).forEach(k => console.log(k))`. Es el puente entre los dos mundos y es utilísimo para depurar, para validar formularios y para construir interfaces dinámicas.

Cuidado con usar objetos como "diccionarios" con claves que el usuario controla. Si permites que alguien guarde datos con cualquier texto como clave, puede pisar propiedades heredadas

como constructor o toString y causar comportamientos rarísimos. Para diccionarios con claves arbitrarias existe la estructura Map, más segura. No necesitas dominarla ahora, pero cuando construyas algo donde las claves vienen de usuarios, recuerda que el objeto literal tiene sus límites. Es un detalle de seguridad real que distingue al programador cuidadoso.

Profundicemos en la sintaxis abreviada de propiedades porque la verás en todas partes. Si tienes `const nombre = "Ana"; const edad = 25;`, escribir `const persona = {nombre: nombre, edad: edad};` es redundante: JavaScript permite `const persona = {nombre, edad};` y entiende que la clave toma el nombre de la variable y el valor su contenido. Parece un ahorro menor, pero en funciones que construyen objetos con muchas variables, elimina la mitad del ruido. Y hay un primo de este atajo: los nombres de propiedades computados. Si la clave está en una variable, puedes crearla directo en el literal: `const campo = "email"; const u = {[campo]: "ana@mail.com"};`. Los corchetes dentro del literal evalúan la variable. Es avanzado pero elegante, y aparece en código que construye objetos dinámicamente.

¿Sabías que los métodos también pueden llamarse entre sí usando this? Un objeto puede tener un método `resumen()` que llama a `this.area()` y a `this.perimetro()`. Esto permite construir métodos de alto nivel sobre métodos de bajo nivel, como piezas de Lego. Ejemplo: `const rect = {base: 10, altura: 5, area() {return this.base * this.altura;}, descripcion() {return "Rectángulo de área"+ this.area();}};`. `descripcion` usa `this.area()` sin repetir la fórmula. Si mañana cambia cómo se calcula el área, solo se toca un lugar. Este es el principio de no repetirse, y los objetos con métodos que se llaman entre sí son su mejor vehículo.

Cuidado con this en callbacks dentro de métodos cuando el callback es función normal. Es el error hermano del de la flecha como método, pero al revés: `const equipo = {nombre: "Tigres", jugadores: ["Ana"], presentar() {this.jugadores.forEach(function(j) {console.log(j + "de"+ this.nombre);});}};`. Aquí el callback es función normal, así que su `this` NO es `equipo`: es `undefined` o el objeto global, y `this.nombre` falla. La solución es la flecha, que hereda el `this` del método. Grábate la pareja completa: método normal afuera para tener `this`, flecha adentro para conservarlo. Son dos reglas que juntas cubren el 95% de los casos reales.

Una curiosidad histórica: `this` es una de las partes más criticadas de JavaScript, y varios lenguajes modernos lo eliminaron o lo simplificaron. Pero en JavaScript llegó para quedarse, y cuando lo entiendes, deja de ser un enemigo y se vuelve una herramienta: permite que un solo método sirva para miles de objetos, cada uno con sus datos. Sin `this`, cada objeto necesitaría su propia copia de cada función, un desperdicio enorme. `this` es, en el fondo, un mecanismo de ahorro y reutilización. Entender el porqué del diseño hace que las reglas dejen de sentirse arbitrarias.

Mini-reto uno: escribe una función flecha llamada soloNombres que reciba un array de objetos usuario, cada uno con nombre y activo, y devuelva los nombres de los activos en mayúsculas, todo en una sola cadena de métodos. Pista: filter, map y toUpperCase. Si lo haces en una línea legible, dominas el encadenamiento.

Mini-reto dos: crea un objeto termometro con propiedad celsius y tres métodos: aFahrenheit() que convierta usando this.celsius, aKelvin() que sume 273.15, y descripcion() que devuelva un texto como "25°C son 77°F y 298.15K" llamando a los otros dos métodos con this. Es el ejercicio perfecto para practicar métodos que se llaman entre sí.

Mini-reto tres, el más sabroso: tienes const ventas = [{producto: "pan", total: 200}, {producto: "leche", total: 150}, {producto: "pan", total: 300}]. Escribe código que produzca un objeto resumen donde cada clave sea un producto y el valor el total acumulado: {pan: 500, leche: 150}. Necesitarás forEach, acceso dinámico con corchetes y el patrón de acumulador, pero ahora el acumulador es un objeto. Este ejercicio combina todo lo de la unidad en un problema típico de reportes reales.

Para cerrar, una reflexión de oficio. Los temas de esta unidad, recorrer con métodos funcionales y modelar con objetos, son el vocabulario con el que se escriben las aplicaciones modernas. Cuando leas código de React, de Node, de cualquier librería actual, verás map para pintar listas, filter para buscar, objetos para el estado y métodos con this o flechas bien colocadas. Ya no es "código avanzado": es el día a día. La diferencia entre el programador que se atora leyendo ese código y el que lo lee como una novela es exactamente esta unidad. Tómame un momento para apreciar lo lejos que llegaste: hace unas unidades sumabas dos números, hoy modelas una tienda completa.

Casos de estudio: dos programas completos

La mejor forma de fijar una unidad es ver sus piezas trabajando juntas en programas que parecen reales. Aquí van dos casos de estudio completos, explicados paso a paso. Léelos como si fueran la solución de un colega y trata de predecir cada línea antes de leer la explicación.

Caso de estudio 1: sistema de calificaciones. Una maestra tiene la lista de sus alumnos con calificaciones y necesita: el promedio del grupo, la lista de aprobados, la lista de reprobados con un mensaje de apoyo, y el cuadro de honor con los tres mejores promedios. Todo con lo aprendido en esta unidad.

```

const alumnos = [
  { nombre: "Ana", calificacion: 9 },
  { nombre: "Luis", calificacion: 5 },
  { nombre: "Marta", calificacion: 10 },
  { nombre: "Pedro", calificacion: 4 },
  { nombre: "Sofía", calificacion: 8 },
  { nombre: "Diego", calificacion: 6 }
];

const promedioGrupo = alumnos => {
  let suma = 0;
  alumnos.forEach(a => {
    suma += a.calificacion;
  });
  return suma / alumnos.length;
};

const aprobados = alumnos.filter(a => a.calificacion >= 6);
const reprobados = alumnos.filter(a => a.calificacion < 6);

const mensajesApoyo = reprobados.map(a =>
  a.nombre + ", ¡tú puedes! A repasar y a intentarlo de nuevo."
);

const cuadroHonor = [...alumnos]
  .sort((a, b) => b.calificacion - a.calificacion)
  .slice(0, 3)
  .map(a => a.nombre);

console.log("Promedio del grupo: " + promedioGrupo(alumnos));
console.log("Aprobados: " + aprobados.map(a => a.nombre));
console.log("Mensajes de apoyo:");
mensajesApoyo.forEach(m => console.log("- " + m));
console.log("Cuadro de honor: " + cuadroHonor);

```

Vamos por partes. Los datos son un array de objetos, la estructura estrella de la unidad. `promedioGrupo` es una flecha que usa `forEach` con un acumulador: recorre, suma cada calificación y divide entre la cantidad. Nota que `forEach` es la elección correcta porque solo acumulamos, no necesitamos un array nuevo. `aprobados` y `reprobados` usan `filter` con predicados opuestos: el mismo array original produce dos subconjuntos sin modificarse. `mensajesApoyo` usa `map` para transformar cada reprobado en un mensaje personalizado: de objetos a textos, uno a uno. El cuadro de honor es la parte elegante: `[...alumnos]` copia el array

para no ordenar el original, sort lo ordena de mayor a menor con una flecha comparadora, slice(0, 3) se queda con los tres primeros, y map extrae los nombres. Es una cadena de cuatro eslabones que se lee como la especificación: "copia, ordena descendente, toma tres, dame nombres". Al final, forEach imprime los mensajes porque imprimir es una acción. Este programa cabe en treinta líneas y resuelve un problema administrativo real. Hace dos unidades te habría tomado el triple.

Cuidado con un detalle del cuadro de honor: sort *SÍ* modifica el array sobre el que se llama, por eso la copia con [...alumnos] es obligatoria. Si hubiéramos ordenado alumnos directamente, la lista original quedaría desordenada para el resto del programa y los reportes saldrían en orden raro. Es el mismo fantasma de la mutación compartida de la unidad 5, ahora con sort. Cada vez que uses un método que muta, pregúntate: "¿me importa el orden original después?" Si sí, copia primero.

Caso de estudio 2: carrito de compras con métodos. Una tienda en línea necesita un carrito que agregue productos, quite productos, calcule el total, aplique un cupón de descuento y muestre el ticket. Lo modelamos como un objeto con métodos, usando this en todas partes.

```

const carrito = {
  items: [],
  agregar(producto) {
    const existente = this.items.filter(i => i.nombre === producto.nombre);
    if (existente.length > 0) {
      existente[0].cantidad += producto.cantidad;
    } else {
      this.items.push({ ...producto });
    }
  },
  quitar(nombre) {
    this.items = this.items.filter(i => i.nombre !== nombre);
  },
  total() {
    let suma = 0;
    this.items.forEach(i => {
      suma += i.precio * i.cantidad;
    });
    return suma;
  },
  totalConDescuento(porcentaje) {
    return this.total() * (1 - porcentaje / 100);
  },
  ticket() {
    console.log("=== Ticket ===");
    this.items.forEach(i => {
      console.log(i.cantidad + " x " + i.nombre + " = $" + (i.precio *
i.cantidad));
    });
    console.log("Total: $" + this.total());
  }
};

carrito.agregar({ nombre: "Pan", precio: 20, cantidad: 2 });
carrito.agregar({ nombre: "Leche", precio: 25, cantidad: 1 });
carrito.agregar({ nombre: "Pan", precio: 20, cantidad: 3 });
carrito.ticket();
console.log("Con 10% de descuento: $" + carrito.totalConDescuento(10));
carrito.quitar("Leche");
carrito.ticket();

```

Analicemos las decisiones. agregar primero busca si el producto ya está con filter por nombre: si existe, suma la cantidad al existente en vez de duplicar la línea; si no, hace push de una

copia con {...producto} para no compartir la referencia con el objeto que pasó quien llamó. Ese detalle de la copia evita que cambios externos al producto original contaminen el carrito. quitar reasigna this.items con un filter que excluye el nombre: en vez de buscar y empalmar, construye la lista sin el elemento. Es el estilo funcional: nuevo array en vez de mutación quirúrgica, más fácil de razonar. total usa forEach con acumulador multiplicando precio por cantidad. totalConDescuento llama a this.total(): métodos que usan otros métodos, sin repetir la lógica de suma. ticket combina forEach para imprimir cada línea y this.total() para el cierre. La prueba agrega pan dos veces y el carrito inteligentemente suma cantidades: 5 panes en una sola línea. Luego muestra el ticket, calcula el descuento y quita la leche. Todo el comportamiento del carrito vive dentro del objeto: quien lo usa solo llama métodos con nombres claros. Eso es encapsulación en acción, y es exactamente así como se diseñan los módulos de una aplicación real, solo que en grande.

¿Sabías que este carrito es, en miniatura, la arquitectura de una tienda real? Los datos viven en un estado, los métodos son las únicas operaciones permitidas sobre ese estado, y la interfaz solo llama métodos. Cuando crezca y necesite guardarse en un servidor, los métodos agregar o quitar serán los que hablen con la API. El patrón escala: lo que hoy es un objeto literal mañana es un módulo, pasado mañana un microservicio. Aprenderlo aquí, en pequeño, te prepara para reconocerlo en grande.

Un último cuidado con los métodos que reasignan propiedades con this, como quitar que hace this.items =.... Reasignar la propiedad está bien; lo que no debes hacer es reasignar this mismo, que no se puede. Y recuerda: si despegas un método del objeto para pasarlo como callback, por ejemplo setTimeout(carrito.ticket, 1000), el this se pierde porque la llamada ya no es carrito.ticket(). La solución clásica es envolverlo en una flecha: setTimeout(() => carrito.ticket(), 1000). La flecha conserva la llamada como método y this sobrevive. Este truco aparece en cada aplicación con temporizadores o eventos, así que guárdalo.

Autoevaluación: pregúntate y respóndete

Antes del resumen, un repaso activo. Tapa las respuestas, responde en voz alta y luego verifica. Si fallas alguna, regresa a su sección: es la forma más rápida de convertir lectura en conocimiento.

Pregunta 1: ¿qué devuelve forEach y para qué sirve? Respuesta: devuelve undefined; sirve para ejecutar una acción con cada elemento, como imprimir o guardar. Si necesitas un resultado, forEach es la herramienta equivocada.

Pregunta 2: ¿qué devuelve map y qué longitud tiene el resultado? Respuesta: devuelve un array NUEVO de la misma longitud que el original, con cada elemento transformado. No modifica el original.

Pregunta 3: ¿qué devuelve filter y cuándo un elemento entra al resultado? Respuesta: devuelve un array NUEVO con solo los elementos cuyo predicado devolvió un valor truthy. Los elementos entran intactos, sin transformar.

Pregunta 4: tienes `const r = [1,2,3].map(n => {n * 10;});` ¿qué vale r y por qué? Respuesta: `[undefined, undefined, undefined]`, porque las llaves desactivan el return implícito y el callback no devuelve nada. La forma correcta es `n => n * 10`.

Pregunta 5: ¿cómo accedes a la propiedad de un objeto cuando el nombre de la propiedad está en una variable? Respuesta: con corchetes: `objeto[variable]`. Con punto buscaría una propiedad literalmente llamada como la variable.

Pregunta 6: ¿qué imprime este código? `const o = {a: 1}; o.b = 2; delete o.a; console.log(Object.keys(o));` Respuesta: `["b"]`. La asignación creó b, delete eliminó a, y keys lista las claves restantes.

Pregunta 7: en un método llamado como `perro.ladRAR()`, ¿qué es this? Respuesta: el objeto perro, lo que está a la izquierda del punto en la llamada. Así `this.nombre` es el nombre de ese perro.

Pregunta 8: ¿por qué una función flecha como método que usa this falla? Respuesta: porque la flecha no tiene this propio; hereda el del lugar donde se creó, que no es el objeto. Los métodos con this usan sintaxis de método o function.

Pregunta 9: ¿cuándo SÍ conviene una flecha relacionada con this? Respuesta: como callback DENTRO de un método, porque hereda y conserva el this del método donde una función normal lo perdería.

Pregunta 10: `const a = [{x: 1}]; const b = a.map(o => o); b[0].x = 99; ¿qué vale a[0].x?` Respuesta: 99. El array es nuevo pero los objetos internos se comparten: copia superficial.

Pregunta 11: ¿cómo obtienes los nombres de los productos disponibles de un catálogo en una sola expresión? Respuesta: `catalogo.filter(p => p.disponible).map(p => p.nombre)`. Filtrar primero, transformar después.

Pregunta 12: ¿qué hace `Object.entries` y para qué sirve con lo de esta unidad? Respuesta: convierte un objeto en un array de pares `[clave, valor]`, y sobre ese array ya puedes usar `forEach`, `map` y `filter`.

Pregunta 13: un método hace `this.items = this.items.filter(...)`. ¿Es válido reasignar así?

Respuesta: sí, reasignar una propiedad del objeto mediante `this` es válido y es un patrón funcional limpio: construyes la lista nueva en vez de mutar la vieja.

Pregunta 14: ¿por qué en el cuadro de honor se copió el array antes de `sort`? Respuesta:

porque `sort` modifica el array original en su lugar; sin la copia con `[...alumnos]`, la lista original quedaría desordenada para el resto del programa.

Pregunta 15: ¿qué tienen en común `[...array]` y `{...objeto}`? Respuesta: ambos crean copias

superficiales usando el operador de propagación: el contenedor es nuevo, el contenido interno se comparte por referencia.

Si respondiste bien al menos doce, esta unidad ya es tuya. Si fallaste varias, no te preocupes: relee las secciones correspondientes y vuelve a intentarlo mañana. La memoria se construye con repetición espaciada, no con una sola lectura heroica.

Para llevar más lejos

Si esta unidad te gustó, hay caminos naturales para seguir profundizando por tu cuenta antes de pasar a la siguiente. Primero: reescribe los ejercicios de la unidad 5 que usaban `for manual` con los métodos de esta unidad. Toma el ejercicio del promedio, el de filtrar mayores de edad y el de invertir, y exprésalos con `map` y `filter`. Vas a sentir físicamente cómo el código se encoge y se aclara; esa sensación es la brújula que te dice que vas en la dirección correcta.

Segundo: juega con datos reales. Inventá un array de diez objetos que representen algo que te interese: tus videojuegos con horas jugadas, tus gastos de la semana con categorías, los jugadores de tu equipo favorito con goles. Luego hazte preguntas y respóndelas con código: ¿cuál es el total? ¿cuáles superan cierto umbral? ¿cómo se vería la lista solo con los nombres? Cada pregunta es un `filter`, un `map` o un `forEach` esperando nacer. Programar con datos que te importan hace que la práctica no se sienta como tarea.

Tercero: modela algo de tu vida como un objeto con métodos. Tu alcancía con métodos para ahorrar y gastar con validación; tu lista de pendientes con métodos para agregar, completar y contar pendientes; tu colección de películas con método para recomendar una al azar. Cuando un objeto modela algo tuyo, `this` deja de ser abstracto: es "mi alcancía", "mi lista". Esa conexión emocional con el código acelera el aprendizaje más que cualquier tutorial.

Y un consejo para cuando pases a la siguiente unidad: el DOM, que es cómo JavaScript toca la página web, te va a pedir exactamente estas habilidades. Vas a seleccionar elementos, que vienen en listas, y los vas a recorrer con `forEach` para cambiarlos; vas a tomar datos en forma

de objetos y los vas a pintar en pantalla con `map`. Todo lo que practicaste aquí se usa allá, casi sin traducción. Llegas preparado.

Resumen de la unidad

`forEach` ejecuta una función por cada elemento y no devuelve nada útil: sirve para acciones como imprimir, enviar o guardar. No se puede romper con `break` y siempre recorre todo el array.

`map` transforma cada elemento y devuelve un array NUEVO de la misma longitud, sin modificar el original. Es el método más usado para transformar listas de datos.

`filter` conserva solo los elementos cuyo predicado devuelve `true` y entrega un array NUEVO, igual o más corto, con los elementos intactos. Sirve para seleccionar subconjuntos.

La diferencia clave: `map` responde "¿en qué convierto cada elemento?" y mantiene la cantidad; `filter` responde "¿con cuáles me quedo?" y reduce la cantidad. Si dudas, pregúntate qué quieres: transformar o seleccionar.

El error clásico de `map` es usar llaves sin `return` en el callback y obtener un array de `undefined`. Sin llaves hay `return` implícito; con llaves, el `return` debe ser explícito.

Puedes encadenar métodos: `usuarios.filter(u => u.activo).map(u => u.nombre)` se lee como una frase y es el estilo estándar en código profesional.

El `for` clásico sigue siendo necesario cuando necesitas romper el ciclo, controlar el índice con precisión o recorrer varios arrays en paralelo. Los métodos son herramientas, no religión.

Un objeto literal agrupa pares clave-valor entre llaves: `const persona = {nombre: "Ana", edad: 25}`. Las claves se acceden por nombre, lo que hace el código autoexplicativo.

Accede con punto cuando puedas: `persona.nombre`. Usa corchetes cuando la clave es dinámica o especial: `persona[campo]`. Con punto y una variable obtienes `undefined`: es el error clásico de acceso dinámico.

Agregar o actualizar propiedades es una simple asignación: `persona.ciudad = "México"`. Cuidado con los errores de dedo: crean propiedades basura silenciosas en vez de actualizar.

`delete` elimina una propiedad: `delete persona.edad`. Úsalo poco en código nuevo; muchas veces conviene marcar como inactivo en vez de borrar.

Los objetos se anidan libremente: objetos dentro de objetos, arrays dentro de objetos y objetos dentro de arrays. Navega con puntos y corchetes combinados, leyendo de izquierda a derecha.

Un método es una función dentro de un objeto. La sintaxis corta es `nombre() {...}`. Dentro del método, `this` es el objeto que está a la izquierda del punto en la llamada.

`this` existe para que un método sepa sobre qué objeto está operando, como la palabra "yo" que cambia según quién la dice. La regla de oro: mira qué hay antes del punto al llamar.

NUNCA uses funciones flecha como métodos que necesiten `this`: la flecha hereda el `this` de afuera y obtendrás `undefined`. Métodos con `this` van con sintaxis de método o `function` tradicional.

En cambio, las flechas son perfectas COMO CALLBACKS dentro de métodos: heredan el `this` del método y lo conservan donde una función normal lo perdería.

Los objetos modelan cosas reales: productos con `aplicarDescuento`, cuentas con `depositar` y `retirar`, usuarios con listas de tareas. Datos y reglas viajan juntos: eso es encapsulación.

La estructura más común del mundo real es el array de objetos: `catalogo.map(p => p.nombre)` y `catalogo.filter(p => p.precio < 20)` son operaciones que harás a diario.

Copia objetos con `{...original}`, con la misma advertencia de copia superficial que con arrays: los objetos anidados se comparten por referencia.

Acceder a propiedades de `undefined` truena el programa. Cuando los datos vienen de fuera y pueden estar incompletos, verifica antes de navegar profundo en la estructura.

Con `forEach`, `map`, `filter`, objetos literales, métodos y `this` ya puedes representar catálogos, filtrarlos, transformarlos y modelar entidades con comportamiento: el esqueleto de una aplicación real.

Elige el método por su intención: `forEach` para actuar, `map` para transformar uno a uno, `filter` para seleccionar. Esa decisión comunica tu propósito a quien lea el código.

`this` se define por la llamada: lo que está a la izquierda del punto es el objeto dueño dentro del método. Si despegas el método de su objeto, el `this` se pierde.

La sintaxis abreviada de propiedades `{nombre, edad}` y los métodos escritos como `nombre() {}` son el estilo moderno estándar: menos ruido, misma potencia.

`Object.keys`, `Object.values` y `Object.entries` convierten objetos en arrays y te devuelven al territorio de `map`, `filter` y `forEach` cuando necesitas recorrer un objeto.

Los métodos que llaman a otros métodos con `this` construyen comportamiento por capas sin repetir lógica: cada fórmula vive en un solo lugar.

Cuando dudes entre mutar o crear nuevo, prefiere crear nuevo: `filter` para quitar, `map` para transformar, propagación para copiar. El código sin sorpresas es el código que escala.

Y sobre todo: practica con datos que te importen. Un carrito con tus compras reales enseña más que cien ejercicios abstractos, porque cada bug te duele y cada acierto te motiva a seguir.

UNIDAD 7: Textos dinámicos y el DOM

Hasta aquí has aprendido a manejar datos dentro de JavaScript: variables, operadores, ciclos, funciones, arreglos y objetos. Todo eso vive en la memoria de tu programa, en un mundo invisible para quien visita tu página. Pero pregúntate esto: ¿de qué sirve un programa que calcula cosas increíbles si nadie las puede ver? Imagina que escribes el mejor algoritmo del mundo para calcular el precio final de una compra con impuestos, descuentos y envío, pero el resultado solo existe dentro de una variable que se pierde cuando el programa termina. Sería como un chef que cocina el platillo perfecto y luego lo tira a la basura sin que nadie lo pruebe. Eso es exactamente lo que pasa cuando tu JavaScript no sabe hablar con la página.

Esta unidad cambia todo eso. Aquí vas a aprender la habilidad que separa a un programador que hace scripts de consola de uno que construye experiencias reales: la capacidad de leer y modificar la página web en vivo, mientras el usuario la está viendo. Vas a descubrir los template literals, que son la forma moderna y elegante de construir textos dinámicos en JavaScript, y luego vas a conocer al DOM, el árbol del documento, que es la representación en memoria de tu página HTML que JavaScript puede tocar, mover y transformar.

¿Por qué importa tanto esto? Porque todo lo que amas de la web moderna depende de esta unidad. Cuando escribes en el buscador de Google y aparecen sugerencias debajo mientras tecleas, es JavaScript modificando el DOM en tiempo real. Cuando le das "me gusta" a una foto y el contador sube de 99 a 100 sin recargar la página, es JavaScript tocando un pedacito del DOM. Cuando llenas mal un formulario y aparece un mensaje rojo diciendo "el correo no es válido" sin que la página se recargue, es JavaScript combinando template literals con el DOM. Tu carrito de compras, tus notificaciones, los menús que se despliegan, los carruseles de imágenes, todo es esta unidad puesta en acción.

Antes de JavaScript moderno, las páginas web eran como carteles impresos: el servidor mandaba un HTML fijo y eso era todo. Si querías cambiar una sola palabra, tenías que pedirle al servidor una página completamente nueva. Era lento, torpe y frustrante. El DOM cambió esa historia: convirtió la página en un organismo vivo que JavaScript puede operar, como un cirujano que trabaja con el paciente despierto. Y los template literals le dieron al cirujano las herramientas finas: ya no tienes que andar pegando textos con el operador + como si cosieras con aguja gruesa; ahora construyes cadenas elegantes con interpolación directa de variables.

En esta unidad vas a aprender tres bloques de conocimiento que se encadenan. Primero, los template literals: las comillas invertidas, la interpolación con `${}`, y la magia de los textos multilínea que antes eran un dolor de cabeza. Vas a ver por qué la interpolación no es solo

"más bonita" que la concatenación, sino que reduce errores reales que cuestan horas de depuración. Segundo, el DOM: qué es exactamente, por qué se representa como un árbol, qué son los nodos, y cómo tu HTML se convierte en una estructura que JavaScript puede recorrer como si fuera un mapa. Y tercero, la parte más práctica: cómo seleccionar elementos con `getElementById`, `querySelector` y `querySelectorAll`, y cómo modificarlos con `textContent`, `innerHTML`, atributos y estilos. También vas a aprender algo que te puede ahorrar un disgusto serio: los peligros de `innerHTML` y por qué a veces es mejor no usarlo.

Una advertencia honesta antes de empezar: esta unidad tiene más "partes móviles" que las anteriores. En las unidades pasadas, si algo salía mal, el error aparecía en la consola y ya. Ahora los errores son visuales y a veces silenciosos: cambias un elemento y no pasa nada porque lo seleccionaste mal, o modificas el HTML y se ve raro. No te frustres cuando eso pase. Cada programador web del planeta, sin excepción, ha pasado por el ritual de escribir `document.getElementById("titulo")` y descubrir que el id en el HTML se llamaba "titulo-principal" y por eso nada funcionaba. Es parte del aprendizaje y, créeme, después de la décima vez ya lo haces sin pensar.

Otro punto importante: para practicar esta unidad necesitas un archivo HTML real. Hasta ahora podías probar todo en la consola del navegador, pero el DOM necesita un documento. No te preocupes, es facilísimo: crea un archivo `index.html` con la estructura básica, agrega tus elementos, y enlaza tu script con `<script src="app.js"></script>` justo antes de cerrar el body. O si prefieres, escribe tu JavaScript directamente entre etiquetas `<script>` al final del body. En los ejemplos de esta unidad siempre te voy a mostrar las dos partes: el HTML y el JavaScript, para que veas la conversación completa entre ellos.

Piensa en esta unidad como aprender a hablarle a la página. Hasta ahora tu JavaScript era un monólogo interno: calculaba, decidía, repetía. A partir de ahora es un diálogo: lee lo que hay en la página, lo transforma, y el usuario ve el resultado al instante. Cuando domines esto, vas a mirar cualquier sitio web con ojos nuevos. Vas a ver un menú y pensar "eso es un `addEventListener` con un `classList.toggle`". Vas a ver un contador y pensar "eso es `textContent` actualizado en un intervalo". Esa es la mirada de quien ya entiende cómo funciona la web por dentro, y esta unidad es donde la consigues.

El panorama de lo que viene es emocionante. En la Unidad 8 vas a llevar esta interactividad al siguiente nivel con eventos: clics, formularios y temporizadores. Pero todo eso se construye sobre lo que aprendas aquí. Los eventos te dicen "cuándo" hacer algo; el DOM te dice "qué" tocar y "cómo" cambiarlo. Sin el DOM, los eventos no servirían de nada porque no tendrías nada que modificar. Así que tómate esta unidad con calma, practica cada ejemplo en tu propio archivo, rompe cosas a propósito para ver qué pasa, y llega a la Unidad 8 con el DOM

dominado. Vamos a empezar con los textos dinámicos, porque antes de modificar la página necesitas saber construir los textos que vas a poner en ella.

Hay algo más que quiero que tengas presente mientras lees: la diferencia entre "saber la sintaxis" y "pensar en DOM". Muchos estudiantes memorizan que `querySelector` selecciona elementos y `textContent` cambia el texto, y aun así se atorán cuando tienen que resolver un problema real. ¿Por qué? Porque piensan en líneas de código aisladas en lugar de pensar en el flujo: "tengo este elemento en la página, necesito agarrarlo, leer o cambiar algo de él, y mostrar el resultado". Ese flujo de tres pasos (seleccionar, leer/modificar, mostrar) es el patrón mental que vas a usar el resto de tu carrera. Cada ejemplo de esta unidad lo sigue, y para el final lo vas a tener tan interiorizado que lo harás sin darte cuenta.

También quiero contarte una pequeña historia para que veas lo lejos que llega esto. Hace años, un amigo mío hizo una página para las reservaciones de su pequeño restaurante. Al principio era un formulario que se enviaba al servidor y tardaba varios segundos en responder; la gente se desesperaba y cerraba la página. Un fin de semana le agregé JavaScript con lo que vas a aprender aquí: cuando elegías la fecha, el calendario se actualizaba al instante; cuando escribías tu nombre, aparecía una vista previa de tu reservación con un template literal bonito; si elegías una hora ocupada, el campo se marcaba en rojo sin recargar nada. Las reservaciones en línea se triplicaron en un mes. No cambió el servidor, no cambió el diseño: solo agregó interactividad con el DOM. Eso es el poder real de esta unidad, no es teoría, es dinero y usuarios contentos.

Y una última cosa sobre los template literals antes de entrar en materia: puede que al principio te parezcan un detalle menor, "una forma más bonita de juntar textos". No lo son. En el desarrollo web real, construir HTML dinámico con textos es una de las tareas más comunes que existen: listas de productos, tarjetas de usuarios, mensajes de error, tablas de resultados. Hacerlo con concatenación tradicional es como construir una casa pegando ladrillo por ladrillo con las manos; hacerlo con template literals es usar una grúa. Cuando en la sección de ejemplos veas un template literal generando una tarjeta completa de producto con nombre, precio e imagen en cinco líneas legibles, vas a entender por qué los programadores profesionales casi nunca usan concatenación con `+` para textos complejos.

Respira hondo. Abre tu editor, crea tu archivo HTML de práctica, y vamos a convertir tus scripts invisibles en páginas vivas. Esta es la unidad donde JavaScript deja de ser un ejercicio y se convierte en una herramienta de verdad.

Antes de cerrar esta introducción, déjame darte el mapa mental completo de cómo se conectan las tres clases que agrupa esta unidad, porque ver la conexión desde el principio te

va a ahorrar confusión después. La clase 19 (template literals) te da el "qué decir": la capacidad de construir textos dinámicos que mezclan texto fijo con valores de variables de forma limpia. La clase 20 (el DOM y seleccionar elementos) te da el "dónde decirlo": la capacidad de ubicar exactamente el rincón de la página donde ese texto debe aparecer. Y la clase 21 (modificar el DOM) te da el "cómo decirlo": las herramientas para escribir ese texto en la página, cambiar atributos como el src de una imagen, o modificar estilos al vuelo. Las tres juntas forman el ciclo completo de la interactividad básica: construyes el contenido, encuentras su lugar, y lo colocas. Cada vez que veas una página que se actualiza sin recargarse, alguien ejecutó ese ciclo.

También es importante que entiendas lo que esta unidad NO cubre, para que no te frustres buscando algo que viene después. Aquí vas a modificar la página en respuesta a código que tú ejecutas directamente (por ejemplo, al cargar el script o al llamar una función). Pero todavía no vas a reaccionar a lo que el usuario hace con el mouse o el teclado; eso son los eventos y llegan en la Unidad 8. Es como aprender primero a mover las piezas de ajedrez y después a jugar contra un oponente: necesitas dominar el movimiento antes de reaccionar a jugadas. Así que si en algún ejemplo piensas "¿y cómo hago para que esto pase cuando el usuario haga clic?", anótalo en tu lista de curiosidades y aguanta una unidad más, porque ahí es donde esa pregunta se responde a fondo.

Un consejo práctico que me hubiera gustado que me dieran cuando aprendí esto: mientras lees la sección de explicación detallada, ten tu archivo de práctica abierto y prueba cada fragmento de código en cuanto lo veas. No lo lees como una novela; léelo como un manual de taller. El DOM se aprende con las manos, no con los ojos. Cada vez que el texto diga "selecciona el elemento con id demo", hazlo en tu archivo y abre la consola para inspeccionar qué te devolvió. Esa costumbre de verificar en la consola te va a salvar cientos de horas en tu carrera: la consola es tu laboratorio, tu microscopio, tu lugar seguro para experimentar sin romper nada.

Y ahora sí, con el mapa en la mano y el editor abierto, entremos a la explicación detallada. Prepárate, porque aquí es donde las piezas empiezan a encajar de verdad.

Explicación detallada

Vamos a empezar por los template literals, porque son la puerta de entrada más amable de esta unidad y porque los vas a usar en prácticamente cada ejemplo de DOM que escribas de aquí en adelante. Si vienes de las unidades anteriores, ya sabes construir textos con comillas simples o dobles y pegarlos con el operador `+`. Funciona, pero tiene límites que se vuelven dolorosos muy rápido.

Primero, recordemos el problema. Imagina que tienes el nombre de un usuario en una variable y quieres saludarlo. Con concatenación escribes algo como `"Hola, " + nombre + ", bienvenido a la academia"`. Con dos variables ya se vuelve incómodo; con cinco variables y algo de HTML en medio, se convierte en una sopa de comillas y signos de más donde es facilísimo perder una comilla o un espacio. Y los espacios son traicioneros: ¿cuántas veces has visto un texto que dice `"Hola,Juan"` sin espacio después de la coma? Eso pasa porque el espacio vive dentro de una de las cadenas pegadas y es invisible a simple vista cuando lees el código rápido.

Los template literals resuelven esto con una idea simple y elegante: en lugar de comillas simples o dobles, usas comillas invertidas, esas que se ven así ``` y que en la mayoría de los teclados están en la tecla a la izquierda del `1` o cerca de la tecla de acentos. Dentro de esas comillas invertidas puedes escribir texto normal, y cuando necesitas meter el valor de una variable o el resultado de una expresión, lo pones entre `${y}`. Eso se llama interpolación. Entonces el saludo de antes se escribe `Hola, ${nombre}, bienvenido a la academia``. Léelo en voz alta: se lee casi como la frase final. No hay que contar comillas, no hay que cazar espacios perdidos, el texto se ve como se va a ver.

Pero ojo, que la interpolación con `${}` es más poderosa de lo que parece a primera vista. Dentro de esas llaves no solo puedes poner variables: puedes poner cualquier expresión válida de JavaScript. Puedes hacer `${precio * 1.16}` para calcular el IVA al vuelo, `${nombre.toUpperCase()}`` para poner el nombre en mayúsculas, `${edad >= 18? "adulto": "menor"}`` para elegir un texto según una condición. Esto es importante porque te evita crear variables temporales solo para armar un texto. Claro, con medida: si la expresión dentro de `${}` ocupa tres líneas, mejor calcúlala antes en una variable con un nombre claro y luego interpóla. La legibilidad manda.

Hay un error clásico de principiante que tienes que conocer desde ya: usar `${}` dentro de comillas normales. Si escribes `"Hola, ${nombre}"` con comillas dobles, JavaScript no interpola nada y el usuario verá literalmente `"Hola, ${nombre}"` en la pantalla. Esto pasa más seguido de lo que crees, sobre todo cuando copias código de un ejemplo y cambias las comillas sin darte cuenta, o cuando tu editor autocompleta comillas dobles. La regla de oro es simple: ¿ves `${}`? Entonces las comillas de afuera TIENEN que ser invertidas. Si el texto te sale con el `${nombre}` crudo en pantalla, el 99% de las veces el problema es ese.

La segunda gran ventaja de los template literals son los textos multilínea. Con comillas normales, un texto no puede saltar de línea: si lo intentas, obtienes un error de sintaxis. Antes la gente lo resolvía pegando `"\n"` por todos lados o concatenando línea por línea, y el código quedaba horrible. Con comillas invertidas, simplemente escribes el texto en varias líneas tal como quieres que se vea, y los saltos de línea se conservan. Esto es oro puro cuando generas HTML desde JavaScript, porque el HTML es naturalmente multilínea: una tarjeta de producto, una fila de tabla, un mensaje con formato. Vas a ver ejemplos de esto en la sección de práctica y te prometo que cuando compares la versión con concatenación contra la versión con template literal, no vas a querer volver atrás.

¿Sabías que? Las comillas invertidas también te permiten meter comillas simples y dobles dentro del texto sin escaparlas. Con comillas normales, si tu texto contiene comillas dobles tienes que escaparlas con `\` o cambiar el tipo de comilla exterior, y eso se vuelve un enredo cuando el texto es HTML con atributos (que usan comillas dobles). Con template literals escribes `<div class="tarjeta">` directamente, sin escapes, sin trucos. Este detalle parece menor hasta el día que generas HTML dinámico de verdad; ese día lo vas a agradecer.

Ahora, una pregunta frecuente: ¿debo usar template literals para TODO? La respuesta honesta es: para textos simples y fijos, las comillas normales están perfecto. Nadie te va a regañar por escribir `"Hola mundo"` con comillas dobles. Los template literals brillan cuando el texto es dinámico (mezcla variables), multilínea, o contiene ambos tipos de comillas. Úsalos donde aporten claridad. Un buen programador no usa la herramienta más moderna por moda, sino la que deja el código más legible.

Hablemos ahora del "por qué" profundo de la interpolación, porque aquí hay una lección de diseño de lenguajes que te va a servir. La concatenación con `+` te obliga a pensar en el texto como piezas separadas que hay que ensamblar, y tu cerebro tiene que simular el ensamblado para saber cómo se verá el resultado. La interpolación te deja pensar en el texto como un todo, con huecos marcados donde van los valores. Es la diferencia entre armar un rompecabezas y leer una carta con espacios en blanco. Los estudios de legibilidad de código (y la experiencia de millones de programadores) coinciden: el código que se lee como el

resultado final tiene menos errores. Cada vez que eliges un template literal sobre una concatenación enredada, estás reduciendo la probabilidad de un bug. No es estética, es ingeniería.

Pasemos al segundo gran tema: el DOM. DOM significa Document Object Model, Modelo de Objetos del Documento. Suena intimidante, pero la idea es hermosa en su simplicidad: el navegador toma tu HTML y lo convierte en una estructura de objetos en memoria que JavaScript puede manipular. Tu HTML es texto estático en un archivo; el DOM es ese mismo contenido pero vivo, organizado como un árbol de nodos donde cada etiqueta, cada atributo y cada pedazo de texto es un objeto con propiedades y métodos.

La analogía del árbol no es casualidad, es la forma exacta en que funciona. Piensa en tu página como un árbol genealógico. El elemento `<html>` es el abuelo, el ancestro de todos. Tiene dos hijos: `<head>` y `<body>`. El `<body>` a su vez tiene sus propios hijos: quizá un `<header>`, un `<main>` y un `<footer>`. Cada uno de esos tiene sus hijos, y así hasta llegar a las hojas del árbol: los elementos que ya no tienen hijos, como un `<img>` o un pedazo de texto dentro de un párrafo. Cuando JavaScript "recorre el DOM", está caminando por este árbol genealógico: puede ir al padre de un elemento, a sus hijos, a sus hermanos. Esta estructura de árbol es lo que permite cosas como "encuentra todos los párrafos dentro de este artículo" o "cambia el color del contenedor padre de este botón".

Es crucial que entiendas la diferencia entre el HTML y el DOM, porque no son lo mismo aunque se parezcan. El HTML es el texto fuente, lo que escribiste en tu archivo. El DOM es la representación en memoria que el navegador construye al leer ese HTML. ¿Por qué importa la diferencia? Por dos razones. Primera: el navegador corrige tu HTML al construir el DOM. Si olvidaste cerrar una etiqueta, el navegador la cierra por ti en el DOM (aunque no debas confiarte de eso). Segunda y más importante: JavaScript modifica el DOM, no tu archivo HTML. Cuando cambias el texto de un párrafo con JavaScript, tu archivo.html sigue intacto; lo que cambió es la representación en memoria. Si recargas la página, el navegador reconstruye el DOM desde el archivo original y tus cambios se pierden. Esto confunde a muchos principiantes: "cambié el texto con JavaScript, recargué la página y volvió el texto viejo". No es un bug, es el diseño: el DOM vive en la memoria de esa carga de página.

Otra forma de verlo: el HTML es la receta escrita en papel, el DOM es el platillo ya servido en la mesa. JavaScript es el chef que puede agregarle sal al platillo servido (modificar el DOM), pero la receta en papel no cambia. Si quieres que el cambio sea permanente, tienes que modificar la receta (el archivo) o guardar los datos en otro lado (eso ya es tema de unidades futuras con almacenamiento).

Cada cosa en ese árbol se llama nodo. Hay varios tipos de nodos, pero los que te importan ahora son dos: los nodos de elemento (las etiquetas HTML como `<p>`, `<div>`, `<h1>`) y los nodos de texto (el texto dentro de esas etiquetas). Cuando escribes `<p>Hola</p>`, el DOM crea un nodo de elemento `p` que tiene un hijo: un nodo de texto con el contenido "Hola". Esta distinción explica por qué existen dos propiedades diferentes para cambiar contenido: `textContent` trabaja con el texto, `innerHTML` trabaja con el HTML. Lo veremos a fondo en la clase 21, pero quédate con la idea: el árbol tiene ramas (elementos) y hojas de texto.

El objeto `document` es tu puerta de entrada al DOM. Es un objeto global que representa toda la página, la raíz del árbol. Todo lo que hagas con el DOM empieza con `document`: `document.getElementById`, `document.querySelector`, `document.createElement`. Piensa en `document` como el conserje del edificio: no necesitas saber dónde vive cada inquilino, le preguntas al conserje y él te lleva. Cuando escribes `document.getElementById("titulo")`, le estás diciendo al conserje: "tráeme al inquilino que vive en el departamento con la placa 'titulo'".

Y aquí llegamos a la clase 20: seleccionar elementos. Hay tres métodos que debes dominar, cada uno con su personalidad. El primero es `getElementById`. Como su nombre dice, busca un elemento por su atributo `id`. Los `id` son como el CURP de los elementos: se supone que son únicos en todo el documento. Entonces `document.getElementById("mi-titulo")` te devuelve directamente el elemento (un solo objeto, no una lista). Es el método más rápido y más directo cuando el elemento tiene `id`. La buena práctica es ponerles `id` a los elementos clave que vas a manipular mucho: el título principal, el contenedor de resultados, el campo de búsqueda.

El segundo es `querySelector`, y este es el favorito de los profesionales. Recibe un selector CSS, o sea, la misma sintaxis que usas en tus hojas de estilo.
`document.querySelector(".tarjeta")` selecciona el primer elemento con clase `tarjeta`;
`document.querySelector("#menu")` selecciona el elemento con `id` `menu`;
`document.querySelector("article p")` selecciona el primer párrafo dentro de un `article`. ¿Ves el poder? Con un solo método puedes expresar selecciones complejas usando el lenguaje de selectores que ya conoces de CSS. Devuelve el primer elemento que coincida, o `null` si no encuentra nada. Ese `null` es importante: si tu selector está mal escrito o el elemento no existe, no obtienes un error inmediato, obtienes `null`, y el error aparece después cuando intentas hacer algo con ese `null`. Por eso muchos bugs de DOM se manifiestan como "Cannot read properties of null". Cuando veas ese error, tu primer sospechoso debe ser el selector.

El tercero es `querySelectorAll`, el hermano plural de `querySelector`. Devuelve TODOS los elementos que coinciden con el selector, en una lista especial llamada `NodeList`.
`document.querySelectorAll(".producto")` te da todos los elementos con clase `producto`. Y aquí

viene un detalle que causa confusión: una NodeList se parece a un arreglo pero no es un arreglo. Puedes recorrerla con `forEach` y con un ciclo `for`, y puedes acceder a sus elementos con corchetes como `lista[0]`, pero no tienes métodos como `map`, `filter` o `push`. Si necesitas esos métodos, la conviertes a arreglo de verdad con `Array.from(lista)` o con el operador `spread` `[...lista]`. Este es uno de esos casos borde que te muerden una vez y no se te olvidan: intentas hacer `lista.map` y obtienes `"lista.map is not a function"`, y te quedas mirando la pantalla hasta que recuerdas que es una NodeList.

Existe también `getElementsByClassName` y `getElementsByTagName`, los abuelos de `querySelectorAll`. Todavía los verás en código viejo y funcionan bien, pero devuelven una `HTMLCollection` que es aún más limitada (ni siquiera tiene `forEach` en navegadores viejos). Mi recomendación: aprende que existen para cuando los veas en código ajeno, pero en tu código nuevo usa `querySelector` y `querySelectorAll`. Son más expresivos, más modernos y más consistentes.

Hablemos de un tema que parece trivial pero causa el 30% de los dolores de cabeza con el DOM: el momento en que se ejecuta tu script. Si tu `<script>` está en el `<head>` y trata de seleccionar un elemento del `<body>`, el elemento todavía no existe cuando el script corre, porque el navegador lee el HTML de arriba hacia abajo. El resultado: tu selector devuelve `null` y nada funciona, sin ningún error claro al principio. Hay tres soluciones. La más simple y la que te recomiendo mientras aprendes: pon tu `<script>` justo antes de cerrar el `</body>`, al final del documento. Así todo el HTML ya existe cuando tu código corre. La segunda: usa el atributo `defer` en la etiqueta script, que le dice al navegador "descarga esto pero ejecútalo hasta que el documento esté listo". La tercera: envuelve tu código en un listener del evento `DOMContentLoaded`, que espera a que el árbol esté completo. Las tres funcionan; la primera es la más fácil de entender y por eso la usaremos en los ejemplos.

Una anécdota real para que veas que esto le pasa a todos: una vez pasé casi una hora depurando una página donde un botón "no hacía nada". Revisé el selector, revisé la lógica, todo se veía bien. El problema era que había copiado el `<script>` al `<head>` "para que se viera más ordenado" y el botón aún no existía cuando el código intentó seleccionarlo. Desde ese día, script al final del body por defecto, sin excepciones, hasta que tengas una razón consciente para hacer otra cosa.

Ahora la clase 21: modificar el DOM. Ya sabes agarrar elementos; ahora vas a cambiarlos. La propiedad más básica es `textContent`. Sirve para leer o cambiar el texto dentro de un elemento. Si tienes `<p id="saludo">Hola</p>` y haces `document.getElementById("saludo").textContent = "Adiós"`, el párrafo ahora dice "Adiós". Simple, seguro y predecible. `textContent` trata todo como texto plano: si le asignas

"Hola", el usuario verá literalmente los caracteres Hola en pantalla, no un texto en negritas. Eso es una característica de seguridad, no una limitación, como verás en un momento.

Su contraparte es innerHTML, que lee o cambia el contenido interpretándolo como HTML. Si haces elemento.innerHTML = "Hola", el usuario verá "Hola" en negritas, porque el navegador interpreta la etiqueta. Esto es poderosísimo para generar contenido dinámico: puedes construir una tarjeta completa de producto con un template literal y meterla en la página de un solo golpe. Pero tiene dos costos. El primero es rendimiento: cada vez que asignas innerHTML, el navegador tiene que interpretar el texto como HTML y reconstruir esa parte del árbol, lo cual es más caro que cambiar solo el texto. Para cambios ocasionales no importa; para actualizar algo 60 veces por segundo sí. El segundo costo es serio: seguridad. Si metes en innerHTML texto que escribió un usuario sin limpiarlo, un atacante puede inyectar <script> malicioso. Esto se llama XSS (cross-site scripting) y es una de las vulnerabilidades más comunes de la web. La regla práctica: usa.textContent para textos (siempre seguro), y reserva innerHTML para cuando realmente necesites meter HTML, asegurándote de que el contenido venga de una fuente confiable o esté sanitizado.

Más allá del contenido, puedes modificar atributos con setAttribute y getAttribute, o con las propiedades directas. Por ejemplo, para cambiar la imagen de un , puedes hacer imagen.src = "nueva-foto.jpg" (propiedad directa) o imagen.setAttribute("src", "nueva-foto.jpg"). Ambas funcionan; las propiedades directas son más cortas para los atributos comunes (src, href, value, id, className). Para atributos personalizados (los que empiezan con data-, como data-precio), usa getAttribute y setAttribute, o la propiedad dataset que es la forma moderna: elemento.dataset.precio lee el atributo data-precio. Los atributos data- son una joya poco conocida: te permiten guardar información extra en el HTML (como el precio o el id de un producto) y leerla desde JavaScript sin mostrarla al usuario.

Y luego están los estilos. Puedes cambiar el CSS de un elemento desde JavaScript con la propiedad style: elemento.style.color = "red", elemento.style.fontSize = "20px". Dos detalles importantes. Primero: en JavaScript, las propiedades CSS con guion se escriben en camelCase. background-color se vuelve backgroundColor, font-size se vuelve fontSize. Si escribes elemento.style.background-color, obtienes un error de sintaxis porque el guion se interpreta como resta. Segundo: los valores numéricos que llevan unidades deben ir como texto con la unidad incluida: "20px", no 20. Si asignas elemento.style.width = 200 sin "px", no pasa nada (el navegador lo ignora silenciosamente en la mayoría de los casos), y te quedas preguntándote por qué no se aplicó el estilo.

Pero aquí viene una buena práctica que separa al principiante del profesional: para cambios de estilo significativos, es mejor manipular clases con `classList` que asignar estilos en línea uno por uno. En lugar de hacer `elemento.style.color = "red"; elemento.style.fontWeight = "bold"; elemento.style.border = "2px solid red"`, defines una clase `error` en tu CSS con todo eso y haces `elemento.classList.add("error")`. ¿Por qué es mejor? Tres razones: tu CSS sigue siendo el único lugar donde vive el diseño (separación de responsabilidades), puedes quitar todos los estilos de un golpe con `classList.remove`, y puedes alternar estados con `classList.toggle`, que es perfecto para menús, modales y modos oscuros. Reserva `elemento.style` para cambios dinámicos calculados en el momento, como la posición de un elemento que sigue al mouse o el ancho de una barra de progreso que depende de un porcentaje.

`classList` tiene cuatro métodos que debes memorizar: `add` (agrega una clase), `remove` (la quita), `toggle` (la agrega si no está, la quita si está; devuelve `true` o `false` según lo que hizo) y `contains` (pregunta si el elemento tiene la clase). Con esos cuatro puedes manejar casi cualquier cambio visual. Un patrón que vas a usar mil veces: `boton.classList.toggle("activo")` para encender y apagar estados. Es tan común que cuando veas un menú hamburguesa que se abre y se cierra, ya sabes qué hay detrás.

Hablemos de errores comunes al modificar el DOM, porque aquí es donde la mayoría se atora. Error número uno: intentar modificar un elemento que es `null` porque el selector falló. Siempre que algo "no hace nada", verifica primero que tu selección devolvió lo que esperabas con un `console.log`. Error número dos: confundir `textContent` con `value` en los inputs. Los campos de formulario (`<input>`, `<textarea>`, `<select>`) no usan `textContent` ni `innerHTML` para su contenido; usan la propiedad `value`. Si haces `input.textContent = "hola"`, no verás nada en el campo; tienes que hacer `input.value = "hola"`. Y para leer lo que el usuario escribió, lees `input.value`. Esto lo veremos a fondo en la Unidad 8 con formularios, pero adelanto la regla para que no te confundas desde ahora: elementos normales usan `textContent/innerHTML`, campos de formulario usan `value`.

Error número tres: pensar que `innerHTML +=` agrega contenido "al final" de forma barata. Mucha gente hace `lista.innerHTML += "<li>nuevo</li>"` dentro de un ciclo para agregar elementos uno por uno. Funciona, pero en cada iteración el navegador reconstruye TODO el contenido del elemento, así que con 1000 elementos haces 1000 reconstrucciones. La forma eficiente es construir todo el HTML en un template literal dentro del ciclo (acumulando en una variable de texto) y asignarlo una sola vez al final. O mejor aún, usar `createElement` y `appendChild`, que veremos en los ejemplos. Para listas pequeñas no notarás la diferencia, pero es un hábito que conviene formar desde temprano.

Error número cuatro: olvidar que los cambios en el DOM no se guardan. Ya lo mencioné, pero lo repito porque es la confusión más reportada: modificas la página con JavaScript, recargas, y todo vuelve a como estaba. El DOM es memoria temporal de la página actual. Si necesitas persistencia, necesitas almacenamiento (localStorage, que verás más adelante, o un servidor).

Ahora, preguntas frecuentes que siempre aparecen. "¿Puedo seleccionar elementos antes de que existan?" No con los métodos normales; el elemento debe estar en el DOM cuando lo buscas. Si creas elementos dinámicamente, guárdalos en variables cuando los crees. "¿Qué pasa si dos elementos tienen el mismo id?" El HTML se vuelve inválido y getElementById te devuelve el primero que encuentre, ignorando los demás. Los id duplicados son una fuente clásica de bugs misteriosos: "¿por qué solo cambia el primer elemento de mi lista?" Porque todos tienen el mismo id y solo estás tocando el primero. Para grupos de elementos usa clases, no ids. "¿querySelector es más lento que getElementById?" Técnicamente sí, una pizca, porque tiene que interpretar el selector CSS. En la práctica, la diferencia es insignificante a menos que lo llares miles de veces por segundo. Prioriza la claridad.

"¿Debo guardar los elementos seleccionados en variables?" Sí, casi siempre. Si vas a usar un elemento más de una vez, selecciónalo una vez y guárdalo en una const con un nombre claro: `const titulo = document.getElementById("titulo")`. Esto es más rápido (no recorres el árbol cada vez) y más legible. Es una de esas micro-buenas-prácticas que distinguen el código cuidado.

Cuidado con... un detalle sutil de querySelectorAll: la NodeList que devuelve es estática, es una foto del momento. Si seleccionas todos los `item` y después agregas un nuevo `item` a la página, tu NodeList vieja no lo incluye. Tienes que volver a seleccionar. En cambio, `getElementsByClassName` devuelve una HTMLCollection "viva" que sí se actualiza sola, pero eso trae sus propios problemas (como ciclos infinitos si agregas elementos mientras la recorres). Otro argumento para quedarte con querySelectorAll y simplemente volver a seleccionar cuando lo necesites.

¿Sabías que? El DOM no es exclusivo de los navegadores. Es un estándar que también existe en otros entornos: Node.js puede manipular HTML con librerías que implementan el DOM, y las herramientas de web scraping recorren el DOM de las páginas para extraer datos. Lo que aprendes aquí te sirve más allá del navegador. Y otro dato curioso: el DOM es lentísimo comparado con las operaciones normales de JavaScript, en términos relativos. Por eso los frameworks modernos como React inventaron el "DOM virtual": en lugar de tocar el DOM real a cada rato, trabajan en una copia en memoria y solo aplican los cambios necesarios al final. Cuando entiendas el costo de modificar el DOM, vas a entender por qué existen esos frameworks.

Para cerrar esta explicación, quiero darte el modelo mental completo en una frase que puedes repetirte: el DOM es el puente entre tu HTML estático y tu JavaScript dinámico; los selectores son cómo cruzas el puente para encontrar cosas; `textContent`, `innerHTML`, los atributos y las clases son las herramientas que llevas en la mochila para transformar lo que encuentres; y los `template literals` son el idioma en el que escribes los mensajes que vas a colocar. Con ese modelo en la cabeza, los ejemplos que vienen a continuación se van a sentir naturales, como ver la teoría en acción. Y recuerda: cada concepto aquí tiene su momento de "clic" mental, ese instante en que deja de ser sintaxis y se vuelve intuición. Si algún concepto todavía se siente borroso, es normal; el clic llega cuando lo practicas, no cuando lo lees. Vamos a los ejemplos.

Ejemplos paso a paso

Vamos a ensuciarnos las manos. Cada ejemplo sigue el mismo ritual: primero el HTML, luego el JavaScript, y después una explicación de qué está pasando línea por línea. Te recomiendo que escribas cada ejemplo en tu archivo de práctica y lo ejecutes, porque leer código de DOM sin ejecutarlo es como leer recetas sin cocinar: entiendes las palabras pero no el sabor. Todos los ejemplos asumen que tu `<script>` está justo antes del `</body>`, como acordamos.

Ejemplo 1: tu primer `template literal` con interpolación. Este es el "hola mundo" de los textos dinámicos. Tenemos dos variables, `nombre` y `curso`, y queremos armar un mensaje de bienvenida. Con concatenación sería un enredo de comillas; con el `template literal` se lee de corrido. Fíjate en cómo el `${}` marca exactamente los huecos donde van los valores, y en cómo el texto alrededor se escribe normal, con sus espacios y su puntuación, sin tener que partir la cadena en pedazos. Ejecútalo y mira la consola: el resultado es un texto limpio y natural.


```
// Datos que podrían venir de un formulario o de una base de datos
const nombre = "María";
const curso = "JavaScript";
const leccion = 7;

// Con concatenación tradicional (así se hacía antes)
const mensajeViejo = "Hola " + nombre + ", bienvenida a la lección " + leccion + "
del curso de " + curso + ".";
console.log(mensajeViejo);

// Con template literal e interpolación (la forma moderna)
const mensajeNuevo = `Hola ${nombre}, bienvenida a la lección ${leccion} del curso
de ${curso}.`;
console.log(mensajeNuevo);

// Dentro de ${} puedes poner CUALQUIER expresión, no solo variables
const precio = 500;
console.log(`El precio con IVA es ${precio * 1.16} pesos.`);
console.log(`Tu nombre en mayúsculas es ${nombre.toUpperCase()}.`);
console.log(`Estatus: ${leccion >= 7 ? "ya vas a la mitad" : "apenas empiezas"}.`);
```

Nota algo importante en ese ejemplo: para mostrar el signo de pesos antes del precio tuve que escribir ``${precio * 1.16}``. El primer `$` es un signo de pesos literal, y el segundo `$` abre la interpolación `${}`. Al principio se ve raro, pero tiene lógica: el navegador lee `$` suelto como texto normal y `${}` como inicio de interpolación. Es un detalle que confunde la primera vez que construyes textos con precios en pesos mexicanos, así que mejor que lo veas aquí y no en un bug a las 2 de la mañana.

Ejemplo 2: textos multilínea y generación de HTML. Aquí es donde los template literals muestran su verdadero poder. Vamos a generar la tarjeta de un producto con nombre, precio e imagen. Con comillas normales esto sería una pesadilla de concatenación; con el template literal escribes el HTML tal como quieres que se vea, en varias líneas, con las variables interpoladas en su lugar. Observa también que las comillas dobles de los atributos HTML (`class="tarjeta"`) conviven sin problema dentro de las comillas invertidas, sin necesidad de escapar nada.

```

const producto = "Audífonos Bluetooth";
const precioProducto = 899;
const imagen = "audifonos.jpg";
const descuento = true;

// Construimos una tarjeta HTML completa con un template literal multilínea
const tarjeta = `
  <div class="tarjeta">
    
    <h3>${producto}</h3>
    <p class="precio">${precioProducto}</p>
    ${descuento ? '<p class="oferta">¡En oferta esta semana!</p>' : ''}
  </div>
`;

console.log(tarjeta);

```

Hay dos cosas que quiero que notes con lupa. Primera: la línea del descuento usa un operador ternario dentro de la interpolación para decidir si incluye o no el párrafo de oferta. Esta técnica (meter un ternario que devuelve un pedazo de HTML o un texto vacío) es extremadamente común en el desarrollo real: es como tener un interruptor dentro de tu plantilla. Segunda: el HTML generado conserva los saltos de línea y la indentación, lo cual no afecta cómo se ve en el navegador (el HTML ignora los espacios extra) pero hace que tu código sea legible. Cuando generes listas de 20 productos con un ciclo, vas a agradecer cada salto de línea.

Ejemplo 3: conociendo el árbol. Antes de seleccionar nada, vamos a inspeccionar la estructura. Este ejemplo no cambia nada en la página; solo explora. Crea este HTML y luego usa la consola para navegar por el árbol: del document al body, del body a sus hijos, de un elemento a su padre. Este ejercicio de exploración es el que construye tu intuición del DOM como árbol. Cuando puedas "ver" el árbol en tu mente mirando un HTML, seleccionar elementos se vuelve trivial.

```

<!-- Guárdalo como index.html y ábrelo en el navegador -->
<h1 id="titulo">Mi página de práctica</h1>
<div id="contenedor">
  <p class="texto">Primer párrafo</p>
  <p class="texto">Segundo párrafo</p>
</div>

```

```
// Explora el árbol desde la consola o desde tu script
console.log(document);           // todo el documento
console.log(document.body);      // el <body>
console.log(document.body.children); // los hijos directos del body

const contenedor = document.getElementById("contenedor");
console.log(contenedor.children); // los dos párrafos
console.log(contenedor.parentNode); // el body (el papá del contenedor)
console.log(contenedor.firstChild); // el primer párrafo
```

Fíjate en la diferencia entre `children` y `childNodes`, un caso borde clásico: `children` solo devuelve nodos de elemento (las etiquetas), mientras que `childNodes` devuelve TODO incluyendo los nodos de texto (los espacios y saltos de línea entre etiquetas también son nodos de texto). Por eso a veces `childNodes` te devuelve más cosas de las que esperabas, incluyendo "textos" que son solo el salto de línea de tu HTML. Para el 95% de los casos, usa `children` y `firstElementChild`, que ignoran esos nodos de texto invisibles y te dan exactamente las etiquetas.

Ejemplo 4: los tres selectores en acción. Este es el ejemplo central de la clase 20. Tenemos una pequeña página con varios elementos y vamos a seleccionarlos de las tres formas, para que compares cuándo conviene cada una. Presta atención a qué devuelve cada método: `getElementById` y `querySelector` devuelven un elemento (o null), `querySelectorAll` devuelve una `NodeList` con todos los que coincidan.

```
<!-- HTML de práctica -->
<h1 id="titulo-principal">Catálogo</h1>
<p class="descripcion">Nuestra selección de hoy</p>
<ul id="lista">
  <li class="producto">Tacos al pastor</li>
  <li class="producto">Quesadillas</li>
  <li class="producto">Agua de horchata</li>
</ul>
```

```
// 1. getElementById: directo y rapidísimo cuando hay un id
const titulo = document.getElementById("titulo-principal");
console.log(titulo); // <h1 id="titulo-principal">Catálogo</h1>

// 2. querySelector: con sintaxis de CSS, devuelve el PRIMERO que coincida
const primerProducto = document.querySelector(".producto");
console.log(primerProducto.textContent); // "Tacos al pastor" (solo el primero)

const parrafoDentroDeLista = document.querySelector("#lista .producto");
console.log(parrafoDentroDeLista.textContent); // también el primero

// 3. querySelectorAll: devuelve TODOS los que coincidan (NodeList)
const productos = document.querySelectorAll(".producto");
console.log(productos.length); // 3

// Recorremos la NodeList con forEach (sí tiene forEach)
productos.forEach(function(item, indice) {
  console.log(indice + ": " + item.textContent);
});

// Pero NO tiene map: esto fallaría
// const nombres = productos.map(p => p.textContent); // ERROR

// Si necesitas métodos de arreglo, conviértela primero
const nombres = [...productos].map(p => p.textContent);
console.log(nombres); // ["Tacos al pastor", "Quesadillas", "Agua de horchata"]
```

El último truco (convertir la NodeList con [...productos]) es de esos que parecen magia la primera vez. El operador spread... "desparrama" los elementos de la NodeList dentro de un arreglo nuevo, y ese arreglo sí tiene map, filter y todo lo demás. Guárdalo en tu caja de herramientas porque lo vas a usar muchísimo cuando combines selección de DOM con métodos de arreglos de las unidades anteriores. Mira qué bonito se conecta todo: lo que aprendiste de map en la Unidad 6 ahora trabaja con elementos de la página.

Ejemplo 5: `textContent` contra `innerHTML`, la comparación definitiva. Este ejemplo muestra lado a lado la diferencia crucial entre las dos propiedades. Tenemos dos párrafos vacíos y les asignamos el mismo texto que contiene una etiqueta `<b>`. Uno usa `textContent` y el otro `innerHTML`. El resultado visual es completamente distinto, y entender por qué es entender la diferencia entre "texto" y "HTML" en el DOM.

```
<!-- HTML -->
<p id="con-texto"></p>
<p id="con-html"></p>
```

```
const parrafoTexto = document.getElementById("con-texto");
const parrafoHtml = document.getElementById("con-html");

const contenido = "Oferta <b>imperdible</b> solo hoy";

// textContent: todo se trata como TEXTO PLANO
parrafoTexto.textContent = contenido;
// El usuario ve literalmente: Oferta <b>imperdible</b> solo hoy
// (con los <b> visibles, sin negritas)

// innerHTML: el texto se interpreta como HTML
parrafoHtml.innerHTML = contenido;
// El usuario ve: Oferta imperdible solo hoy
// (con "imperdible" en negritas, los <b> no se ven)
```

Ahora el ejemplo de seguridad, el que justifica toda la advertencia. Imagina un campo donde el usuario escribe su nombre y tú lo muestras en la página. Si usas innerHTML con lo que escribió el usuario, y el usuario escribe algo malicioso, tienes un problema grave. Mira la diferencia:

```
// Simulemos lo que escribiría un usuario travieso
const nombreUsuario = '';

// PELIGROSO: innerHTML ejecuta el HTML del usuario
// document.getElementById("bienvenida").innerHTML = "Hola " + nombreUsuario;
// Esto mostraría una imagen rota Y ejecutaría el código del onerror. ¡No lo hagas!

// SEGURO: textContent muestra el texto tal cual, sin interpretarlo
document.getElementById("bienvenida").textContent = "Hola " + nombreUsuario;
// El usuario ve el texto raro tal cual lo escribió, pero nada se ejecuta.
```

La regla de oro que debes tatuarte: textContent para todo lo que venga del usuario o de fuentes que no controlas; innerHTML solo para HTML que tú mismo construiste en tu código (como las tarjetas de producto del Ejemplo 2, donde los datos vienen de tus variables, no de un input libre). Esta distinción es la que separa una página segura de una vulnerable, y los reclutadores la preguntan en entrevistas. Ya la sabes.

Ejemplo 6: cambiando atributos e imágenes. Vamos a hacer algo visualmente satisfactorio: una galería donde cambiamos la imagen mostrada. Esto demuestra que el DOM no es solo texto: puedes tocar cualquier atributo. Usaremos la propiedad directa `src`, que es la forma corta y legible, y también veremos `setAttribute` para que conozcas ambas.

```
<!-- HTML -->

<p id="descripcion" data-categoria="naturaleza">Foto de un paisaje</p>
```

```
const foto = document.getElementById("foto");

// Cambiar el src con la propiedad directa (forma recomendada para atributos comunes)
foto.src = "foto2.jpg";
foto.alt = "Atardecer en la playa";

// Lo mismo con setAttribute (útil para atributos personalizados o menos comunes)
foto.setAttribute("src", "foto3.jpg");
foto.setAttribute("alt", "Montañas al amanecer");

// Leer atributos
console.log(foto.getAttribute("src")); // "foto3.jpg"

// Atributos data-*: información extra invisible para el usuario
const descripcion = document.getElementById("descripcion");
console.log(descripcion.dataset.categoria); // "naturaleza"
descripcion.dataset.categoria = "viajes"; // cambia data-categoria a "viajes"
```

Los atributos `data-` merecen un momento extra porque son una de esas herramientas que los principiantes tardan en descubrir y luego usan para todo. Imagina una lista de productos donde cada tarjeta necesita "recordar" su precio y su id para cuando el usuario haga clic (eso lo verás en la Unidad 8). En lugar de inventar variables globales enredadas, guardas los datos en el HTML: `<div class="producto" data-id="42" data-precio="899">`. Desde JavaScript los lees con `elemento.dataset.id` y `elemento.dataset.precio`. El HTML lleva los datos pegados al elemento al que pertenecen, como etiquetas en maletas de viaje. Limpio, organizado y sin variables globales.

Ejemplo 7: estilos y clases, la forma profesional. Vamos a construir un validador visual simple: un campo que se pone verde si el texto es válido y rojo si no. Primero te muestro la forma ingenua (estilos en línea uno por uno) y luego la forma profesional (clases con classList). Compara y dime cuál preferirías mantener dentro de seis meses.

```
<!-- HTML -->
<style>
  .valido { border: 2px solid green; background-color: #e8f5e9; }
  .invalido { border: 2px solid red; background-color: #ffebee; }
  .resaltado { font-weight: bold; color: darkblue; }
</style>
<input id="correo" type="text" placeholder="tu@correo.com">
<p id="mensaje"></p>
```

```
// FORMA INGENUA: estilos en línea, uno por uno (funciona pero es tediosa)
const campo = document.getElementById("correo");
const mensaje = document.getElementById("mensaje");
```

```
function validarIngenua(texto) {
  if (texto.includes("@")) {
    campo.style.border = "2px solid green";
    campo.style.backgroundColor = "#e8f5e9";
    mensaje.textContent = "Correo válido";
    mensaje.style.color = "green";
  } else {
    campo.style.border = "2px solid red";
    campo.style.backgroundColor = "#ffebee";
    mensaje.textContent = "Falta el @ en el correo";
    mensaje.style.color = "red";
  }
}
```

```
// FORMA PROFESIONAL: clases CSS + classList (limpia y mantenible)
function validarProfesional(texto) {
  const esValido = texto.includes("@");

  campo.classList.remove("valido", "invalido"); // limpiamos estado anterior
  mensaje.classList.remove("resaltado");

  if (esValido) {
    campo.classList.add("valido");
    mensaje.textContent = "Correo válido";
  } else {
```

```

    campo.classList.add("invalido");
    mensaje.textContent = "Falta el @ en el correo";
    mensaje.classList.add("resaltado");
  }

  console.log("¿El campo está marcado como válido?",
campo.classList.contains("valido"));
}

// Probemos la profesional
validarProfesional("juan@example.com"); // campo en verde
validarProfesional("juanexample.com"); // campo en rojo

// toggle: el interruptor mágico para menús, modos oscuros, favoritos
// Cada vez que se llama, prende o apaga la clase
mensaje.classList.toggle("resaltado"); // si la tenía, la quita; si no, la pone

```

Fíjate en el detalle de `classList.remove("valido", "invalido")` al inicio: limpiamos el estado anterior antes de aplicar el nuevo. Sin esa línea, un campo que primero fue inválido y luego válido tendría AMBAS clases, y el resultado visual dependería de cuál clase gane en el CSS, un bug sutil y molesto. Este patrón de "limpiar antes de aplicar" aparece en todo el desarrollo frontend: menús, tabs, pasos de un formulario. Grábatelo.

Ejemplo 8: generando una lista completa con template literals e innerHTML. Este es el ejemplo que une todo lo de la unidad: tomamos un arreglo de datos (de la Unidad 6), construimos el HTML con un template literal dentro de un ciclo (acumulando en una variable), y lo insertamos de una sola vez. Es exactamente lo que hacen las tiendas en línea para mostrar productos, lo que hacen los blogs para mostrar artículos, lo que hacen las apps para mostrar comentarios.

```

<!-- HTML -->
<h2>Menú del día</h2>
<div id="menu"></div>

```



```
// Datos (podrían venir de un servidor en una app real)
const platillos = [
  { nombre: "Tacos al pastor", precio: 85, picante: true },
  { nombre: "Quesadilla de huitlacoche", precio: 65, picante: false },
  { nombre: "Agua de horchata", precio: 30, picante: false },
  { nombre: "Salsa habanero extra", precio: 15, picante: true }
];

// Construimos TODO el HTML primero, en una variable de texto
let htmlMenu = "";
for (const platillo of platillos) {
  htmlMenu += `
    <div class="platillo">
      <h3>${platillo.nombre} ${platillo.picante ? "[picante]" : ""}</h3>
      <p>$$${platillo.precio} pesos</p>
    </div>
  `;
}

// Y lo insertamos UNA SOLA VEZ (eficiente: una sola reconstrucción del DOM)
document.getElementById("menu").innerHTML = htmlMenu;
```

Este patrón (acumular en una variable + asignar una vez) es tan fundamental que merece su propia explicación del "por qué". Cada asignación a innerHTML obliga al navegador a tirar a la basura la estructura anterior de ese elemento y reconstruirla desde cero interpretando el HTML. Si lo haces dentro del ciclo con +=, con 100 platillos haces 100 reconstrucciones, cada una más grande que la anterior. Acumulando el texto y asignando una vez, haces UNA reconstrucción. Para 4 platillos no se nota; para 400 sí, y mucho. Además, el código queda más claro: primero construyes, luego insertas. Dos fases, dos responsabilidades.

Ejemplo 9: createElement y appendChild, la alternativa quirúrgica. innerHTML es como remodelar tirando un muro y construyéndolo de nuevo; createElement es como poner ladrillo por ladrillo. Para agregar un solo elemento, la forma quirúrgica es más precisa y evita reconstruir lo que ya existe. Vamos a agregar elementos a una lista uno por uno, creando cada nodo con document.createElement, configurándolo, y pegándolo con appendChild.

```
<!-- HTML -->
<ul id="tareas"></ul>
```

```
// Creamos un <li> desde cero, sin innerHTML
const nuevaTarea = document.createElement("li"); // nace un <li> vacío
nuevaTarea.textContent = "Estudiar el DOM";      // le ponemos texto (seguro)
nuevaTarea.classList.add("pendiente");           // le ponemos clase
nuevaTarea.dataset.prioridad = "alta";           // le guardamos un dato extra

// Lo pegamos al final de la lista (ya existía en el DOM, no se reconstruye nada)
document.getElementById("tareas").appendChild(nuevaTarea);

// Otro más, en una función reutilizable (así se hace en la vida real)
function agregarTarea(texto, prioridad) {
  const li = document.createElement("li");
  li.textContent = texto;
  li.dataset.prioridad = prioridad;
  li.classList.add(prioridad === "alta" ? "urgente" : "normal");
  document.getElementById("tareas").appendChild(li);
}

agregarTarea("Hacer la práctica de la unidad 7", "alta");
agregarTarea("Regar las plantas", "baja");
```

¿Cuándo usar innerHTML y cuándo createElement? Mi regla práctica: si vas a generar mucho HTML de un golpe a partir de datos (una lista completa, una tabla), innerHTML con template literal es más legible. Si vas a agregar o quitar elementos individuales con frecuencia (una lista de tareas donde el usuario agrega una por una), createElement + appendChild es más eficiente y más seguro (puedes usar textContent para el texto del usuario sin riesgo de XSS). No es una ley, es un criterio; con la experiencia desarrollarás tu propio olfato.

Ejemplo 10: mini-reto guiado, el termómetro de la unidad. Vamos a juntar selectores, template literals y modificación en un mini proyecto: una tarjeta de perfil que se genera desde un objeto. Te doy el código comentado paso a paso; tu trabajo después será modificarlo (cámbiale los datos, agrégale un campo nuevo, cambia los estilos). Este ejemplo es deliberadamente más largo porque simula un componente real.

```
<!-- HTML -->
<div id="perfil"></div>
```

```

// Paso 1: los datos (imagina que vienen de una base de datos)
const usuario = {
  nombre: "Ana García",
  edad: 28,
  profesion: "Diseñadora UX",
  habilidades: ["Figma", "CSS", "Prototipado"],
  disponible: true
};

// Paso 2: construimos las habilidades como HTML con map + join
// (mira cómo se conectan las unidades: map de la Unidad 6 + template literals)
const listaHabilidades = usuario.habilidades
  .map(habilidad => `<span class="tag">${habilidad}</span>`)
  .join("");

// Paso 3: armamos la tarjeta completa con un template literal
const tarjetaPerfil = `
  <div class="tarjeta-perfil">
    <h2>${usuario.nombre}</h2>
    <p>${usuario.edad} años · ${usuario.profesion}</p>
    <div class="habilidades">${listaHabilidades}</div>
    <p class="estado ${usuario.disponible ? "si" : "no"}">
      ${usuario.disponible ? "Disponible para proyectos" : "No disponible"}
    </p>
  </div>
`;

// Paso 4: la insertamos en la página
document.getElementById("perfil").innerHTML = tarjetaPerfil;

```

Detengámonos en el paso 2 porque ahí hay una técnica que vale oro: `.map(...).join("")`. El `map` convierte cada habilidad en un pedazo de HTML (un `<span>`), produciendo un arreglo de textos. El `join("")` los pega todos en un solo texto, sin separador (las comillas vacías significan "pégalos directo, sin comas ni espacios"). Sin el `join`, JavaScript pegaría los elementos del arreglo con comas entre ellos y verías comas raras en tu página. Este `map + join` es EL patrón estándar para generar listas de HTML desde arreglos; lo verás en tutoriales, en código profesional y en frameworks. Si solo te llevas una técnica de esta unidad, que sea esta.

Y con esto cerramos los ejemplos. Si los ejecutaste todos, ya hiciste lo esencial: interpolar textos, recorrer el árbol, seleccionar de tres formas, modificar texto y HTML con criterio de seguridad, cambiar atributos, manejar clases como un profesional, generar listas desde datos y crear elementos quirúrgicamente. La sección de práctica va a convertir todo eso en habilidad real, porque leer ejemplos te da conocimiento y resolver ejercicios te da oficio.

Práctica

Es hora de trabajar. Estos ejercicios están ordenados de menor a mayor dificultad y cada uno usa algo distinto de la unidad. Hazlos en tu archivo de práctica antes de mirar la solución: el esfuerzo de intentarlo es lo que construye la habilidad. Si te atorras más de 15 minutos, mira la solución, entiéndela línea por línea, ciérrala y vuelve a escribirla tú solo. Esa es la forma honesta de aprender.

Ejercicio 1: saludo personalizado con template literal.

Enunciado: Crea tres variables: tu nombre, tu ciudad y tu comida favorita. Usando UN template literal con interpolación, construye el texto: "Hola, soy [nombre], vivo en [ciudad] y mi comida favorita es [comida]." Además, en el mismo template literal, calcula cuántas letras tiene tu nombre usando una expresión dentro de `${}` y agrega al final: "Mi nombre tiene [n] letras." Muestra el resultado con `console.log`. No uses el operador `+` ni una sola vez.

Solución:

```
const nombre = "Carlos";
const ciudad = "Guadalajara";
const comida = "pozole";

const presentacion = `Hola, soy ${nombre}, vivo en ${ciudad} y mi comida favorita
es ${comida}. Mi nombre tiene ${nombre.length} letras.`;

console.log(presentacion);
// Hola, soy Carlos, vivo en Guadalajara y mi comida favorita es pozole. Mi nombre
tiene 6 letras.
```

Explicación paso a paso: Primero declaramos las tres variables con `const`, porque sus valores no van a cambiar. Luego construimos el template literal usando comillas invertidas, no comillas normales; esto es lo que permite la interpolación. Cada vez que necesitamos un valor dinámico, abrimos ``${`, ponemos la expresión y cerramos ```. Para el conteo de letras usamos `nombre.length`, que es una expresión válida dentro de ``${}`: JavaScript la evalúa y mete el

resultado (6) en el texto. Fíjate en que no hubo que partir el texto ni cazar espacios: los espacios alrededor de las variables se escriben naturalmente. Si hubieras usado concatenación, habrías necesitado seis o siete pedazos pegados con + y sería facilísimo olvidar un espacio. El error típico aquí es usar comillas dobles por costumbre y que el texto salga con el `${nombre}` sin interpolar; si te pasa, revisa las comillas exteriores.

Ejercicio 2: selector detective.

Enunciado: Dado este HTML, escribe el JavaScript para: a) seleccionar el h1 por su id y mostrar su `textContent` en consola; b) seleccionar el PRIMER elemento con clase "aviso" usando `querySelector`; c) seleccionar TODOS los elementos con clase "aviso" usando `querySelectorAll` y mostrar cuántos hay; d) convertir esa `NodeList` en un arreglo real y crear un arreglo solo con los textos de cada aviso usando `map`.

HTML:

```
<h1 id="titulo-noticias">Noticias de hoy</h1>
<p class="aviso">Hay partido a las 8pm</p>
<p class="aviso">Mañana no hay clases</p>
<p class="aviso">Nueva cafetería abierta</p>
```

Solución:

```
// a) Por id: directo al elemento único
const titulo = document.getElementById("titulo-noticias");
console.log(titulo.textContent); // "Noticias de hoy"

// b) querySelector: el PRIMERO que coincida
const primerAviso = document.querySelector(".aviso");
console.log(primerAviso.textContent); // "Hay partido a las 8pm"

// c) querySelectorAll: TODOS los que coincidan (NodeList)
const avisos = document.querySelectorAll(".aviso");
console.log(avisos.length); // 3

// d) Convertir a arreglo real y usar map
const textos = [...avisos].map(aviso => aviso.textContent);
console.log(textos);
// ["Hay partido a las 8pm", "Mañana no hay clases", "Nueva cafetería abierta"]
```

Explicación paso a paso: En el inciso a usamos `getElementById` porque el h1 tiene un id único; le pasamos el id SIN el símbolo # (el # solo se usa en selectores CSS). En el inciso b,

`querySelector(".aviso")` usa sintaxis CSS, por eso el punto antes de "aviso": en CSS, `.aviso` significa "clase aviso". Devuelve solo el primero aunque haya tres. En el inciso c, `querySelectorAll` devuelve una `NodeList` con los tres párrafos; `.length` nos dice cuántos hay. En el inciso d está el truco importante: la `NodeList` no tiene `map`, así que la convertimos con el `spread [...aviso]` en un arreglo de verdad, y sobre ese arreglo sí podemos usar `map` para extraer los textos. El error típico es intentar `aviso.map` directamente y obtener "`aviso.map is not a function`". Otro error típico es escribir `querySelector("#aviso")` con `#` cuando "aviso" es una clase, no un id: el selector no encontraría nada y devolvería `null`.

Ejercicio 3: `textContent` seguro contra `innerHTML` peligroso.

Enunciado: Simula un libro de visitas. Crea en el HTML un `<div id="muro"></div>`. En JavaScript, crea una función `publicarMensaje(nombre, mensaje)` que agregue al muro un párrafo con el formato "nombre dice: mensaje". La función debe ser SEGURA: si el mensaje contiene HTML (por ejemplo "`<b>hola</b>`" o algo malicioso), debe mostrarse como texto plano, sin interpretarse. Implementa la función usando `createElement` y `textContent` (no `innerHTML`). Pruébala con un mensaje normal y con un mensaje que contenga "`<img src=x onerror=alert(1)>`".

Solución:

```
<!-- HTML -->
<div id="muro"></div>
```

```
function publicarMensaje(nombre, mensaje) {
  // Creamos el párrafo desde cero
  const parrafo = document.createElement("p");

  // textContent trata TODO como texto plano: nada se interpreta como HTML
  parrafo.textContent = nombre + " dice: " + mensaje;

  // Lo pegamos al muro
  document.getElementById("muro").appendChild(parrafo);
}

// Prueba 1: mensaje normal
publicarMensaje("Ana", "¡Qué buena página!");

// Prueba 2: mensaje con HTML (se muestra como texto, no se ejecuta nada)
publicarMensaje("Travieso", "<b>hola</b>");

// Prueba 3: intento de ataque (se muestra como texto inofensivo)
publicarMensaje("Atacante", '<img src=x onerror=alert("hackeado")>');
```

Explicación paso a paso: La función recibe nombre y mensaje como parámetros. Dentro, creamos un elemento `<p>` vacío con `document.createElement("p")`. La línea clave es `parrafo.textContent = ...`: al asignar el texto con `textContent`, el navegador JAMÁS lo interpreta como HTML, así que las etiquetas que escriba el usuario aparecen como texto visible pero inerte. Luego pegamos el párrafo al muro con `appendChild`. En la prueba 2, el usuario verá literalmente los caracteres `<b>hola</b>`, sin negritas: eso es lo correcto y lo seguro. En la prueba 3, el intento de inyectar una imagen con código malicioso también se muestra como texto plano y el `alert` nunca se ejecuta. Si hubiéramos usado `innerHTML`, la prueba 3 habría ejecutado el código del atacante. Este ejercicio es la lección de seguridad más importante de la unidad: los datos que vienen de usuarios siempre se muestran con `textContent` o se sanitizan antes de usar `innerHTML`. En una entrevista de trabajo, explicar este ejercicio te pone por encima del 80% de los candidatos.

Ejercicio 4: semáforo de contraseña con `classList`.

Enunciado: Crea un `<input id="clave" type="password">` y un `<p id="medidor"></p>`. Define en CSS tres clases: `.debil` (rojo), `.media` (naranja) y `.fuerte` (verde). Escribe una función `evaluarClave()` que lea el valor del input con `.value`, y según la longitud asigne la clase correspondiente al medidor y escriba en él el texto "Débil", "Media" o "Fuerte": menos de 6 caracteres es débil, de 6 a 9 es media, 10 o más es fuerte. La función debe LIMPIAR las clases anteriores antes de poner la nueva. Llama a la función manualmente con distintos valores para probarla (los eventos para llamarla automáticamente llegan en la Unidad 8).

Solución:

```
<!-- HTML -->
<style>
  .debil { color: white; background-color: red; padding: 5px; }
  .media { color: white; background-color: orange; padding: 5px; }
  .fuerte { color: white; background-color: green; padding: 5px; }
</style>
<input id="clave" type="password" placeholder="Escribe una contraseña">
<p id="medidor"></p>
```



```

function evaluarClave() {
  const campo = document.getElementById("clave");
  const medidor = document.getElementById("medidor");

  // OJO: los inputs se leen con .value, NO con textContent
  const texto = campo.value;
  const longitud = texto.length;

  // Limpiamos el estado anterior (si no, se acumulan las clases)
  medidor.classList.remove("debil", "media", "fuerte");

  if (longitud < 6) {
    medidor.classList.add("debil");
    medidor.textContent = "Débil: usa al menos 6 caracteres";
  } else if (longitud <= 9) {
    medidor.classList.add("media");
    medidor.textContent = "Media: vas bien, intenta 10 o más";
  } else {
    medidor.classList.add("fuerte");
    medidor.textContent = "Fuerte: buena contraseña";
  }
}

// Pruebas manuales (en la Unidad 8 esto se disparará solo al escribir)
document.getElementById("clave").value = "abc";
evaluarClave(); // medidor en rojo: "Débil..."

document.getElementById("clave").value = "abc12345";
evaluarClave(); // medidor en naranja: "Media..."

document.getElementById("clave").value = "abc123XYZ!";
evaluarClave(); // medidor en verde: "Fuerte..."

```

Explicación paso a paso: Primero seleccionamos el campo y el medidor y los guardamos en constantes, porque los usamos varias veces. Leemos lo escrito con `campo.value`, la propiedad correcta para inputs (`textContent` aquí no serviría). Medimos la longitud del texto. El paso que muchos olvidan: `medidor.classList.remove("debil", "media", "fuerte")` limpia cualquier clase anterior; sin esto, si primero evalúas "abc" (débil) y luego "abc123XYZ!" (fuerte), el medidor tendría las clases "debil" y "fuerte" a la vez y el color final dependería del orden en el CSS, un bug visual confuso. Luego el `if/else if/else` asigna la clase y el texto correspondientes. Observa que el diseño (colores) vive en el CSS y la lógica (cuándo aplicar cada clase) vive en JavaScript: separación de responsabilidades, la marca del código profesional. En la Unidad 8

conectarás esta función al evento de "escribir" para que se evalúe en tiempo real, como los medidores de los sitios de verdad.

Ejercicio 5: catálogo de productos con map + join (el patrón profesional).

Enunciado: Tienes este arreglo de productos. Genera con JavaScript el HTML de una tarjeta por producto dentro del `<div id="catalogo"></div>`. Cada tarjeta debe mostrar el nombre, el precio con formato "\$X pesos" y, SOLO si el producto está disponible, una etiqueta "En stock"; si no está disponible, la tarjeta debe llevar la clase "agotado". Usa UN template literal por tarjeta, construye todo el HTML con map + join (sin ciclos for), y asígnalo con UNA sola asignación a innerHTML. Los datos son de tu código (confiables), así que innerHTML es apropiado aquí.

```
const productos = [
  { nombre: "Playera colibrí", precio: 299, disponible: true },
  { nombre: "Taza programador", precio: 149, disponible: false },
  { nombre: "Stickers pack x10", precio: 99, disponible: true }
];
```

Solución:

```
<!-- HTML -->
<div id="catalogo"></div>
```

```

const productos = [
  { nombre: "Playera colibrí", precio: 299, disponible: true },
  { nombre: "Taza programador", precio: 149, disponible: false },
  { nombre: "Stickers pack x10", precio: 99, disponible: true }
];

// Paso 1: cada producto se convierte en su tarjeta HTML (map)
const tarjetas = productos.map(producto => `
  <div class="tarjeta ${producto.disponible ? "" : "agotado"}">
    <h3>${producto.nombre}</h3>
    <p class="precio">${producto.precio} pesos</p>
    ${producto.disponible ? '<p class="stock">En stock</p>' : '<p class="stock no">Agotado</p>'}
  </div>
`);

// Paso 2: pegamos todas las tarjetas en un solo texto (join sin separador)
const htmlCatalogo = tarjetas.join("");

// Paso 3: UNA sola inserción en el DOM
document.getElementById("catalogo").innerHTML = htmlCatalogo;

```

Explicación paso a paso: El map recorre el arreglo y por cada producto devuelve un template literal con su tarjeta. Dentro del template hay dos ternarios: uno decide si la tarjeta lleva la clase "agotado" (fíjate en cómo interpolamos una clase CSS condicional: `class="tarjeta ${...}"`), y otro decide qué etiqueta de stock mostrar. El map produce un arreglo de tres textos HTML. El `join("")` los pega en un solo texto largo sin nada entre ellos (las comillas vacías son el pegamento invisible; si omitieras el join, JavaScript uniría los textos con comas). Finalmente, una sola asignación a `innerHTML` inserta todo el catálogo de golpe: eficiente y limpio. Este es exactamente el patrón que usan las tiendas en línea reales para pintar sus catálogos desde datos. Si dominas este ejercicio, dominas el 80% de la generación dinámica de contenido. El error típico es olvidar el join y ver comas misteriosas entre las tarjetas; el otro error típico es hacer `innerHTML +=` dentro de un for, reconstruyendo el DOM en cada vuelta.

Ejercicio 6: galería con atributos data (reto integrador).

Enunciado: Crea una galería con un `<img id="foto-grande">` y un `<p id="pie-foto"></p>`. Debajo, en tu JavaScript, define un arreglo de 3 objetos con `{src, titulo, autor}`. Genera tres botones (uno por foto) usando `createElement` en un ciclo: cada botón debe mostrar el título de la foto y guardar en atributos data- el src y el autor (`data-src` y `data-autor`). Al final, escribe una función `mostrarFoto(indice)` que lea los data- del botón correspondiente y actualice la imagen

grande (src), el pie de foto (textContent con "titulo -- foto de autor") usando un template literal. Pruébala llamándola con 0, 1 y 2. (En la Unidad 8 harás que los botones respondan a clics; por ahora la llamada manual está bien.)

Solución:

```
<!-- HTML -->
<img id="foto-grande" src="" alt="Galería">
<p id="pie-foto"></p>
<div id="botones"></div>
```

```

const fotos = [
  { src: "playa.jpg", titulo: "Atardecer en la playa", autor: "Ana" },
  { src: "montana.jpg", titulo: "Cima nevada", autor: "Luis" },
  { src: "ciudad.jpg", titulo: "Luces de la ciudad", autor: "María" }
];

const contenedorBotones = document.getElementById("botones");

// Generamos un botón por foto, guardando los datos en atributos data-
fotos.forEach((foto, indice) => {
  const boton = document.createElement("button");
  boton.textContent = foto.titulo;
  boton.dataset.src = foto.src;      // se guarda como data-src="playa.jpg"
  boton.dataset.autor = foto.autor;  // se guarda como data-autor="Ana"
  boton.dataset.indice = indice;     // guardamos también su posición
  contenedorBotones.appendChild(boton);
});

function mostrarFoto(indice) {
  const botones = document.querySelectorAll("#botones button");
  const boton = botones[indice];

  // Leemos los datos pegados al botón
  const src = boton.dataset.src;
  const autor = boton.dataset.autor;

  // Actualizamos la imagen grande y el pie con un template literal
  const fotoGrande = document.getElementById("foto-grande");
  fotoGrande.src = src;
  fotoGrande.alt = boton.textContent;

  document.getElementById("pie-foto").textContent =
    `${boton.textContent} -- foto de ${autor}`;
}

// Probamos las tres fotos
mostrarFoto(0); // Atardecer en la playa -- foto de Ana
mostrarFoto(1); // Cima nevada -- foto de Luis
mostrarFoto(2); // Luces de la ciudad -- foto de María

```

Explicación paso a paso: Primero definimos los datos como un arreglo de objetos, separando los datos de la presentación (buena práctica: si mañana cambian las fotos, solo tocas el arreglo). Luego generamos los botones dinámicamente con `forEach` y `createElement`: cada

botón lleva su título como texto visible y guarda src, autor e índice en atributos data-, como etiquetas de equipaje pegadas a cada botón. La función mostrarFoto recibe un índice, recupera el botón correspondiente de la NodeList, lee sus data- con dataset, y actualiza la imagen grande (propiedad src) y el pie de foto (textContent con un template literal que combina título y autor). Este ejercicio integra todo: creación dinámica, data attributes, selectores, modificación de atributos y template literals. Es el antecesor directo de una galería real con clics, que construirás en la Unidad 8 cuando aprendas eventos. El detalle a cuidar: dataset convierte data-src en dataset.src automáticamente (quita el "data-" y el guion se vuelve mayúscula: data-fecha-nac se leería como dataset.fechaNac).

Ejercicio 7: modo oscuro con toggle (el clásico que todo sitio tiene).

Enunciado: Crea una página con un <body> que contenga un <h1> y un <p>. Define en CSS una clase.oscuro aplicada al body que cambie el fondo a casi negro y el texto a blanco. Escribe una función cambiarTema() que use classList.toggle("oscuro") sobre document.body y que además actualice el texto de un <p id="estado-tema"> con un template literal que diga "Tema actual: oscuro" o "Tema actual: claro" según corresponda (usa classList.contains para averiguarlo). Llama a la función dos veces para verificar que prende y apaga.

Solución:

```
<!-- HTML -->
<style>
  body { background-color: white; color: black; transition: all 0.3s; }
  body.oscuro { background-color: #121212; color: #f0f0f0; }
</style>
<h1>Mi página</h1>
<p>Este texto cambia con el tema.</p>
<p id="estado-tema"></p>
```

```
function cambiarTema() {
  // toggle prende o apaga la clase, y devuelve true si la dejó prendida
  const ahoraOscuro = document.body.classList.toggle("oscuro");

  // contains nos dice el estado actual para mostrarlo en el texto
  const tema = document.body.classList.contains("oscuro") ? "oscuro" : "claro";
  document.getElementById("estado-tema").textContent = `Tema actual: ${tema}`;

  return ahoraOscuro;
}

cambiarTema(); // prende el oscuro, el texto dice "Tema actual: oscuro"
cambiarTema(); // lo apaga, el texto dice "Tema actual: claro"
```

Explicación paso a paso: El CSS define cómo se ve la página normal y cómo se ve con la clase `oscuro` en el `body`; observa el selector `body.oscuro`, que significa "el `body` cuando tiene la clase `oscuro`". La función usa `classList.toggle("oscuro")` sobre `document.body`: la primera vez agrega la clase (modo oscuro prendido), la segunda la quita (apagado). El valor que devuelve `toggle` (`true/false`) nos dice el estado resultante, aunque aquí preferimos verificarlo con `classList.contains` para que el ejemplo te enseñe ambos métodos. El texto de estado se construye con un template literal y un ternario. Este patrón (`toggle` + `contains`) es el que usan los botones de modo oscuro de los sitios reales; en la Unidad 8 lo conectarás a un clic y tendrás un modo oscuro funcional de verdad. Nota el `transition` en el CSS: es un detalle de pulido que hace que el cambio de tema se vea suave en lugar de brusco; los detalles visuales también son parte del oficio.

Mini-retos para cuando termines (sin solución, para que investigues): primero, modifica el ejercicio 5 para que los productos agotados no muestren el precio sino el texto "Próximamente"; segundo, en el ejercicio 6, agrega una cuarta foto sin tocar el HTML, solo el arreglo; tercero, investiga qué hace `insertAdjacentHTML` y reescribe el ejercicio 5 usándolo en lugar de `innerHTML`, luego decide cuál te parece más legible. Si logras los tres, el DOM básico ya es tuyo.

Resumen de la unidad

Los template literals usan comillas invertidas y permiten interpolar cualquier expresión con `${}`, además de escribir textos multilínea sin trucos. Se leen como el resultado final, reducen errores de concatenación y son la herramienta estándar para construir HTML dinámico.

El DOM es la representación en memoria de tu página como un árbol de nodos: el HTML es la receta en papel, el DOM es el platillo servido. JavaScript modifica el DOM, no tu archivo, así que los cambios se pierden al recargar. Todo empieza con el objeto global `document`.

Selecciona elementos con `getElementById` para ids únicos (rápido y directo), `querySelector` para el primer elemento que coincida con un selector CSS, y `querySelectorAll` para todos los que coincidan (devuelve una `NodeList`: tiene `forEach` pero no `map`; conviértela con `[...lista]` si necesitas métodos de arreglo).

Tu script debe ejecutarse cuando los elementos ya existan: ponlo antes del cierre del `body`, o usa `defer`, o espera a `DOMContentLoaded`. Si un selector devuelve `null`, el error real aparecerá después como "Cannot read properties of null".

`textContent` lee y escribe texto plano (seguro contra XSS); `innerHTML` interpreta el texto como HTML (potente pero peligroso con datos de usuarios). Para inputs usa `value`, no `textContent`. Cambia atributos con propiedades directas (`src`, `href`) o `setAttribute`, y usa atributos `data-` con `dataset` para guardar datos extra en el HTML.

Para estilos, prefiere `classList` (`add`, `remove`, `toggle`, `contains`) con clases definidas en tu CSS, en lugar de asignar estilos en línea uno por uno. Limpia las clases anteriores antes de aplicar un nuevo estado.

Genera contenido dinámico acumulando HTML en un template literal y asignándolo una sola vez a `innerHTML`, o crea elementos individuales con `createElement` + `appendChild` + `textContent` cuando trabajes con datos de usuarios. El patrón `map` + `join` es la forma estándar de convertir un arreglo de datos en HTML.

Con esto ya puedes leer y transformar cualquier página. En la Unidad 8 vas a hacer que la página reaccione al usuario: clics, formularios y temporizadores, contruidos sobre todo lo que aprendiste aquí.

UNIDAD 8: Interactividad en la página

En la unidad anterior aprendiste a hablarle a la página: a construir textos dinámicos, a encontrar elementos en el árbol del DOM y a transformarlos. Pero había una pieza faltante, y seguro la notaste: todo lo que hacías ocurría cuando el script se ejecutaba, de forma automática, sin que el usuario interviniera. Era como un actor que se presenta en un teatro vacío: gran actuación, cero público. Esta unidad llena el teatro y le da al público un papel en la obra. Vas a aprender los eventos: la forma en que JavaScript escucha lo que el usuario hace (clics, teclas, movimientos) y responde en consecuencia. Con eventos, tu página deja de ser una presentación y se convierte en una conversación.

Piensa en la diferencia con un ejemplo cotidiano. Una página sin eventos es como un folleto impreso de un restaurante: bonito, informativo, pero mudo. No importa cuántas veces lo toques, siempre dice lo mismo. Una página con eventos es como un mesero atento: te ve llegar y te saluda, cuando levantas la mano viene a tu mesa, cuando pides la cuenta la trae sin que tengas que gritar. Los eventos son los sentidos de tu página: sin ellos, tu JavaScript es sordo y ciego a lo que el usuario hace; con ellos, cada clic, cada tecla y cada movimiento del mouse se convierte en una oportunidad de responder.

¿Qué vas a aprender exactamente? Tres bloques que se encadenan como en la unidad anterior. Primero, los eventos: qué son, cómo se "escuchan" con `addEventListener`, el evento click que es el rey de la web, y el objeto event, ese mensajero que trae información valiosa sobre lo que pasó (dónde se hizo clic, qué tecla se presionó, cuál elemento disparó el evento). Segundo, los formularios: cómo leer lo que el usuario escribe con `input.value`, cómo capturar el envío con el evento submit, y por qué `preventDefault` es una de las llamadas más importantes que vas a aprender (sin ella, el formulario se comporta como en 1998 y recarga la página). Tercero, el tiempo: `setTimeout` para hacer algo una vez después de una espera, `setInterval` para repetir algo cada cierto tiempo, cómo medir todo en milisegundos sin equivocarte, y cómo detener los temporizadores con `clearTimeout` y `clearInterval`. Cerraremos con mini proyectos que unen todo: un reloj digital en vivo y una cuenta regresiva, los clásicos que todo programador web construye alguna vez.

¿Por qué importa esto? Porque la interactividad ES la web moderna. Haz la prueba mental: abre tu red social favorita y trata de encontrar algo que NO sea un evento. El botón de "me gusta": evento click. El campo de búsqueda que sugiere mientras escribes: evento de teclado. El menú que se abre al pasar el mouse: evento de mouse. El formulario de comentarios: evento submit. Las notificaciones que llegan solas: temporizadores o eventos del servidor. Incluso el scroll infinito que carga más contenido cuando llegas al final: evento scroll. Si el

DOM de la Unidad 7 era el cuerpo de la página, los eventos son su sistema nervioso: llevan las señales del mundo exterior (el usuario) al cerebro (tu JavaScript) y devuelven respuestas a los músculos (el DOM).

Hay una idea profunda aquí que quiero que captures desde el principio: la programación con eventos invierte la forma en que piensas. Hasta ahora tus programas eran secuenciales: línea 1, línea 2, línea 3, de arriba hacia abajo, como leer un libro. Con eventos, tu programa se vuelve reactivo: tú no decides cuándo se ejecuta cada parte, el usuario lo decide. Tú solo preparas las respuestas por adelantado ("cuando hagan clic aquí, haz esto; cuando envíen el formulario, haz lo otro") y luego te sientas a esperar. Es como la diferencia entre dar un discurso (tú controlas el orden) y atender una taquería (los clientes llegan cuando quieren y tú respondes a cada pedido). Este cambio de mentalidad, de "yo controlo el flujo" a "yo preparo respuestas", es el salto más importante que vas a dar como programador en este curso, porque así funcionan las aplicaciones reales: nadie usa una app de principio a fin en orden, la gente toca lo que quiere cuando quiere.

Déjame contarte una historia que ilustra el poder y el peligro de los eventos. Un conocido hizo una página para vender boletos de un concierto. Todo funcionaba: elegías tus asientos, llenabas tus datos, hacías clic en "comprar". El problema era que no había deshabilitado el botón después del primer clic, y la confirmación tardaba unos 5 segundos en llegar. La gente, impaciente, hacía clic dos, tres, cuatro veces en "comprar"... y el sistema les vendía los boletos cuatro veces. Cuatro cargos a la tarjeta, cuatro correos de confirmación, un caos de devoluciones. La solución fue de tres líneas: al primer clic, deshabilitar el botón y cambiar su texto a "Procesando...". Tres líneas que le habrían ahorrado miles de pesos en devoluciones y un fin de semana de pesadilla. Esa es la lección: los eventos te dan poder sobre cada interacción, y con ese poder viene la responsabilidad de pensar en lo que el usuario impaciente, distraído o travieso podría hacer.

Otro punto que debes entender desde ya: los eventos y los temporizadores comparten una idea central de JavaScript que te va a acompañar toda la carrera: las funciones como valores que se pasan a otros lados. Cuando escribes `boton.addEventListener("click", saludar)`, le estás entregando la función `saludar` al navegador y diciéndole "guárdala y ejecútala cuando alguien haga clic". No la estás ejecutando tú (fíjate: es `saludar`, sin paréntesis). Este concepto, pasar funciones como argumentos para que se ejecuten después, se llama *callback*, y ya lo viste en la Unidad 6 con `forEach` y `map`. Los eventos lo llevan al siguiente nivel: el "después" ya no es "después de recorrer el arreglo" sino "después de que pase algo en el mundo real, en un momento que nadie puede predecir". Si los *callbacks* de la Unidad 6 te costaron trabajo, esta unidad te los va a terminar de aclarar a la fuerza, porque aquí no hay forma de evitarlos.

Hablemos del panorama de práctica. Para esta unidad necesitas, igual que en la anterior, un archivo HTML real con tu script al final del body. Pero ahora además necesitas interactuar físicamente: hacer clic en botones, escribir en campos, enviar formularios. No basta con leer el código; tienes que probarlo con tus manos, porque los eventos son por definición algo que se siente. Te recomiendo que por cada ejemplo hagas el ritual completo: escribe el HTML, escribe el JavaScript, abre la página, haz clic, escribe, observa. Y luego rompe cosas a propósito: ¿qué pasa si haces doble clic rápido? ¿Qué pasa si envías el formulario vacío? ¿Qué pasa si pones un intervalo de 0 milisegundos? Romper cosas a propósito es la forma más rápida de entender los límites.

Una advertencia sobre los temporizadores antes de empezar: `setInterval` es una de las herramientas más útiles y más peligrosas de JavaScript. Útil porque te permite crear relojes, animaciones, actualizaciones automáticas. Peligrosa porque un intervalo olvidado sigue ejecutándose para siempre, consumiendo memoria y procesador, incluso cuando ya no lo necesitas. Hay una regla de oro que veremos a fondo: todo `setInterval` que creas debe tener un `clearInterval` que lo detenga en algún momento, igual que toda puerta que abres debes cerrarla. Los programadores profesionales han aprendido esto a golpes, con páginas que se volvían lentas misteriosamente hasta descubrir un intervalo zombie corriendo en segundo plano desde hacía una hora.

También quiero prepararte para una confusión típica de esta unidad: la diferencia entre "asignar un manejador" y "ejecutar el manejador". Cuando escribes `addEventListener("click", mostrarMensaje)`, NO se ejecuta `mostrarMensaje` en ese momento; solo la registras. Es como poner una trampa para ratones: la pones hoy, pero se activa cuando el ratón llegue, que puede ser en una hora o nunca. Muchos principiantes escriben `addEventListener("click", mostrarMensaje())` con paréntesis, y entonces la función se ejecuta INMEDIATAMENTE (una sola vez, al cargar la página) y el evento queda registrado con el resultado de esa ejecución (usualmente `undefined`), así que cuando el usuario hace clic no pasa nada. Este error del paréntesis es probablemente el error número uno de esta unidad. Grábatelo desde ahora: al registrar, la función va SIN paréntesis; los paréntesis son para cuando TÚ quieres ejecutarla en ese momento.

El mapa de la unidad, para que lo tengas claro: primero dominarás `addEventListener` y el evento click con el objeto event (clases 22). Después conquistarás los formularios: leer valores, validar, interceptar el submit con `preventDefault` (clase 23). Luego dominarás el tiempo con `setTimeout` y `setInterval`, aprendiendo a pensar en milisegundos y a limpiar tus temporizadores (clase 24). Y al final, los mini proyectos van a unir las tres cosas: un reloj que se actualiza solo (intervalo + DOM) y una cuenta regresiva con formulario (eventos +

temporizadores + DOM). Cuando termines, habrás construido piezas que aparecen en sitios reales todos los días.

Y una reflexión final para esta introducción: si la Unidad 7 te dio ojos para ver el DOM en cada página, esta unidad te va a dar oídos para escuchar los eventos. Vas a empezar a notar cosas que antes eran invisibles: "este botón se deshabilita mientras carga, eso es un evento con cambio de estado", "este formulario valida mientras escribo, eso es un evento de teclado con preventDefault implícito", "este carrusel cambia solo cada 5 segundos, eso es un setInterval". Esa es la mirada completa del desarrollador web: ver la estructura (DOM) y escuchar el comportamiento (eventos). Con las dos unidades dominadas, ya puedes construir páginas que se sienten vivas, y eso, colega, es el 80% de lo que hace un desarrollador frontend junior en su trabajo diario. Vamos a darle.

Un último apunte antes de la explicación detallada, sobre cómo estudiar esta unidad. En las unidades anteriores podías avanzar concepto por concepto de forma bastante lineal. Aquí los tres bloques (eventos, formularios, temporizadores) se entrelazan constantemente en los ejemplos reales: un formulario usa eventos, un temporizador actualiza el DOM, un evento inicia un temporizador. Te sugiero que, mientras lees la explicación de cada bloque, vayas anotando en un papel (o en un archivo) las preguntas de conexión que se te ocurran: "¿puedo iniciar un setInterval dentro de un click?", "¿qué pasa si el usuario envía el formulario dos veces rápido?". Esas preguntas son oro, porque son exactamente las que aparecen en el trabajo real, y la sección de ejemplos y práctica está diseñada para responderlas. No las ignores; persíguelas.

Explicación detallada

Empecemos por la pregunta más básica: ¿qué es un evento? Un evento es un aviso de que algo pasó. El navegador está constantemente vigilando: ¿el usuario hizo clic? ¿presionó una tecla? ¿movió el mouse? ¿se terminó de cargar la página? ¿se envió un formulario? Cada una de esas cosas genera un evento, que es básicamente un objeto con información sobre lo que ocurrió. Tu trabajo como programador no es generar los eventos (eso lo hace el navegador), sino escucharlos y responder. Es como ser el recepcionista de un hotel: tú no decides cuándo llegan los huéspedes, pero cuando la campanita suena (el evento), tú actúas (el manejador).

La herramienta fundamental para escuchar es `addEventListener`. Se usa así: `elemento.addEventListener("click", miFuncion)`. Tiene tres partes: el elemento que vas a vigilar (cualquiera del DOM), el nombre del evento como texto ("click", "submit", "keydown", etc.), y la función que se ejecutará cuando el evento ocurra, llamada manejador o handler. Fíjate en que

el nombre del evento va sin el prefijo "on": es "click", no "onclick". (Existe una forma vieja usando propiedades como `boton.onclick = miFuncion`, que todavía funciona pero solo permite UN manejador por evento y por eso los profesionales prefieren `addEventListener`, que permite registrar todos los manejadores que quieras sobre el mismo evento.)

Hay un detalle de diseño hermoso en `addEventListener` que vale la pena apreciar: separa el QUÉ del CUÁNDO. El "qué" es tu función (mostrar un mensaje, validar un campo); el "cuándo" es el evento (al hacer clic, al enviar). Esta separación es lo que hace que el código con eventos sea mantenible: puedes cambiar la respuesta sin tocar la escucha, y viceversa. Compáralo con la forma antigua de poner `onclick="hacerAlgo()"` directamente en el HTML: ahí el qué y el cuándo están pegados en el mismo lugar, y si tienes 50 botones tienes 50 pedazos de JavaScript regados en tu HTML. Con `addEventListener`, todo tu JavaScript vive en tu `archivo.js`, organizado y buscable. Esta es una de esas buenas prácticas que no se negocian en código profesional.

El evento click es el rey porque el clic es la interacción fundamental de la web: botones, enlaces, tarjetas, íconos, todo se activa con clics. Pero el sistema de eventos es mucho más rico. Estos son los eventos que vas a usar el 95% del tiempo: `click` (clic del mouse o toque en pantalla), `dblclick` (doble clic), `submit` (envío de formulario), `input` (cada vez que cambia el valor de un campo, tecla por tecla), `change` (cuando un campo pierde el foco habiendo cambiado, o cuando cambia un select o checkbox), `keydown` / `keyup` (tecla presionada / tecla soltada), `mouseover` / `mouseout` (el mouse entra / sale de un elemento), `focus` / `blur` (un campo gana / pierde el foco), y `DOMContentLoaded` (el documento terminó de cargarse). No necesitas memorizarlos todos hoy; con que sepas que existen y dónde buscarlos es suficiente. Lo importante es el patrón: todos se escuchan igual, con `addEventListener`.

Ahora, el objeto event, el mensajero. Cuando tu manejador se ejecuta, el navegador le pasa automáticamente un objeto con información sobre el evento. Por convención se llama `event` o simplemente `e`. Este objeto es una mina de oro: `event.target` te dice QUÉ elemento disparó el evento (utilísimo cuando un mismo manejador atiende varios elementos), `event.type` te dice qué tipo de evento fue, `event.clientX` y `event.clientY` te dicen las coordenadas del mouse al hacer clic, `event.key` te dice qué tecla se presionó en eventos de teclado. Mira este ejemplo mental: tienes una lista de 10 botones de productos y un solo manejador; con `event.target` sabes exactamente en cuál botón hizo clic el usuario, sin necesidad de 10 manejadores distintos. Esa técnica tiene nombre: delegación de eventos, y aunque la veremos más a fondo en tu futuro, la semilla queda plantada aquí.

Un error clásico con el objeto event: olvidar declararlo como parámetro. Si tu función es `function saludar()` sin parámetros y dentro intentas usar `event.target`, obtendrás un error (o

peor, en algunos navegadores viejos funcionaba por una variable global event, lo que hacía el bug intermitente y enloquecedor). La regla: si vas a usar información del evento, tu manejador DEBE recibir el parámetro: `boton.addEventListener("click", function(event) {...})` o con función flecha: `boton.addEventListener("click", (e) => {...})`. Acostúmbrate a ponerlo siempre, aunque no lo uses todavía; no cuesta nada y te salva cuando lo necesites.

Hablemos de la propagación de eventos, un concepto que confunde a todos la primera vez pero que es esencial para no volverse loco depurando. Cuando haces clic en un botón que está dentro de un div que está dentro del body, el clic no solo le llega al botón: viaja. Primero baja desde el document hasta el botón (fase de captura), luego sube de regreso desde el botón hasta el document (fase de burbuja). Por defecto, tus manejadores se ejecutan en la fase de burbuja, o sea, de adentro hacia afuera: primero el manejador del botón, luego el del div, luego el del body. Esto significa que un clic en un botón TAMBIÉN dispara los manejadores de clic de todos sus ancestros. ¿Es un bug? No, es el diseño, y es útil (es la base de la delegación de eventos). Pero si no lo sabes, te vuelves loco: "¿por qué se ejecuta dos veces mi código?" Porque tienes un manejador en el botón y otro en su contenedor, y ambos responden al mismo clic. Si alguna vez necesitas detener esa subida, existe `event.stopPropagation()`, que le dice al evento "quédate aquí, no sigas subiendo". Úsalo con moderación y con conciencia, porque detener la propagación también puede romper otros manejadores que sí esperaban enterarse.

Pasemos a los formularios, el segundo bloque. Los formularios son la forma principal en que los usuarios le entregan datos a tu página: registros, inicios de sesión, búsquedas, comentarios, compras. Y aquí hay una verdad incómoda: el comportamiento POR DEFECTO de un formulario es del siglo pasado. Cuando el usuario hace clic en un botón de tipo submit (o presiona Enter dentro del campo), el navegador recoge los datos, los empaqueta y recarga la página enviándolos al servidor. En 1998 eso estaba bien. Hoy, si tu página se recarga cada vez que el usuario envía un formulario, la experiencia se siente rota: se pierde el estado, parpadea la pantalla, y cualquier validación bonita que hiciste con JavaScript se va al caño.

La solución es interceptar el envío con el evento submit y llamar a `event.preventDefault()`. Esa llamada le dice al navegador: "gracias, pero NO hagas lo que ibas a hacer por defecto; yo me encargo". Es como quitarle el volante al piloto automático: a partir de ese momento, tú decides qué pasa con los datos (validarlos, mostrarlos, enviarlos con fetch como verás en unidades futuras, o simplemente usarlos en la página). El patrón completo de un formulario moderno es siempre el mismo: escuchas submit en el `<form>` (no click en el botón, porque el usuario también puede enviar con Enter), llamas `preventDefault()` como primera línea, lees los valores con `input.value`, los validas, y respondes. Memoriza ese patrón; lo vas a escribir cientos de veces.

¿Por qué escuchar submit en el formulario y no click en el botón? Por accesibilidad y por completitud. Muchos usuarios envían formularios presionando Enter mientras están en un campo, sin tocar el botón. Si solo escuchas el clic del botón, esos envíos se te escapan y el formulario se recarga sin tu validación. El evento submit captura TODAS las formas de envío: clic en el botón, Enter en un campo, e incluso envíos programáticos. Es el lugar correcto. Y el botón debe ser `<button type="submit">` o `<input type="submit">` para que el navegador sepa que ese botón envía el formulario.

Leer valores es sencillo pero tiene sus matices. Para `<input>` de texto, `<textarea>` y `<select>`, usas `elemento.value`, que siempre devuelve un TEXTO, incluso si el usuario escribió un número. Este es un caso borde clásico: lees la edad con `input.value` y obtienes "25" (texto), luego haces `edad + 5` y obtienes "255" en lugar de 30, porque el + con texto concatena. La solución es convertir: `Number(input.value)` o `parseInt(input.value)`. Para checkboxes usas `elemento.checked` (true/false, no value), y para grupos de radio buttons recorres las opciones buscando cuál tiene checked en true. Estos detalles de cada tipo de campo son de esas cosas que no se memorizan de un jalón: las aprendes usándolas y consultando cuando dudas. Lo importante es el principio: cada tipo de campo tiene su forma de leerse, y value es la puerta de entrada para los campos de texto.

La validación de formularios merece su propia reflexión porque es donde la interactividad se encuentra con la experiencia de usuario. Hay dos escuelas: validar al enviar (cuando el usuario hace submit, revisas todo y muestras los errores) o validar mientras escribe (con el evento input, das retroalimentación en tiempo real como el medidor de contraseña del ejercicio de la Unidad 7). La validación al enviar es más simple y es suficiente para la mayoría de los casos; la validación en tiempo real se siente más pulida pero requiere más cuidado (¿le muestras el error en rojo cuando apenas lleva 2 letras escritas? Eso se siente agresivo). Mi recomendación práctica: valida al enviar siempre, y agrega validación en tiempo real solo donde aporte mucho (contraseñas, disponibilidad de nombre de usuario). Y una regla de oro

de la validación: nunca confíes solo en la validación del navegador (JavaScript se puede desactivar o manipular); la validación real de seguridad siempre ocurre en el servidor. La validación con JavaScript es para la experiencia del usuario, no para la seguridad.

¿Sabías que? El HTML5 trae validación nativa gratis: si pones required en un campo, el navegador no deja enviar el formulario vacío; si el input es type="email", verifica que parezca un correo. Es útil como primera capa, pero tiene límites (los mensajes de error son genéricos y feos, y no puedes personalizar las reglas). En el desarrollo real se combinan: validación nativa como red de seguridad barata, y validación con JavaScript para la experiencia pulida. Y cuando quieras controlar TODO tú, puedes poner novalidate en el <form> para desactivar la validación nativa y hacerla 100% con tu código.

Ahora el tercer bloque: el tiempo. JavaScript tiene dos funciones para programar cosas en el futuro: setTimeout y setInterval. setTimeout(ejecutarEsto, milisegundos) ejecuta una función UNA VEZ después de esperar los milisegundos indicados. setInterval(ejecutarEsto, milisegundos) ejecuta una función REPETIDAMENTE, cada tantos milisegundos, hasta que la detengas. Ambas reciben la función SIN paréntesis (el mismo principio de los eventos: la registras, no la ejecutas) y devuelven un número identificador, un id de temporizador, que es tu "control remoto" para detenerlo después con clearTimeout(id) o clearInterval(id).

Los milisegundos son la unidad, y aquí hay una fuente inagotable de errores tontos: 1000 milisegundos = 1 segundo. Si quieres esperar 5 segundos, son 5000, no 5. Si pones setTimeout(hacer, 5), no esperas 5 segundos: esperas 5 milisegundos, o sea, prácticamente nada. Todo programador ha cometido este error al menos una vez y se ha quedado mirando su programa preguntándose por qué "no esperó". Mi truco: escribe siempre el cálculo explícito cuando no sean segundos cerrados. ¿2 minutos? $2 * 60 * 1000$. ¿Medio segundo? 500. Hacer la multiplicación visible en el código sirve como documentación y como verificación.

setTimeout tiene usos muy prácticos: mostrar un mensaje temporal ("¡Guardado!" que desaparece a los 3 segundos), retrasar una acción (esperar 1 segundo después de que el usuario deja de escribir antes de buscar, para no saturar), o encadenar pasos (primero muestra "cargando", después de 2 segundos muestra el resultado). Un patrón elegante es guardar el id: const temporizador = setTimeout(ocultarMensaje, 3000). Y si el usuario hace algo que cancela la necesidad (cierra el mensaje manualmente), llamas clearTimeout(temporizador) para que no se ejecute dos veces. Esto de "programar y posiblemente cancelar" es un patrón mental que aparece en todo el software: reservas una acción futura, pero te quedas con el boleto de cancelación por si acaso.

`setInterval` es el motor de todo lo que se actualiza solo: relojes digitales, cuentas regresivas, carruseles automáticos, barras de progreso, juegos simples. El patrón del reloj es el ejemplo canónico: cada 1000 milisegundos, lees la hora actual con `new Date()`, la formateas y la escribes en el DOM con `textContent`. Simple y poderoso. Pero aquí viene la advertencia seria: cada `setInterval` que no detienes sigue corriendo para siempre, incluso si el usuario ya no está viendo esa parte de la página. En aplicaciones grandes, los intervalos olvidados son una causa famosa de fugas de memoria y lentitud misteriosa. La disciplina es: cuando crees un intervalo, decide EN ESE MOMENTO cuándo se va a detener, y escribe el `clearInterval` correspondiente. Para un reloj que vive toda la sesión, está bien no detenerlo; para una cuenta regresiva, se detiene al llegar a cero; para una animación temporal, se detiene al terminar.

Un caso borde fascinante de `setInterval`: no garantiza puntualidad perfecta. Si tu función tarda más en ejecutarse que el intervalo, las ejecuciones se empiezan a encimar o a retrasar. Y si la pestaña del navegador está en segundo plano, el navegador reduce la frecuencia de los intervalos para ahorrar batería (los baja a 1 por segundo o menos). Para un reloj casero no importa; para un cronómetro de precisión sí. La técnica profesional para temporizadores precisos no es contar intervalos, sino comparar marcas de tiempo: guardas la hora de inicio con `Date.now()` y en cada tick calculas cuánto ha pasado realmente. Así, aunque el navegador retrase algunos ticks, tu cuenta sigue siendo exacta. Es un detalle avanzado, pero quiero que sepas que existe para cuando lo necesites.

Otra pregunta frecuente: ¿cuál es la diferencia entre `setTimeout` y `setInterval` si puedo simular uno con el otro? Buena observación: puedes simular un intervalo con timeouts encadenados (al final de la función, programas otro `setTimeout` que la vuelva a llamar). De hecho, esa técnica (llamada "timeout recursivo") tiene una ventaja: garantiza que la siguiente ejecución espera a que termine la anterior, evitando encimamientos. Los profesionales la prefieren para tareas que podrían tardar un tiempo variable. `setInterval` es más simple y directo para ritmos fijos y rápidos (como un reloj). Conoce ambas; usa la simple por defecto y la robusta cuando la tarea lo pida.

Cuidado con... un error sutil al pasar argumentos a temporizadores. `setTimeout(saludar, 1000)` está bien. Pero si tu función necesita argumentos, NO hagas `setTimeout(saludar("Ana"), 1000)`, porque eso ejecuta `saludar` inmediatamente (el problema del paréntesis otra vez). Las formas correctas: envolver en una función flecha, `setTimeout(() => saludar("Ana"), 1000)`, o pasar los argumentos después de los milisegundos: `setTimeout(saludar, 1000, "Ana")`. La forma de la flecha es la más legible y la que te recomiendo. Este mismo principio aplica a `addEventListener` cuando tu manejador necesita argumentos extra: `boton.addEventListener("click", () => saludar("Ana"))`. La función flecha como envoltorio es una

de las técnicas más usadas en JavaScript; si aún no te sale natural, esta unidad te la va a grabar a fuego.

Hablemos del "por qué" de los temporizadores en el diseño de JavaScript. JavaScript es de un solo hilo: hace una cosa a la vez. Entonces, ¿cómo puede "esperar" 3 segundos sin congelar la página? La respuesta es el modelo de concurrencia con cola de eventos: `setTimeout` no "duerme" al programa; le dice al navegador "en 3 segundos, pon esta función en la fila de pendientes", y mientras tanto JavaScript sigue haciendo otras cosas (respondiendo a clics, actualizando la página). Cuando pasan los 3 segundos y el hilo está libre, se ejecuta la función. Este modelo es la razón por la que una página puede tener un reloj actualizándose, un carrusel moviéndose y botones respondiendo, todo "a la vez", con un solo hilo. No necesitas entenderlo a fondo hoy, pero sí la idea clave: los temporizadores NO bloquean; programan. Si alguna vez necesitas una espera que SÍ bloquee (casi nunca la necesitas en frontend), esa es otra historia.

Para cerrar la explicación, el modelo mental unificado de la unidad: los eventos son el sistema de "cuando pase X, haz Y"; los formularios son el caso especial más importante de eventos (cuando el usuario entrega datos, intercepte, lea, valide y responda); los temporizadores son eventos del tiempo (cuando pasen N milisegundos, haz Y, una vez o repetidamente). Los tres usan el mismo mecanismo de fondo: registrar una función para que se ejecute después, sin paréntesis, con el navegador como intermediario. Y los tres se combinan con el DOM de la Unidad 7: el evento te dice cuándo, el temporizador te dice cada cuánto, y el DOM es lo que transformas en respuesta. Con ese modelo, vamos a los ejemplos, donde verás cada pieza en movimiento.

Ejemplos paso a paso

Mismo ritual que en la Unidad 7: HTML, JavaScript, explicación línea por línea. Pero ahora con una regla extra: PRUÉBALO TODO CON TUS MANOS. Haz clic, escribe, envía. Los eventos no se entienden leyendo; se entienden haciendo. Todos los ejemplos asumen el script al final del body.

Ejemplo 1: tu primer evento click, y el error del paréntesis. Este ejemplo tiene dos versiones: la correcta y la incorrecta, para que veas con tus propios ojos la diferencia. La versión correcta registra la función sin paréntesis; la incorrecta la ejecuta de inmediato. Prueba ambas y observa: en la incorrecta, el mensaje aparece al cargar la página (sin que hagas clic) y luego el botón no hace nada.

```
<!-- HTML -->
<button id="boton-saludar">Salúdame</button>
<p id="respuesta"></p>
```

```
// VERSIÓN CORRECTA: la función se registra SIN paréntesis
function saludar() {
    document.getElementById("respuesta").textContent = "¡Hola! Gracias por hacer clic.";
}

const boton = document.getElementById("boton-saludar");
boton.addEventListener("click", saludar); // sin paréntesis: "ejecútala cuando hagan clic"
// Prueba: al cargar la página NO pasa nada. Al hacer clic, aparece el saludo. Perfecto.

// VERSIÓN INCORRECTA (no la uses, es para que veas el error):
// boton.addEventListener("click", saludar());
// Con paréntesis, saludar() se ejecuta AHORA MISMO, una vez, al cargar.
// El addEventListener recibe lo que saludar DEVUELVE (undefined), no la función.
// Resultado: el saludo aparece solo al cargar y el botón queda muerto.
```

La analogía que nunca falla: `addEventListener("click", saludar)` es como dejarle una nota al portero: "cuando llegue el paquete, avísame". Le das la INSTRUCCIÓN, no ejecutas la acción. Con paréntesis es como si tú mismo fueras por el paquete en ese momento y le dijeras al portero "ya fui yo", dejándolo sin instrucciones para cuando llegue el paquete de verdad. Cada vez que un botón "no hace nada" al hacer clic, lo primero que reviso es si hay paréntesis de más en el registro. El 80% de las veces, ahí está el bug.

Ejemplo 2: contador de clics, el "hola mundo" de la interactividad. Vamos a combinar eventos con DOM: cada clic incrementa un contador y lo muestra. Este ejemplo introduce el patrón que vas a repetir mil veces: variable de estado + evento que la modifica + actualización del DOM. La variable vive fuera de la función (para que no se reinicie en cada clic) y la función solo la incrementa y la pinta.

```
<!-- HTML -->
<button id="boton-contar">Haz clic</button>
<p id="contador">Clics: 0</p>
```

```
// Estado: vive FUERA del manejador para que se conserve entre clics
let clics = 0;

const botonContar = document.getElementById("boton-contar");
const textoContador = document.getElementById("contador");

botonContar.addEventListener("click", function() {
  clics++; // modificamos el estado
  // Actualizamos el DOM con un template literal
  textoContador.textContent = `Clics: ${clics}`;

  // Un detallito de experiencia: mensaje especial cada 10 clics
  if (clics % 10 === 0) {
    textoContador.textContent += ` ¡${clics} clics, vas con todo!`;
  }
});
```

Tres decisiones de diseño que quiero que notes. Primera: clics es let y está fuera, porque cambia y debe sobrevivir entre eventos. Si la declararas dentro de la función, cada clic empezaría en 0 y el contador nunca avanzaría; es un error típico. Segunda: seleccionamos los elementos una sola vez, fuera del manejador; el manejador solo los usa. Tercera: usamos una función anónima como manejador porque solo se usa aquí; si la función fuera a reutilizarse en otro lado, le pondríamos nombre. Son micro-decisiones, pero el código profesional está hecho de miles de micro-decisiones correctas.

Ejemplo 3: el objeto event y event.target en acción. Tenemos tres botones de sabor de tacos y un solo párrafo de resultado. En lugar de tres manejadores, usamos UNO SOLO y dejamos que event.target nos diga en cuál botón se hizo clic. Además mostramos las coordenadas del clic con clientX/clientY, para que veas la información que trae el mensajero.

```
<!-- HTML -->
<button class="sabor" data-sabor="pastor">Tacos al pastor</button>
<button class="sabor" data-sabor="suadero">Tacos de suadero</button>
<button class="sabor" data-sabor="carnitas">Tacos de carnitas</button>
<p id="eleccion">Elige tu taco favorito</p>
```

```
// UN solo manejador para los tres botones
const botonesSabor = document.querySelectorAll(".sabor");
const eleccion = document.getElementById("eleccion");

botonesSabor.forEach(function(boton) {
  boton.addEventListener("click", function(event) {
    // event.target es el botón específico donde se hizo clic
    const botonPresionado = event.target;
    const sabor = botonPresionado.dataset.sabor;

    eleccion.textContent = `Elegiste tacos de ${sabor}. ¡Buena elección!`;

    // El evento también trae las coordenadas del clic
    console.log(`Clic en x=${event.clientX}, y=${event.clientY}`);
    console.log(`Tipo de evento: ${event.type}`); // "click"
  });
});
```

Este ejemplo es una semilla de delegación de eventos: en lugar de N manejadores para N botones, un manejador inteligente que pregunta "¿quién me llamó?" con `event.target`. Imagina una tienda con 200 productos: ¿vas a escribir 200 `addEventListener`? No; usas uno solo y lees los data- del botón presionado (mira cómo se conectan las unidades: los data attributes de la Unidad 7 ahora brillan). El `console.log` de las coordenadas es para que explores: abre la consola, haz clic en distintas partes del botón y mira cómo cambian los números. Esa curiosidad exploratoria es la que forma programadores.

Ejemplo 4: formulario completo con `preventDefault` y validación. Este es el ejemplo más importante de la unidad. Un formulario de registro con nombre y correo. Interceptamos el submit, evitamos la recarga con `preventDefault`, leemos los valores, validamos y mostramos el resultado. Fíjate en el orden: primero `preventDefault` (línea 1 del manejador, sin excepciones), luego leer, luego validar, luego responder.

```
<!-- HTML -->
<form id="registro">
  <input id="nombre" type="text" placeholder="Tu nombre">
  <input id="correo" type="text" placeholder="tu@correo.com">
  <button type="submit">Registrarme</button>
</form>
<p id="resultado-form"></p>
```

```

const formulario = document.getElementById("registro");
const campoNombre = document.getElementById("nombre");
const campoCorreo = document.getElementById("correo");
const resultadoForm = document.getElementById("resultado-form");

formulario.addEventListener("submit", function(event) {
  event.preventDefault(); // LÍNEA SAGRADA: evita la recarga de la página

  // Leemos los valores (siempre texto, con .value)
  const nombre = campoNombre.value.trim(); // trim quita espacios al inicio/final
  const correo = campoCorreo.value.trim();

  // Validamos
  if (nombre === "") {
    resultadoForm.textContent = "Por favor escribe tu nombre.";
    resultadoForm.className = "invalido"; // reutilizamos las clases de la Unidad 7
    campoNombre.focus(); // devolvemos el cursor al campo problemático
    return; // salimos de la función, no seguimos
  }

  if (!correo.includes("@") || !correo.includes(".")) {
    resultadoForm.textContent = "Ese correo no parece válido.";
    resultadoForm.className = "invalido";
    campoCorreo.focus();
    return;
  }

  // Si llegamos aquí, todo está bien
  resultadoForm.className = "valido";
  resultadoForm.textContent = `¡Bienvenido, ${nombre}! Te escribiremos a ${correo}.`;

  formulario.reset(); // limpiamos los campos para el próximo registro
});

```

Hay varias joyas aquí. El `trim()` es un detalle profesional: los usuarios a veces dejan espacios accidentales al inicio o final (" Ana "), y sin trim esos espacios cuentan como contenido válido o ensucian tus datos. El `return` temprano después de cada error es un patrón limpio: en lugar de anidar `if/else` profundos, cada validación fallida sale de la función inmediatamente; el "camino feliz" queda al final, sin indentación extra. El `campoNombre.focus()` devuelve el cursor al campo con problema, un detallito de experiencia que los usuarios agradecen sin saber por qué. Y `formulario.reset()` limpia todos los campos de un golpe al final, dejándolo listo para el

siguiente registro. Prueba este formulario de tres formas: con el botón, con Enter dentro de un campo, y con campos vacíos. Las tres deben comportarse bien; si alguna recarga la página, tu `preventDefault` no está donde debe.

Ejemplo 5: `setTimeout`, el mensaje que se autodestruye. Un clásico de la web: haces algo, aparece "¡Guardado!" y a los 3 segundos desaparece solo. Usamos `setTimeout` para programar la desaparición y guardamos el id para poder cancelarla con `clearTimeout` si el usuario cierra el mensaje manualmente antes.

```
<!-- HTML -->
<button id="boton-guardar">Guardar</button>
<p id="aviso-guardado"></p>
```

```
const botonGuardar = document.getElementById("boton-guardar");
const aviso = document.getElementById("aviso-guardado");
let temporizadorAviso = null; // aquí guardamos el id del timeout

botonGuardar.addEventListener("click", function() {
  // Si había un aviso anterior programado para desaparecer, lo cancelamos
  // (evita que el mensaje nuevo desaparezca antes de tiempo)
  if (temporizadorAviso !== null) {
    clearTimeout(temporizadorAviso);
  }

  aviso.textContent = "¡Guardado correctamente!";
  aviso.className = "valido";

  // Programamos que desaparezca en 3 segundos (3000 milisegundos)
  temporizadorAviso = setTimeout(function() {
    aviso.textContent = "";
    aviso.className = "";
    temporizadorAviso = null;
  }, 3000);
});
```

El detalle del `clearTimeout` previo es lo que separa el código que "funciona" del código robusto. Imagina: el usuario hace clic en guardar, el mensaje aparece; a los 2 segundos vuelve a hacer clic (porque sí, los usuarios hacen eso); sin el `clearTimeout`, el primer temporizador seguiría vivo y borraría el mensaje nuevo 1 segundo después de aparecer, cuando al usuario le correspondían 3 segundos completos. Con el `clearTimeout`, cada clic reinicia la cuenta de 3 segundos limpiamente. Este patrón (cancelar lo anterior antes de programar lo nuevo) aparece

en buscadores con autocompletado, en validaciones con retraso y en docenas de lugares más. Y nota que inicializamos temporizadorAviso en null y lo volvemos null al terminar: así siempre sabemos si hay un temporizador activo o no.

Ejemplo 6: setInterval, el reloj digital en vivo. El ejemplo canónico de los temporizadores: cada segundo leemos la hora actual y la mostramos. Introducimos el objeto Date para obtener la hora y vemos el patrón completo de un intervalo bien manejado.

```
<!-- HTML -->
<p id="reloj">--:--:--</p>
```

```
const reloj = document.getElementById("reloj");

function actualizarReloj() {
  const ahora = new Date();
  const horas = String(ahora.getHours()).padStart(2, "0");
  const minutos = String(ahora.getMinutes()).padStart(2, "0");
  const segundos = String(ahora.getSeconds()).padStart(2, "0");

  reloj.textContent = `${horas}:${minutos}:${segundos}`;
}

actualizarReloj(); // la llamamos UNA vez de inmediato (para no esperar 1 segundo
en blanco)
const intervaloReloj = setInterval(actualizarReloj, 1000); // y luego cada segundo
```

Dos detalles finos. Primero: llamamos actualizarReloj() una vez manualmente antes de crear el intervalo. ¿Por qué? Porque setInterval espera el primer intervalo completo antes de la primera ejecución: sin esa llamada inicial, el reloj mostraría "--:--:--" durante un segundo completo al cargar. Es un detalle que distingue al programador cuidadoso. Segundo: padStart(2, "0") rellena con un cero a la izquierda cuando el número tiene un solo dígito, para que se vea "09:05:03" en lugar de "9:5:3". String(9).padStart(2, "0") convierte el 9 en texto y lo deja en "09". Los relojes sin ceros a la izquierda se ven amateurs; con ceros se ven profesionales. Son dos líneas que cambian la percepción completa.

Ejemplo 7: cuenta regresiva con clearInterval, el temporizador que se apaga solo. Una cuenta regresiva de 10 segundos que al llegar a cero se detiene sola y muestra un mensaje. Este ejemplo enseña la disciplina del intervalo: nace, trabaja y muere cuando cumple su misión.


```
<!-- HTML -->
<p id="cuenta">10</p>
<button id="boton-iniciar">Iniciar cuenta regresiva</button>
```

```
const displayCuenta = document.getElementById("cuenta");
const botonIniciar = document.getElementById("boton-iniciar");
let intervaloCuenta = null;

botonIniciar.addEventListener("click", function() {
  // Evitamos intervalos duplicados: si ya hay uno corriendo, no creamos otro
  if (intervaloCuenta !== null) {
    return; // salimos sin hacer nada
  }

  let segundos = 10;
  displayCuenta.textContent = segundos;
  botonIniciar.disabled = true; // deshabilitamos el botón (la historia de los boletos)
  botonIniciar.textContent = "Contando...";

  intervaloCuenta = setInterval(function() {
    segundos--;
    displayCuenta.textContent = segundos;

    if (segundos <= 0) {
      clearInterval(intervaloCuenta); // NOS DETENEMOS: misión cumplida
      intervaloCuenta = null;
      displayCuenta.textContent = "¡Tiempo!";
      botonIniciar.disabled = false; // reactivamos el botón
      botonIniciar.textContent = "Iniciar cuenta regresiva";
    }
  }, 1000);
});
```

Este ejemplo es una pequeña obra maestra de buenas prácticas, analicémoslo. La guarda al inicio (`if intervaloCuenta !== null return`) evita el bug del doble intervalo: sin ella, si el usuario hace clic dos veces rápido en "iniciar", se crearían dos intervalos restando al mismo tiempo y la cuenta bajaría de 2 en 2, un bug clásico y confuso. El `botonIniciar.disabled = true` es la lección de la historia de los boletos: mientras el proceso está en marcha, el botón se deshabilita para que el usuario impaciente no dispare el proceso dos veces. Y el `clearInterval` dentro del `if (segundos <= 0)` es la disciplina del temporizador: el intervalo sabe cuándo morir.

Observa también que restauramos todo al estado inicial al terminar (botón reactivado, texto original, variable en null): dejar el estado limpio después de un proceso es lo que permite repetirlo sin bugs. Prueba romperlo a propósito: quita la guarda del inicio y haz doble clic rápido; mira cómo la cuenta se vuelve loca. Luego restaura la guarda y aprecia la diferencia.

Ejemplo 8: mini proyecto integrador, lista de tareas interactiva. Vamos a unir TODO: eventos (clic y submit), DOM (crear elementos, template literals) y un toque de temporizadores. Una lista de tareas donde agregas tareas con un formulario (sin recarga, con preventDefault), cada tarea tiene un botón para eliminarla (eventos en elementos creados dinámicamente), y un contador de tareas pendientes que se actualiza solo.

```
<!-- HTML -->
<form id="form-tarea">
  <input id="nueva-tarea" type="text" placeholder="Escribe una tarea">
  <button type="submit">Agregar</button>
</form>
<p id="contador-tareas">Tareas pendientes: 0</p>
<ul id="lista-tareas"></ul>
```

```

const formTarea = document.getElementById("form-tarea");
const campoTarea = document.getElementById("nueva-tarea");
const listaTareas = document.getElementById("lista-tareas");
const contadorTareas = document.getElementById("contador-tareas");

function actualizarContador() {
  const pendientes = document.querySelectorAll("#lista-tareas li").length;
  contadorTareas.textContent = `Tareas pendientes: ${pendientes}`;
}

formTarea.addEventListener("submit", function(event) {
  event.preventDefault();

  const texto = campoTarea.value.trim();
  if (texto === "") {
    return; // no agregamos tareas vacías
  }

  // Creamos la tarea con createElement (seguro contra XSS, Unidad 7)
  const li = document.createElement("li");
  li.textContent = texto + " ";

  // Botón para eliminarla, creado dinámicamente
  const botonBorrar = document.createElement("button");
  botonBorrar.textContent = "Borrar";

  // El evento se registra en el botón recién creado (sí se puede: ya existe)
  botonBorrar.addEventListener("click", function(event) {
    event.target.parentNode.remove(); // quitamos el <li> papá del botón
    actualizarContador();
  });

  li.appendChild(botonBorrar);
  listaTareas.appendChild(li);

  campoTarea.value = ""; // limpiamos el campo
  campoTarea.focus();    // dejamos el cursor listo para la siguiente tarea
  actualizarContador();
});

```

Este proyecto merece un análisis pausado porque cada línea enseña algo. El submit con preventDefault evita la recarga; el trim evita tareas de puros espacios; el return temprano evita tareas vacías. Creamos el li con createElement y textContent (seguro: si el usuario escribe

HTML, se muestra como texto). El botón de borrar se crea dinámicamente y le registramos su evento INMEDIATAMENTE después de crearlo: esto responde a la pregunta "¿puedo registrar eventos en elementos que no existían al cargar la página?" Sí, siempre que el elemento ya exista en el momento del `addEventListener` (aunque aún no esté pegado al documento). El truco `event.target.parentNode.remove()` es elegante: desde el botón, subimos a su papá (el `li`) y lo eliminamos del árbol. Y los detalles de experiencia: limpiar el campo y devolverle el foco hacen que agregar 10 tareas seguidas sea fluido, sin tener que hacer clic en el campo cada vez. Ese tipo de pulido es lo que hace que una app se sienta "bien hecha". Como reto mental: ¿qué pasaría si en lugar de crear el botón con `createElement` usaras `innerHTML` con el texto del usuario? Piénsalo antes de seguir: sería el agujero XSS de la Unidad 7. Por eso aquí usamos `createElement + textContent`. Las unidades se cuidan entre sí.

Y con esto tienes el repertorio completo en movimiento. Si ejecutaste los 8 ejemplos con tus manos, ya sentiste lo que es una página viva: botones que responden, formularios que no recargan, mensajes que desaparecen solos, relojes que laten cada segundo. La práctica va a convertir esa sensación en habilidad sólida.

Práctica

Los mismos 7 ejercicios, de menor a mayor dificultad, integrando las tres clases. La regla de oro sigue: intenta primero, atórate con dignidad, y solo entonces mira la solución. Cada ejercicio te pide construir algo que existe en sitios reales.

Ejercicio 1: botón que cambia de mensaje (calentamiento de eventos).

Enunciado: Crea un botón con `id="boton-magia"` y un párrafo con `id="texto-magia"`. Registra un evento click en el botón (con `addEventListener`, sin paréntesis en la función) que cambie el texto del párrafo a "¡Hiciste clic! La magia funciona." usando un template literal que incluya la hora actual del clic (`new Date().toLocaleTimeString()`). Haz que cada clic actualice la hora.

Solución:

```
<!-- HTML -->
<button id="boton-magia">Hazme clic</button>
<p id="texto-magia">Aún no has hecho clic.</p>
```

```
const botonMagia = document.getElementById("boton-magia");
const textoMagia = document.getElementById("texto-magia");

botonMagia.addEventListener("click", function() {
  const hora = new Date().toLocaleTimeString();
  textoMagia.textContent = `¡Hiciste clic! La magia funciona. Hora del clic:
  ${hora}`;
});
```

Explicación paso a paso: Seleccionamos el botón y el párrafo una sola vez y los guardamos en constantes. Registramos el manejador con `addEventListener("click", function() {...})`: observa que pasamos la función sin ejecutarla, sin paréntesis, para que el navegador la guarde y la llame en cada clic. Dentro del manejador, creamos la hora actual con `new Date().toLocaleTimeString()`, que devuelve algo como "10:35:22". Construimos el mensaje con un template literal interpolando la hora, y lo asignamos con `textContent`. Cada clic genera una hora nueva porque `new Date()` se ejecuta dentro del manejador, en el momento del clic, no al cargar la página. El error típico sería calcular la hora fuera del manejador: entonces todos los clics mostrarían la hora de carga de la página. La lección: lo que debe actualizarse en cada evento se calcula DENTRO del manejador.

Ejercicio 2: interruptor de like con toggle y contador.

Enunciado: Crea un botón "Me gusta" con id "boton-like" y un párrafo con id "conteo-likes" que inicie en "0 me gusta". Cada clic debe alternar el estado: si no tenía like, suma 1 y el botón muestra "Ya no me gusta" con la clase "activo" (fondo rojo en CSS); si ya tenía like, resta 1 y el botón vuelve a "Me gusta" sin la clase. Usa una variable booleana de estado, `classList.toggle` y un template literal para el texto. El contador nunca debe bajar de 0.

Solución:

```
<!-- HTML -->
<style>
  .activo { background-color: crimson; color: white; }
</style>
<button id="boton-like">Me gusta</button>
<p id="conteo-likes">0 me gusta</p>
```

```

const botonLike = document.getElementById("boton-like");
const conteoLikes = document.getElementById("conteo-likes");

let likes = 0;
let meGusta = false; // estado: ¿el usuario ya dio like?

botonLike.addEventListener("click", function() {
  meGusta = !meGusta; // invertimos el estado (true->false, false->true)

  if (meGusta) {
    likes++;
    botonLike.textContent = "Ya no me gusta";
  } else {
    likes--;
    botonLike.textContent = "Me gusta";
  }

  botonLike.classList.toggle("activo", meGusta); // forma con segundo argumento
  conteoLikes.textContent = `${likes} me gusta`;
});

```

Explicación paso a paso: Tenemos dos variables de estado fuera del manejador: likes (el número) y meGusta (booleano que recuerda si el usuario ya dio like). En cada clic, invertimos meGusta con !meGusta, el operador de negación: si era false se vuelve true y viceversa. Según el nuevo estado, sumamos o restamos y cambiamos el texto del botón. El `classList.toggle` con segundo argumento es una joya poco conocida: `toggle("activo", meGusta)` agrega la clase si el segundo argumento es true y la quita si es false; así la clase siempre refleja el estado, sin ambigüedad. El contador se muestra con un template literal. ¿Por qué no puede bajar de 0? Porque likes solo se resta cuando meGusta pasa de true a false, y solo puede ser true si antes se sumó: la lógica del booleano lo garantiza. Este es el patrón exacto de los botones de "me gusta" reales (con la diferencia de que en una app real el conteo se guardaría en un servidor). El error típico aquí es no llevar el booleano y tratar de deducir el estado del texto del botón o de la clase; eso funciona pero es frágil: el estado debe vivir en variables, no en el DOM.

Ejercicio 3: adivinanza con teclado (evento `keydown` + `event.key`).

Enunciado: El programa elige un número secreto del 1 al 5 (usa un valor fijo, por ejemplo 3, para simplificar). Crea un párrafo con id "pista" que diga "Presiona una tecla del 1 al 5". Registra un evento `keydown` en `document` que lea `event.key`: si la tecla es el número secreto, muestra "¡Adivinaste! Era el 3."; si es otro número del 1 al 5, muestra "No es el X, intenta de

nuevo."; si no es un número del 1 al 5, muestra "Presiona un número del 1 al 5.". Usa template literals en los mensajes.

Solución:

```
<!-- HTML -->
<p id="pista">Presiona una tecla del 1 al 5</p>
```

```
const secreto = 3;
const pista = document.getElementById("pista");

document.addEventListener("keydown", function(event) {
  const tecla = event.key; // qué tecla se presionó, como texto ("1", "a",
  "Enter"...)

  // Verificamos que sea un dígito del 1 al 5
  if (tecla >= "1" && tecla <= "5") {
    if (Number(tecla) === secreto) {
      pista.textContent = `¡Adivinaste! Era el ${secreto}.`;
    } else {
      pista.textContent = `No es el ${tecla}, intenta de nuevo.`;
    }
  } else {
    pista.textContent = "Presiona un número del 1 al 5.";
  }
});
```

Explicación paso a paso: Registramos el evento `keydown` en `document` (todo el documento), porque no hay un campo específico: queremos capturar teclas en cualquier parte de la página. El manejador recibe `event` y lee `event.key`, que trae la tecla como texto: si presionas el 3, `event.key` es "3" (texto, no número). La comparación `tecla >= "1" && tecla <= "5"` funciona con textos porque los dígitos tienen orden alfabético consistente; es un truco válido para este rango. Si la tecla está en rango, la convertimos a número con `Number(tecla)` y la comparamos con el secreto usando `===` estricto. Los mensajes usan template literals con la tecla o el secreto interpolados. Nota que `event.key` distingue mayúsculas y teclas especiales: si presionas "Enter", `event.key` es "Enter", y cae en el mensaje de "presiona un número". Este ejercicio te enseña que los eventos no son solo clics: el teclado es una fuente riquísima de interacción (atajos, juegos, búsquedas). Como extensión, prueba cambiar el secreto por un número aleatorio con `Math.floor(Math.random() * 5) + 1`.

Ejercicio 4: formulario de contacto con validación completa.

Enunciado: Crea un formulario con id "form-contacto" que tenga: input de nombre (id "c-nombre"), input de correo (id "c-correo"), textarea de mensaje (id "c-mensaje"), botón submit "Enviar" y un párrafo id "c-resultado". Al enviar: usa preventDefault; lee los tres valores con trim; valida que el nombre tenga al menos 2 caracteres, que el correo contenga @ y ., y que el mensaje tenga al menos 10 caracteres; muestra el PRIMER error encontrado en el párrafo de resultado (con clase "invalido") y pon el foco en el campo del error; si todo está bien, muestra "¡Gracias, [nombre]! Recibimos tu mensaje y te contactaremos en [correo]." con clase "valido" y limpia el formulario con reset().

Solución:

```
<!-- HTML -->
<form id="form-contacto">
  <input id="c-nombre" type="text" placeholder="Tu nombre">
  <input id="c-correo" type="text" placeholder="tu@correo.com">
  <textarea id="c-mensaje" placeholder="Tu mensaje"></textarea>
  <button type="submit">Enviar</button>
</form>
<p id="c-resultado"></p>
```



```

const formContacto = document.getElementById("form-contacto");
const cNombre = document.getElementById("c-nombre");
const cCorreo = document.getElementById("c-correo");
const cMensaje = document.getElementById("c-mensaje");
const cResultado = document.getElementById("c-resultado");

formContacto.addEventListener("submit", function(event) {
    event.preventDefault();

    const nombre = cNombre.value.trim();
    const correo = cCorreo.value.trim();
    const mensaje = cMensaje.value.trim();

    // Limpiamos el estado visual anterior
    cResultado.className = "";

    if (nombre.length < 2) {
        cResultado.textContent = "El nombre debe tener al menos 2 caracteres.";
        cResultado.classList.add("invalido");
        cNombre.focus();
        return;
    }

    if (!correo.includes("@") || !correo.includes(".")) {
        cResultado.textContent = "Escribe un correo válido (debe tener @ y punto).";
        cResultado.classList.add("invalido");
        cCorreo.focus();
        return;
    }

    if (mensaje.length < 10) {
        cResultado.textContent = "El mensaje debe tener al menos 10 caracteres.";
        cResultado.classList.add("invalido");
        cMensaje.focus();
        return;
    }

    cResultado.classList.add("valido");
    cResultado.textContent = `¡Gracias, ${nombre}! Recibimos tu mensaje y te
contactaremos en ${correo}.`;
    formContacto.reset();
});

```

Explicación paso a paso: El esqueleto es el patrón de formulario moderno: submit + preventDefault como primera línea, lectura con trim, validaciones con return temprano, camino feliz al final. Cada validación muestra el error, marca el párrafo con la clase "invalido" (reutilizando el CSS de la Unidad 7), devuelve el foco al campo problemático y sale con return, de modo que solo se muestra el primer error (menos abrumador para el usuario que una lista de cinco errores). Observa que limpiamos cResultado.className = "" al inicio: si el envío anterior fue válido (clase "valido") y este falla, sin esa limpieza el párrafo tendría ambas clases. El textarea se lee igual que un input: con.value, no con.textContent. Y el mensaje final usa un template literal con nombre y correo interpolados. Este ejercicio es un formulario de contacto real, funcional, del tipo que aparece en miles de sitios; la única pieza que le faltaría en producción es enviar los datos a un servidor (eso llega con fetch en unidades futuras).

Ejercicio 5: semáforo con setTimeout encadenado.

Enunciado: Crea tres párrafos (o divs) con ids "luz-roja", "luz-amarilla", "luz-verde" que representen un semáforo, y un botón "Iniciar semáforo". Define una clase CSS "encendida" que les dé color brillante a cada una (puedes usar una sola clase con colores distintos por id, o tres clases). Al hacer clic en el botón: se enciende la roja por 3 segundos, luego se apaga y se enciende la verde por 3 segundos, luego la amarilla por 1 segundo, y al final todo se apaga y el botón se reactiva. Usa setTimeout encadenados (cada timeout programa el siguiente). Deshabilita el botón mientras el ciclo está en marcha y guarda los ids de los timeouts para poder cancelarlos si el usuario hace clic en un botón "Detener" que también debes crear (usa clearTimeout).

Solución:

```
<!-- HTML -->
<style>
  .luz { width: 60px; height: 60px; border-radius: 50%; background-color: #333;
margin: 5px; }
  #luz-roja.encendida { background-color: red; }
  #luz-amarilla.encendida { background-color: gold; }
  #luz-verde.encendida { background-color: limegreen; }
</style>
<div class="luz" id="luz-roja"></div>
<div class="luz" id="luz-amarilla"></div>
<div class="luz" id="luz-verde"></div>
<button id="boton-semaforo">Iniciar semáforo</button>
<button id="boton-detener">Detener</button>
```

```
const luzRoja = document.getElementById("luz-roja");
```

```

const luzAmarilla = document.getElementById("luz-amarilla");
const luzVerde = document.getElementById("luz-verde");
const botonSemaforo = document.getElementById("boton-semaforo");
const botonDetener = document.getElementById("boton-detener");

let timeouts = []; // guardamos TODOS los ids para poder cancelarlos

function apagarTodo() {
  luzRoja.classList.remove("encendida");
  luzAmarilla.classList.remove("encendida");
  luzVerde.classList.remove("encendida");
}

function programar(funcion, ms) {
  const id = setTimeout(funcion, ms);
  timeouts.push(id); // guardamos el boleto de cancelación
}

botonSemaforo.addEventListener("click", function() {
  botonSemaforo.disabled = true;
  apagarTodo();

  // Fase 1: roja, ahora mismo
  luzRoja.classList.add("encendida");

  // Fase 2: a los 3 segundos, verde
  programar(function() {
    apagarTodo();
    luzVerde.classList.add("encendida");
  }, 3000);

  // Fase 3: a los 6 segundos, amarilla
  programar(function() {
    apagarTodo();
    luzAmarilla.classList.add("encendida");
  }, 6000);

  // Fase 4: a los 7 segundos, todo apagado y botón reactivado
  programar(function() {
    apagarTodo();
    botonSemaforo.disabled = false;
    timeouts = [];
  }, 7000);
});

```

```

botonDetener.addEventListener("click", function() {
  timeouts.forEach(function(id) {
    clearTimeout(id); // cancelamos cada fase programada
  });
  timeouts = [];
  apagarTodo();
  botonSemaforo.disabled = false;
});

```

Explicación paso a paso: La función auxiliar programar envuelve setTimeout y guarda cada id en el arreglo timeouts: es el patrón del "boleto de cancelación" llevado a múltiples temporizadores. El ciclo del semáforo se programa por fases con tiempos absolutos desde el inicio (0, 3000, 6000, 7000 ms): cada fase apaga todo y enciende su luz, y la última además reactiva el botón y limpia el arreglo. El botón de detener recorre los ids guardados y los cancela uno por uno con clearTimeout, apaga las luces y reactiva el botón principal: el semáforo se congela donde estaba, sin fases fantasma que se disparen después. Este ejercicio enseña dos cosas grandes: primera, que los timeouts encadenados sirven para secuencias temporizadas (animaciones, tutoriales paso a paso, presentaciones automáticas); segunda, que todo lo que programas debe ser cancelable, y para eso guardas los ids. El error típico es no guardar los ids y descubrir que el botón "detener" no puede detener nada. Como reto, modifica los tiempos para que la amarilla parpadee 3 veces antes de apagarse (pista: más fases con programar).

Ejercicio 6: cuenta regresiva con fecha objetivo (proyecto real).

Enunciado: Crea un formulario con un input de tipo datetime-local (id "fecha-meta") y un botón "Iniciar cuenta regresiva", más un párrafo id "restante". Al enviar el formulario (con preventDefault), lee la fecha elegida, y cada segundo calcula cuánto falta (días, horas, minutos, segundos) hasta esa fecha y lo muestra con un template literal como "Faltan 2 días, 4 horas, 15 minutos y 30 segundos". Si la fecha ya pasó o llega a cero, detén el intervalo con clearInterval y muestra "¡Llegó el momento!". Si el usuario inicia una nueva cuenta regresiva, detén la anterior primero. Usa la técnica de marcas de tiempo (Date.now()) para que la cuenta sea precisa.

Solución:

```

<!-- HTML -->
<form id="form-cuenta">
  <input id="fecha-meta" type="datetime-local">
  <button type="submit">Iniciar cuenta regresiva</button>
</form>
<p id="restante">Elige una fecha y hora futura.</p>

```

```

const formCuenta = document.getElementById("form-cuenta");
const fechaMeta = document.getElementById("fecha-meta");
const restante = document.getElementById("restante");
let intervaloRegresiva = null;

formCuenta.addEventListener("submit", function(event) {
  event.preventDefault();

  // Si había una cuenta anterior, la detenemos (sin intervalos zombies)
  if (intervaloRegresiva !== null) {
    clearInterval(intervaloRegresiva);
    intervaloRegresiva = null;
  }

  const meta = new Date(fechaMeta.value).getTime();

  if (isNaN(meta)) {
    restante.textContent = "Elige una fecha válida.";
    return;
  }

  function actualizar() {
    const ahora = Date.now();
    let diferencia = meta - ahora;

    if (diferencia <= 0) {
      clearInterval(intervaloRegresiva);
      intervaloRegresiva = null;
      restante.textContent = "¡Llegó el momento!";
      return;
    }

    // Descomponemos los milisegundos en días, horas, minutos y segundos
    const dias = Math.floor(diferencia / (1000 * 60 * 60 * 24));
    diferencia = diferencia % (1000 * 60 * 60 * 24);
    const horas = Math.floor(diferencia / (1000 * 60 * 60));
    diferencia = diferencia % (1000 * 60 * 60);

```

```

const minutos = Math.floor(diferencia / (1000 * 60));
const segundos = Math.floor((diferencia % (1000 * 60)) / 1000);

restante.textContent =
  `Faltan ${dias} días, ${horas} horas, ${minutos} minutos y ${segundos}
segundos.`;
}

actualizar(); // primera actualización inmediata (sin segundo en blanco)
intervaloRegresiva = setInterval(actualizar, 1000);
});

```

Explicación paso a paso: Al enviar, primero detenemos cualquier cuenta anterior: sin esto, dos cuentas correrían a la vez peleando por el mismo párrafo. Convertimos el valor del input (un texto como "2026-12-31T23:59") a milisegundos con `new Date(...).getTime()`; si el usuario no eligió fecha, eso da NaN y lo detectamos con `isNaN` para mostrar un error amable. La función `actualizar` calcula la diferencia entre la meta y el ahora usando `Date.now()`: esta es la técnica de marcas de tiempo, precisa aunque el navegador retrase algún tick. Si la diferencia es cero o negativa, detenemos el intervalo y celebramos. Si no, descomponemos los milisegundos: dividimos entre los milisegundos de un día para los días, tomamos el resto con `%` para las horas, y así sucesivamente; `Math.floor` redondea hacia abajo porque no queremos "2.7 días" sino "2 días". El resultado se muestra con un template literal. Llamamos `actualizar()` una vez de inmediato (el detalle del segundo en blanco) y luego cada segundo con `setInterval`. Este es un proyecto real: las páginas de "próximo evento", "la oferta termina en" y "faltan X para el concierto" funcionan exactamente así. El error típico es calcular con segundos en lugar de milisegundos al crear la meta; otro es olvidar detener la cuenta anterior.

Ejercicio 7: buscador con retraso inteligente (debounce, técnica profesional).

Enunciado: Crea un input de búsqueda (id "buscador") y un párrafo id "estado-busqueda". Cada vez que el usuario escribe (evento input), muestra "Escribiendo..." y programa con `setTimeout` una "búsqueda" que se ejecute 800ms DESPUÉS de que deje de escribir, mostrando `Buscando: "[texto]"` con template literal. Si el usuario sigue escribiendo antes de los 800ms, cancela la búsqueda anterior con `clearTimeout` (para no buscar en cada tecla). Si el campo queda vacío, muestra "Escribe algo para buscar." y no programes nada. Esta técnica se llama `debounce` y es la que usan Google y todos los buscadores con autocompletado.

Solución:

```
<!-- HTML -->
<input id="buscador" type="text" placeholder="Buscar...">
<p id="estado-busqueda">Escribe algo para buscar.</p>
```

```
const buscador = document.getElementById("buscador");
const estadoBusqueda = document.getElementById("estado-busqueda");
let timeoutBusqueda = null;

buscador.addEventListener("input", function(event) {
  const texto = event.target.value.trim();

  // Cancelamos la búsqueda anterior si el usuario sigue escribiendo
  if (timeoutBusqueda !== null) {
    clearTimeout(timeoutBusqueda);
    timeoutBusqueda = null;
  }

  if (texto === "") {
    estadoBusqueda.textContent = "Escribe algo para buscar.";
    return;
  }

  estadoBusqueda.textContent = "Escribiendo...";

  // Programamos la búsqueda 800ms después de la ÚLTIMA tecla
  timeoutBusqueda = setTimeout(function() {
    estadoBusqueda.textContent = `Buscando: "${texto}"`;
    // Aquí iría la búsqueda real (con fetch, en unidades futuras)
    timeoutBusqueda = null;
  }, 800);
});
```

Explicación paso a paso: El evento input se dispara con CADA tecla (a diferencia de change, que espera a que el campo pierda el foco): es el evento correcto para reaccionar mientras se escribe. En cada disparo, lo primero es cancelar el timeout anterior con clearTimeout: esta es la esencia del debounce, "reiniciar el cronómetro en cada tecla". Si el texto está vacío, mostramos el mensaje neutral y salimos sin programar nada (no tiene sentido "buscar" un texto vacío). Si hay texto, mostramos "Escribiendo..." como retroalimentación inmediata y programamos la búsqueda real a 800ms. Solo si el usuario deja de escribir por 800ms completos, el timeout sobrevive y se ejecuta la búsqueda. ¿Por qué 800ms? Es un punto dulce empírico: suficiente para que una persona normal termine de escribir una palabra, no tanto

como para que se sienta lento. Esta técnica es OBLIGATORIA en buscadores reales: sin debounce, cada tecla dispararía una petición al servidor, saturándolo y mostrando resultados parpadeantes para búsquedas a medias ("c", "ca", "cas", "casa"...). Con debounce, solo se busca la palabra completa. Es el ejemplo perfecto de cómo un temporizador bien usado mejora tanto el rendimiento como la experiencia. Si entiendes este ejercicio a fondo, ya piensas como un desarrollador frontend profesional.

Mini-retos sin solución (para investigar y romper cosas): primero, agrega al ejercicio 2 un doble clic que reinicie el contador a cero (pista: evento `dblclick`); segundo, modifica el reloj del Ejemplo 6 para que también muestre la fecha en formato "sábado, 19 de septiembre de 2026" (pista: `getDay`, `getDate`, `getMonth`, `getFullYear` y arreglos con los nombres); tercero, combina el ejercicio 5 y el 6: un semáforo que cambie solo cada 4 segundos con `setInterval` en lugar de timeouts encadenados, y un botón para pausarlo y reanudarlo. Si completas los tres, la interactividad ya es parte de tu caja de herramientas.

Preguntas frecuentes y casos borde

Esta sección responde las preguntas que siempre aparecen cuando se trabaja con eventos y temporizadores en proyectos reales. Léela como un botiquín: quizá no necesites todo hoy, pero cuando te topes con uno de estos problemas, ya sabrás dónde está la cura.

¿Por qué mi botón necesita dos clics para funcionar la primera vez? Casi siempre es un problema de estado inicial. Por ejemplo, un botón que alterna un panel: si tu variable empieza en `false` pero el panel empieza visible en el HTML, el primer clic pone la variable en `true` y oculta el panel... espera, eso suena bien. El problema real aparece cuando la lógica está invertida respecto al estado visual inicial. La solución es simple: asegúrate de que tu variable de estado refleje la realidad del HTML al cargar. Si el panel empieza visible, la variable debe empezar en `true`. Estado inicial consistente, siempre.

¿Por qué mi evento se dispara varias veces con un solo clic? Hay tres sospechosos. Uno: registraste el manejador más de una vez (por ejemplo, tu función de inicialización corre dos veces y cada vez hace `addEventListener`). Dos: la propagación: tienes manejadores en el botón y en su contenedor y ambos responden. Tres: en el caso de submit, a veces el botón dentro del formulario dispara submit y además tienes un listener de click que hace lo mismo. Para diagnosticar, pon un `console.log` con una etiqueta única dentro del manejador y cuenta cuántas veces aparece por cada clic. El número te dice cuántos registros o cuántas fases están activos.

¿Debo usar onclick en el HTML o addEventListener en el JavaScript? addEventListener, sin discusión, en código nuevo. El atributo onclick="hacer()" en el HTML mezcla estructura con comportamiento, solo permite un manejador por evento (si dos scripts asignan onclick, el segundo borra el primero), y es más difícil de depurar y de probar. La única ventaja de onclick es que es visible en el HTML, pero esa "ventaja" es justo el problema: dispersa tu lógica. Verás onclick en tutoriales viejos y en código heredado; entiéndelo, pero no lo uses en tu código nuevo.

¿Qué pasa si registro un evento en un elemento que aún no existe? Nada: obtienes null del selector y luego "Cannot read properties of null" al intentar addEventListener sobre null. Por eso el script va al final del body. Y para elementos creados dinámicamente, registra el evento justo después de crearlos (como en el proyecto de tareas del Ejemplo 8), o usa delegación: registra el evento en un contenedor que SÍ existe desde el inicio y dentro del manejador pregunta con event.target qué hijo fue el que disparó. La delegación es la técnica profesional para listas dinámicas largas.

¿setInterval con 0 milisegundos hace un ciclo infinito instantáneo? No exactamente, y el detalle es interesante: los navegadores imponen un mínimo (típicamente 4ms para timeouts anidados) y, más importante, entre cada ejecución el navegador atiende otras cosas (pintar la pantalla, responder a clics). Un setInterval(fn, 0) ejecuta tu función miles de veces por segundo y congela la página en la práctica. Nunca lo uses así; si necesitas "lo antes posible pero sin bloquear", el mínimo razonable depende del caso, y para animaciones existe requestAnimationFrame, que es la herramienta correcta (sincroniza con el refresco de la pantalla).

¿Los temporizadores siguen corriendo si cambio de pestaña? Sí, pero más lento. Los navegadores reducen la frecuencia de setInterval/setTimeout en pestañas inactivas (a un mínimo de 1 por segundo en muchos casos) para ahorrar batería. Por eso la técnica de marcas de tiempo (Date.now()) es superior para cuentas regresivas: aunque los ticks se espacien, el cálculo sigue siendo correcto porque mide tiempo real transcurrido, no número de ticks. Si tu cuenta regresiva cuenta ticks (segundos-- en cada ejecución), se atrasará en pestañas inactivas; si calcula diferencia de marcas de tiempo, no.

¿Puedo tener varios setInterval corriendo a la vez? Sí, todos los que quieras, cada uno con su id independiente. Un reloj cada segundo, un carrusel cada 5 segundos y un guardado automático cada 30 segundos pueden coexistir sin problema. Solo recuerda la disciplina: cada uno debe tener su clearInterval en el momento adecuado, y guarda cada id en su propia variable con nombre claro (intervaloReloj, intervaloCarrusel), no en una sola variable reutilizada que se pise.

¿Cómo paso el objeto event Y mis propios argumentos a un manejador? Con la función flecha envoltorio: `boton.addEventListener("click", (event) => miFuncion(event, "dato extra"))`. La flecha recibe el event del navegador y se lo pasa a tu función junto con lo que tú quieras. Es el patrón estándar y lo verás en todo el código profesional. Alternativa menos común: usar bind, pero la flecha es más legible.

¿preventDefault evita TODO lo que el navegador haría? Solo lo predeterminado de ESE evento en ESE elemento. En un submit, evita la recarga y el envío nativo. Pero ojo: no detiene la propagación (para eso es stopPropagation) ni evita que otros manejadores del mismo evento se ejecuten. Son tres conceptos distintos: preventDefault ("no hagas tu acción por defecto"), stopPropagation ("no avises a mis ancestros") y return ("sal de mi función"). No los confundas: cada uno frena una cosa distinta.

Cuidado con... el evento submit y el botón fuera del formulario. Si tu botón de "enviar" está FUERA de la etiqueta `<form>` (por razones de diseño, pasa mucho), hacer clic en él NO dispara el evento submit del formulario. Soluciones: mete el botón dentro del form, o usa el atributo `form="id-del-formulario"` en el botón para asociarlo, o escucha el clic del botón y llama `formulario.requestSubmit()` (que sí dispara el evento submit con validación incluida; no uses `formulario.submit()` porque ese se salta los eventos y la validación).

¿Sabías que? Hay un evento llamado `beforeunload` que te permite advertir al usuario si intenta cerrar la pestaña con cambios sin guardar ("¿Seguro que quieres salir? Tienes cambios sin guardar"). Los navegadores lo restringen mucho (no puedes personalizar el mensaje por seguridad), pero bien usado evita pérdidas de datos. Úsalo solo cuando haya cambios reales sin guardar; abusar de él es una de las formas más rápidas de irritar usuarios.

Otro caso borde: los eventos de teclado y los inputs. Si registras `keydown` en `document` para atajos (por ejemplo, presionar "g" para guardar), ese evento TAMBIÉN se dispara cuando el usuario está escribiendo en un campo de texto, y tu atajo se activaría a media palabra. La solución profesional: dentro del manejador, verifica `event.target`; si es un input, textarea o select, ignora el atajo (o manéjalo distinto). Es un detalle que separa los atajos que "funcionan" de los que "funcionan bien".

Y un último consejo de arquitectura para cuando tus páginas crezcan: organiza tus eventos por zonas. En lugar de registrar 30 `addEventListener` regados por todo el script, agrúpalos en una función `inicializarEventos()` que se llame al cargar. Así, cuando algo no responde, sabes exactamente dónde buscar: en un solo lugar están todos los oídos de tu página. Es una convención simple que paga dividendos en cuanto tu código pasa de juguete a proyecto.

Resumen de la unidad

Los eventos son avisos del navegador de que algo pasó; tu código los escucha con `elemento.addEventListener("evento", funcionSinParentesis)` y responde con manejadores. Pasar la función sin paréntesis la registra para después; con paréntesis la ejecuta de inmediato (el error número uno de la unidad).

El objeto `event` trae información de lo ocurrido: `event.target` dice qué elemento disparó el evento, `event.key` qué tecla se presionó, `event.clientX/clientY` dónde fue el clic. Los eventos se propagan del elemento hacia sus ancestros (burbuja); `event.stopPropagation()` detiene esa subida cuando lo necesitas.

Los formularios se interceptan escuchando `submit` en el `<form>` (no click en el botón) y llamando `event.preventDefault()` como primera línea para evitar la recarga. Los valores se leen con `input.value` (siempre texto: convierte a número cuando lo necesites), se limpian con `trim()`, se validan con `returns` tempranos, y el formulario se reinicia con `reset()`. La validación con JavaScript mejora la experiencia; la seguridad real siempre está en el servidor.

`setTimeout` ejecuta una función una vez después de N milisegundos; `setInterval` la repite cada N milisegundos hasta que lo detengas. Ambas devuelven un id que guardas para cancelar con `clearTimeout/clearInterval`. Todo intervalo que creas debe tener un momento de muerte definido; los intervalos zombies causan lentitud misteriosa. Para pasar argumentos usa una función flecha envoltorio: `setTimeout(() => saludar("Ana"), 1000)`.

Los temporizadores no bloquean: programan la función en la cola de eventos y JavaScript sigue trabajando. Para secuencias usa `timeouts` encadenados; para ritmos fijos usa `setInterval` con una primera ejecución inmediata; para precisión usa marcas de tiempo con `Date.now()`. La técnica `debounce` (cancelar y reprogramar un `timeout` en cada tecla) es el estándar para buscadores con autocompletado.

Con eventos, formularios y temporizadores sobre el DOM de la Unidad 7, ya puedes construir páginas que se sienten vivas: es el corazón del desarrollo frontend y la base sobre la que se construyen las aplicaciones modernas.

UNIDAD 9: Datos del servidor

Hasta ahora, todo lo que has construido en este curso ha vivido en un mundo cerrado: la página, el navegador, tu código. Las variables que declaraste, los arreglos que manipulaste, los objetos que creaste... todo existía únicamente dentro de esa pequeña burbuja que se abría y se cerraba cada vez que recargabas la página. Y está bien, porque para aprender necesitaste dominar la lógica, la sintaxis, las estructuras de control y el DOM sin distracciones. Pero ahora llegó el momento de romper la burbuja. Porque el JavaScript de verdad, el que mueve la web que usas todos los días, no vive solo: habla con servidores, pide datos a distancia, espera respuestas y reacciona cuando algo sale mal. Cuando abres tu red social favorita y aparecen tus notificaciones, cuando buscas un producto y la lista se actualiza sin recargar la página, cuando el mapa muestra tu ubicación en tiempo real... detrás de cada una de esas magias hay un programa JavaScript pidiendo datos a un servidor y pintándolos en pantalla. En esta unidad vas a aprender exactamente cómo se hace eso.

Vamos a desarmar el tema en tres piezas que encajan entre sí como engranes. Primero, JSON: el idioma universal con el que los servidores hablan con tu código. Aprenderás su sintaxis exacta, cómo convertir un objeto de JavaScript a texto y de vuelta, y qué errores típicos te van a tronar el programa si no cuidas el formato. Segundo, las promesas: la forma en que JavaScript maneja las esperas, esas operaciones que tardan tiempo y cuyo resultado no llega de inmediato. Entenderás los estados (pendiente, cumplida, rechazada), cómo reaccionar a cada uno con `then`, `catch` y `finally`, y cómo encadenar varias operaciones una tras otra sin que el código se convierta en un enredo. Tercero, la fetch API: la herramienta real que usan las páginas modernas para pedir datos por internet. Con ella harás peticiones GET de verdad, leerás respuestas, convertirás el resultado y manejarás los errores de red como un profesional. Al terminar esta unidad, sabrás conectar tu página con el mundo exterior. Es el salto más grande del curso hasta ahora, así que respira, agarra tu café, y vamos paso a paso.

Piensa en lo que sabes hacer hasta este momento. Sabes seleccionar elementos del DOM, sabes escuchar clics, sabes recorrer arreglos y mostrar listas. Imagina que quieres mostrar una lista de tus diez películas favoritas. Hoy la escribes a mano dentro de tu código, en un arreglo que tú mismo tecleaste. Funciona, sí. Pero ¿qué pasa si quieres que esa lista cambie sin que toques el código? ¿Qué pasa si las películas están en una base de datos de alguien más, en un servidor en otro país, actualizándose cada hora? No puedes meterte a su servidor a copiar y pegar. Lo que sí puedes hacer es pedirle los datos por internet, recibirlos como texto, convertirlos a objetos de JavaScript y pintarlos en tu página. Esa conversación, esa petición-respuesta, es el corazón de la web moderna, y tú vas a aprender a participar en ella.

Aquí hay algo importante que tienes que entender antes de seguir: tu página y el servidor son dos computadoras distintas. La tuya ejecuta el JavaScript en el navegador; la otra, en algún centro de datos, guarda la información. Se comunican por internet usando mensajes con un formato acordado, casi siempre texto plano. JSON es ese formato: una manera de escribir objetos y arreglos como texto, tan simple que cualquier lenguaje de programación puede leerlo y escribirlo. Python lo entiende, PHP lo entiende, Java lo entiende, y por supuesto JavaScript lo entiende, porque JSON nació de la sintaxis de los objetos de JavaScript. Cuando domines JSON, habrás aprendido a hablar el idioma de los servidores.

Pero pedir datos por internet tiene una complicación fundamental que no existe en el código que has escrito hasta ahora: el tiempo. Cuando tu código declara una variable, el resultado es instantáneo. Cuando le pide datos a un servidor, la respuesta tarda: décimas de segundo, segundos, a veces más. El navegador no puede quedarse congelado esperando; la página tiene que seguir respondiendo a clics, a scroll, a todo. JavaScript resuelve esto con un modelo que al principio parece raro: en lugar de esperar la respuesta en ese momento, el código dice "ve a pedir los datos, y cuando los tengas, avísame". Esa promesa de avisar es literalmente una Promesa, el objeto Promise, y en esta unidad lo vas a entender a fondo: qué significa que una operación esté pendiente, cómo se cumple, cómo se rechaza, y cómo tu código reacciona en cada caso.

Y para pedir los datos existe fetch, una función del navegador que devuelve una promesa. Es la puerta de entrada a las APIs: esos servicios en internet que te dan datos a cambio de una petición bien formada. Hay APIs públicas gratuitas para practicar: de países, de chistes, de usuarios falsos, de clima. Con fetch vas a consumir algunas de verdad. Vas a ver respuestas reales, vas a manejar errores reales de red, y vas a sentir esa satisfacción especial de programador cuando tu página muestra datos que viven en otro continente.

También quiero que entiendas por qué este tema importa en tu carrera, no solo en este curso. La gran mayoría de los trabajos de desarrollo web implican, de una forma u otra, pedir datos a un servidor y mostrarlos. Las entrevistas técnicas preguntan sobre promesas. Los frameworks modernos (React, Vue, Angular) están contruidos sobre estos conceptos. Si entiendes JSON, promesas y fetch de verdad, no de memoria sino de comprensión, tienes la base para todo lo que sigue. Es un tema que intimida al principio porque introduce ideas nuevas (asincronía, estados, errores de red), pero una vez que "hace clic" en tu cabeza, se vuelve natural. La mayoría de los programadores recuerdan el momento en que entendieron las promesas como un parteaguas. Esta unidad está diseñada para que ese momento te llegue a ti.

Una advertencia honesta antes de empezar: vas a cometer errores con este tema, y está bien. Los errores típicos tienen nombre y apellido: olvidar que JSON no permite comas al final,

olvidar que las claves llevan comillas dobles, tratar de usar los datos de fetch antes de que lleguen, olvidar convertir la respuesta a JSON, no manejar el caso en que el servidor falla. Todos, absolutamente todos los programadores, hemos cometido cada uno de esos errores. La diferencia entre un principiante y un profesional no es quién comete menos errores, sino quién los reconoce más rápido. En la sección de explicación detallada te voy a mostrar cada error típico, te voy a contar cómo se siente cuando lo cometes, y te voy a enseñar a detectarlo. Considéralo un mapa de minas: mejor saber dónde están antes de pisarlas.

Otra cosa: en esta unidad el código deja de ser instantáneo y eso cambia tu manera de pensar. Hasta ahora, cuando escribías una línea, la siguiente línea se ejecutaba después, en orden, siempre. Con operaciones asíncronas, el orden se vuelve flexible: pides los datos, el código sigue avanzando, y la respuesta llega "cuando llegue". Tu trabajo es escribir código que funcione bien en ese mundo flexible. Requiere práctica, requiere que te equivoques unas cuantas veces, y requiere paciencia contigo mismo. No te compares con nadie: este tema le cuesta trabajo a todo el mundo al principio, incluso a quienes ya programan en otros lenguajes.

El panorama de la unidad es este. En "Explicación detallada" vamos a construir los fundamentos: primero JSON con su sintaxis exacta y sus reglas estrictas, luego la conversión entre objetos y texto con `JSON.stringify` y `JSON.parse`, después las promesas con sus estados y sus métodos, y finalmente fetch con peticiones GET reales y manejo de errores. En "Ejemplos paso a paso" vas a ver todo eso en acción con código comentado que puedes copiar y probar. En "Práctica" resolverás ejercicios que van de lo simple a lo retador, incluyendo simular una petición completa con manejo de errores. Y en el "Resumen" tendrás los puntos clave para repasar.

Un consejo práctico: para esta unidad te conviene tener la consola del navegador abierta mientras lees. Muchos ejemplos los puedes pegar directamente en la consola y ver qué pasa. Los que usan fetch necesitan internet, pero eso lo tienes. Y si algo no sale a la primera, no borres el código con coraje: lee el mensaje de error, que casi siempre te dice exactamente qué está mal. Los mensajes de error son tus aliados, no tus enemigos. Vamos a empezar.

Explicación detallada

JSON: el idioma de los servidores

Empecemos por lo más concreto: JSON. Las siglas significan JavaScript Object Notation, "notación de objetos de JavaScript". Es una forma de escribir datos como texto. Eso es todo. No es un lenguaje de programación, no se ejecuta, no hace nada por sí solo: es simplemente un formato de texto para representar datos estructurados. Y sin embargo, este humilde formato de texto es uno de los acuerdos más importantes de toda la industria del software. Casi cada vez que una app de tu teléfono carga información, hay JSON viajando por internet.

Mira este ejemplo. En JavaScript tienes un objeto así:

```
let persona = {  
  nombre: "María",  
  edad: 28,  
  esEstudiante: false  
};
```

Ese objeto vive en la memoria del navegador. No puedes enviarlo por internet tal cual, porque internet transporta texto (y bytes), no objetos de memoria de JavaScript. Entonces lo conviertes a JSON, que se ve así:

```
{"nombre":"María","edad":28,"esEstudiante":false}
```

¿Ves la diferencia? Es el mismo dato, pero escrito como una sola cadena de texto, con reglas muy estrictas. Las claves llevan comillas dobles. Los textos llevan comillas dobles. No hay variables, no hay funciones, no hay comentarios, no hay comas colgando al final. Es un formato deliberadamente simple y deliberadamente estricto. Simple, para que cualquier lenguaje pueda leerlo; estricto, para que no haya ambigüedades.

Aquí viene la primera analogía que te va a servir para todo el tema. Piensa en JSON como el formato de una carta formal. Cuando escribes una carta formal, hay reglas: fecha arriba, saludo, cuerpo, despedida, firma. Si te saltas la firma, la carta sigue siendo legible, pero no cumple el formato. JSON es igual: tiene reglas de formato, y si las rompes, el texto deja de ser JSON válido y nadie lo puede leer. La diferencia es que una persona puede adivinar qué quisiste decir en una carta mal formateada, pero una computadora no adivina: si el JSON está mal, el programa truena con un error. Por eso la sintaxis exacta importa tanto.

Vamos a ver las reglas de JSON una por una, con calma, porque cada regla existe por una razón.

Regla uno: las claves siempre llevan comillas dobles. En JavaScript puedes escribir `{nombre: "María"}` sin comillas en la clave y funciona. En JSON, eso es inválido. Tiene que ser `{ "nombre": "María"}`. ¿Por qué? Porque JSON no es JavaScript: es un formato independiente que debe poder leerse desde cualquier lenguaje. Las comillas dobles eliminan cualquier duda sobre dónde empieza y termina el nombre de la clave. Es una regla de interoperabilidad: si todos escriben las claves igual, todos los lenguajes pueden leerlas igual.

Regla dos: los valores de texto usan comillas dobles, nunca simples. En JavaScript `"hola"` y `'hola'` son lo mismo. En JSON, solo las dobles valen. `{"mensaje": "hola"}` es válido; `{'mensaje': 'hola'}` no lo es. Esto confunde a muchísima gente al principio porque en JavaScript estamos acostumbrados a mezclar comillas. Pero piénsalo así: JSON eligió UN estilo y lo hizo obligatorio para que no haya dos formas de escribir lo mismo. Menos opciones, menos errores de interpretación entre sistemas distintos.

Regla tres: no hay comas al final. En JavaScript moderno puedes escribir `[1, 2, 3,]` con una coma colgando y no pasa nada. En JSON, `{"a": 1,}` es inválido. Esa coma final es probablemente el error más común de todos, porque muchos editores y muchos programadores la usan en JavaScript por comodidad (facilita agregar elementos). En JSON tienes que quitarla. Cuando veas el error `"Unexpected token" in JSON`, casi siempre es una coma colgando antes de la llave de cierre.

Regla cuatro: no hay comentarios. En JavaScript pones `//` o `/* */` para explicar tu código. En JSON no existen los comentarios. Si intentas poner uno, el texto deja de ser JSON válido. Esto molesta al principio, pero tiene lógica: JSON es para que las máquinas se comuniquen, y las máquinas no necesitan comentarios. Si quieres documentar algo, hazlo fuera del JSON.

Regla cinco: no hay funciones, no hay undefined, no hay fechas como tipo especial. JSON solo admite estos tipos de datos: textos, números, booleanos, null, arreglos y objetos. Nada más. Si tu objeto de JavaScript tiene una función, al convertirlo a JSON la función se pierde (desaparece en silencio). Si tiene undefined, también desaparece. Si tiene una fecha (un objeto Date), se convierte a texto automáticamente. Esto es crucial: JSON es un formato de DATOS, no de código. Solo transporta información, nunca comportamiento.

Regla seis: los números se escriben sin comillas y sin formatos raros. `28` es válido, `"28"` es un texto (no un número), y cosas como `NaN` o `Infinity` no existen en JSON. Si intentas convertir `NaN` a JSON, se convierte en `null`. Esto tiene implicaciones prácticas: si tu API te manda un número como texto (`"precio": "199.99"`), tendrás que convertirlo tú en JavaScript.

Memoriza estas reglas no como una lista aburrida, sino como el "contrato" que firmas cada vez que hablas con un servidor. El servidor te manda texto que cumple estas reglas; tú le mandas

texto que cumple estas reglas. Si alguno de los dos rompe el contrato, la comunicación falla. Y cuando falla, el error casi siempre es de formato: una comilla mal puesta, una coma de más, una comilla simple donde iba doble.

¿Sabías que? JSON fue creado por Douglas Crockford a principios de los 2000, y se popularizó porque era muchísimo más ligero que el formato que dominaba antes, XML. Donde XML necesitaba etiquetas de apertura y cierre para todo (<nombre>María</nombre>), JSON lo decía en la mitad de caracteres ({\"nombre\":\"María\"). En la época en que el internet era lento y cada byte contaba, eso fue una revolución. Hoy JSON es el estándar de facto para las APIs web. Cuando aprendas JSON, estás aprendiendo el formato que mueve billones de mensajes al día.

Cuidado con... confundir un objeto de JavaScript con un JSON. Se parecen muchísimo, y esa similitud es una trampa. Un objeto de JavaScript vive en memoria y puede tener funciones; un JSON es texto y solo tiene datos. Cuando ves {\"nombre\":\"María\"} en la pantalla, pregúntate: ¿esto es texto o es un objeto? Si está entre comillas (es una cadena), es JSON en forma de texto. Si no está entre comillas (es código que escribiste), es un objeto de JavaScript. La conversión entre uno y otro es justamente lo que hacen JSON.stringify y JSON.parse, que veremos ahora.

JSON.stringify y JSON.parse: el puente entre texto y objetos

Tu programa trabaja con objetos. Internet transporta texto. Necesitas un puente en ambas direcciones, y JavaScript te da dos funciones para eso: JSON.stringify y JSON.parse. Son el par más importante de esta unidad; los vas a usar en cada proyecto que hable con un servidor.

JSON.stringify toma un valor de JavaScript y lo convierte a una cadena de texto en formato JSON. El nombre lo dice: \"stringify\" = \"convertir a string\". Mira:

```
let usuario = {
  nombre: \"Carlos\",
  edad: 30,
  activo: true,
  hobbies: [\"leer\", \"correr\"]
};

let texto = JSON.stringify(usuario);
```

Después de esto, texto contiene la cadena {\"nombre\":\"Carlos\",\"edad\":30,\"activo\":true,\"hobbies\":[\"leer\",\"correr\"]}. Ya no es un objeto: es

texto. Puedes guardarlo, puedes enviarlo por internet, puedes meterlo en el localStorage del navegador. Es la "versión transportable" de tus datos.

JSON.parse hace lo contrario: toma una cadena en formato JSON y la convierte en un valor de JavaScript. "Parse" significa analizar sintácticamente: la función lee el texto, verifica que cumpla las reglas de JSON, y construye el objeto correspondiente.

```
let textoRecibido = '{"nombre":"Ana","edad":25}';  
let persona = JSON.parse(textoRecibido);  
console.log(persona.nombre); // "Ana"
```

Ahora persona es un objeto de verdad: puedes acceder a persona.nombre, pasarlo a funciones, meterlo en arreglos. El texto se convirtió en datos utilizables.

El flujo completo, el que vas a repetir cientos de veces en tu vida como programador, es este: pides datos al servidor -> recibes texto JSON -> lo conviertes con JSON.parse -> trabajas con el objeto -> si necesitas enviar datos de vuelta, conviertes tu objeto con JSON.stringify -> envías el texto. Parse para recibir, stringify para enviar. Grábatelo: parse = entrada (texto -> objeto), stringify = salida (objeto -> texto).

Ahora, los casos borde y los errores típicos, que es donde realmente se aprende.

Error típico uno: JSON.parse con texto mal formado. Si el texto no cumple las reglas de JSON, parse lanza un error y tu programa se detiene (a menos que lo captures, cosa que veremos con try/catch en la unidad 10, pero puedes anticipar la idea). Ejemplo:

```
JSON.parse('{"nombre": "Luis",}'); // ERROR: coma al final  
JSON.parse('{\"nombre\": \"Luis\"}'); // ERROR: comillas simples  
JSON.parse('{"nombre": "Luis"}'); // ERROR: falta la llave de cierre
```

Cada uno de estos truena con un SyntaxError y un mensaje como "Unexpected token". Cuando veas ese error, no entres en pánico: significa "el texto no es JSON válido", y el mensaje te dice más o menos dónde está el problema. Revisa comas, comillas y llaves. El 90% de las veces es una de esas tres cosas.

Error típico dos: hacer parse de algo que ya es un objeto. Esto pasa mucho con fetch, como verás: algunas funciones ya te devuelven el objeto convertido, y si le aplicas JSON.parse otra vez, truena porque parse espera texto, no objetos. Si haces JSON.parse({nombre: "Luis"}), JavaScript primero convierte el objeto a texto (que queda como "[object Object]") y luego intenta parsearlo, lo cual falla. Regla de oro: parse solo se aplica a cadenas de texto. Si ya tienes un objeto, no lo parsees.

Error típico tres: hacer stringify de algo que ya es texto. Si tu variable ya es una cadena JSON y le aplicas JSON.stringify, obtienes una cadena con comillas escapadas:

"{\\\"nombre\\\":\\\"Luis\\\"}\"". Es un texto dentro de un texto, un desastre. Regla de oro espejo: stringify solo se aplica a objetos (o arreglos, números, etc.), no a textos que ya son JSON.

Caso borde: stringify con funciones y undefined. Mira qué pasa:

```
let raro = {
  nombre: "Pedro",
  saludar: function() { return "hola"; },
  apodo: undefined,
  edad: 40
};
JSON.stringify(raro); // '{"nombre":"Pedro","edad":40}'
```

La función y el undefined desaparecieron en silencio, sin error, sin aviso. Esto es una fuente clásica de bugs sutiles: mandas un objeto al servidor, y del otro lado faltan campos, y no entiendes por qué. Ahora ya sabes por qué: JSON no transporta funciones ni undefined. Si necesitas mandar algo parecido, conviértelo a un formato que JSON sí entienda (por ejemplo, una descripción en texto).

Caso borde: fechas. Los objetos Date se convierten a texto ISO automáticamente:

```
JSON.stringify({ hoy: new Date() });
// '{"hoy":"2026-09-19T14:30:00.000Z"}'
```

Del otro lado, cuando hagas parse, obtendrás un texto, no un objeto Date. Si necesitas operar con la fecha, tendrás que crear el Date de nuevo con new Date(texto). Es un detalle que muerde cuando trabajas con APIs que mandan fechas.

Caso borde: stringify con formato bonito. JSON.stringify acepta parámetros extra para que el texto salga legible, con sangría. JSON.stringify(objeto, null, 2) produce un texto con saltos de línea y dos espacios de sangría. Es utilísimo para depurar: cuando quieras ver en la consola

cómo se ve tu objeto como JSON, usa el 2. No lo uses para enviar datos al servidor (ocupa más espacio), pero para tus ojos es oro.

Buena práctica: cuando recibas JSON de un servidor y algo falle, lo primero que debes hacer es imprimir el texto crudo antes de parsearlo. `console.log` del texto tal cual te llegó. La mitad de las veces descubrirás que el servidor no te mandó lo que esperabas: a veces manda una página de error en HTML en lugar de JSON, a veces manda texto vacío, a veces manda un mensaje de error en texto plano. Si intentas parsear HTML como JSON, truena, y el error te confunde porque habla de "Unexpected token <" (el < de <html>). Ver el texto crudo te ahorra horas de confusión.

Pregunta frecuente: "¿JSON y objeto de JavaScript son lo mismo?" No. Se parecen porque JSON nació de la sintaxis de objetos de JS, pero uno es texto y el otro es una estructura en memoria. Piensa en una fotografía de una casa versus la casa: se parecen, una representa a la otra, pero no son lo mismo y no puedes vivir dentro de la foto. JSON es la foto; el objeto es la casa.

Pregunta frecuente: "¿Por qué las claves necesitan comillas dobles si en JS no las necesitan?" Porque JSON es un estándar independiente del lenguaje. Si las claves no llevaran comillas, cada lenguaje tendría que adivinar dónde termina el nombre de la clave, y los lenguajes con reglas distintas de identificadores se confundirían. Las comillas dobles son el acuerdo universal: "aquí empieza y aquí termina el nombre".

Promesas: aprender a esperar sin bloquearse

Ahora cambiamos de tercio. Dejamos el formato de los datos y entramos al tema del tiempo: las promesas. Este es el concepto más importante de la unidad y, para muchos, el más difícil de todo el curso. Tómate tu tiempo con esta sección. Léela dos veces si hace falta. Vale la pena.

El problema que resuelven las promesas es este: algunas operaciones tardan. Pedir datos a un servidor tarda. Leer un archivo tarda. Esperar un temporizador tarda. En los lenguajes antiguos, el programa se quedaba congelado esperando (espera bloqueante): nada más se movía hasta que llegaba la respuesta. En una página web, eso sería fatal: la página se congelaría, los botones no responderían, el usuario pensaría que se trabó todo. JavaScript, que nació para páginas web interactivas, necesitaba otra solución: pedir la operación, seguir con lo suyo, y que le avisen cuando termine. Esa es la espera no bloqueante, y las promesas son su herramienta principal.

Una promesa (Promise) es un objeto que representa un valor que todavía no existe, pero que existirá en el futuro. Es como el ticket que te dan en la farmacia cuando dejas una receta: el ticket no es tu medicina, es la promesa de que tu medicina estará lista. Con el ticket puedes irte a hacer otras cosas; cuando te llaman por tu número, regresas y recoges. La promesa funciona igual: tu código pide la operación, recibe una promesa (el ticket), sigue ejecutando otras cosas, y cuando la operación termina, la promesa "avisa" y tu código reacciona.

Toda promesa vive en uno de tres estados, y entenderlos es entender las promesas:

1. Pendiente (pending): la operación todavía no termina. Es el estado inicial. Pediste los datos, el ticket está en tu mano, la medicina se está preparando. En este estado, la promesa no tiene resultado todavía.
2. Cumplida (fulfilled): la operación terminó bien y la promesa tiene un valor. Te llamaron por tu número y te entregaron la medicina. El valor puede ser cualquier cosa: un texto, un objeto, un arreglo.
3. Rechazada (rejected): la operación falló y la promesa tiene un motivo de error. Se acabó el medicamento, hubo un problema. En lugar de un valor, la promesa guarda la razón del fallo (normalmente un objeto Error).

Una regla fundamental: una promesa solo cambia de estado una vez. De pendiente pasa a cumplida o a rechazada, y ahí se queda para siempre. No puede pasar de cumplida a rechazada, ni volver a pendiente. Es como el ticket de la farmacia: una vez que te entregaron la medicina (o te dijeron que no hay), el ticket ya cumplió su función. Esta inmutabilidad es lo que hace a las promesas confiables: cuando reaccionas a una promesa, sabes que su resultado no va a cambiar debajo de tus pies.

¿Cómo reaccionas a una promesa? Con tres métodos: `then`, `catch` y `finally`.

`then` se ejecuta cuando la promesa se cumple. Le pasas una función, y esa función recibe el valor resultante:

```
miPromesa.then(function(datos) {  
  console.log("Llegaron los datos:", datos);  
});
```

`catch` se ejecuta cuando la promesa se rechaza. Le pasas una función que recibe el error:

```
miPromesa.then(function(datos) {  
  console.log("Llegaron los datos:", datos);  
}).catch(function(error) {  
  console.log("Algo falló:", error);  
});
```

finally se ejecuta siempre, se cumpla o se rechace la promesa. Es para limpieza: ocultar un indicador de "cargando", por ejemplo:

```
miPromesa  
  .then(function(datos) { console.log("Éxito:", datos); })  
  .catch(function(error) { console.log("Fallo:", error); })  
  .finally(function() { console.log("Terminó, sea como sea"); });
```

Fíjate en algo importante: then, catch y finally no se ejecutan de inmediato. Se registran ahora y se ejecutan después, cuando la promesa cambie de estado. Tu código sigue avanzando mientras tanto. Esto es lo que más confunde al principio, así que vamos a verlo con un ejemplo concreto usando setTimeout, que ya conoces de la clase 24:

```
console.log("1: Empiezo");  
  
let promesa = new Promise(function(resolve, reject) {  
  setTimeout(function() {  
    resolve("datos listos");  
  }, 2000);  
});  
  
promesa.then(function(resultado) {  
  console.log("3: " + resultado);  
});  
  
console.log("2: Sigo con lo mío");
```

¿Qué imprime esto y en qué orden? Primero "1: Empiezo". Luego se crea la promesa (queda pendiente) y se registra el then. Luego se imprime "2: Sigo con lo mío" INMEDIATAMENTE, sin esperar los 2 segundos. Y dos segundos después, cuando el temporizador termina y se llama a resolve, se imprime "3: datos listos". El orden real es 1, 2, 3, aunque en el código el then aparece antes del console.log del 2. Si esperabas 1, 3, 2, necesitas reprogramar tu intuición: el código asíncrono no se ejecuta en el orden en que está escrito, sino en el orden en que las cosas terminan.

Vamos a desarmar el constructor de Promise porque lo vas a ver mucho: `new Promise(function(resolve, reject) {...})`. Le pasas una función (el "ejecutor") que recibe dos funciones: `resolve` y `reject`. Dentro del ejecutor haces la operación que tarda (un temporizador, una petición, lo que sea). Cuando termina bien, llamas a `resolve(valor)` y la promesa pasa a cumplida con ese valor. Cuando falla, llamas a `reject(error)` y pasa a rechazada con ese error. Tú decides cuándo y con qué. Es como si tú fueras el farmacéutico: tú decides cuándo la medicina está lista (`resolve`) o cuándo avisas que no hay (`reject`).

Analogía de la vida real que te va a acompañar: pedir comida a domicilio. Llamas al restaurante (inicias la operación), te dan un número de pedido (la promesa, pendiente). Cuelgas y sigues con tu vida: ves la tele, trabajas (el código sigue ejecutándose). Cuando el repartidor llega (la operación termina), te avisan (`then`) y recibes tu comida (el valor). Si el restaurante se quedó sin ingredientes, te llaman para cancelar (`catch`) y te explican por qué (el error). Y de cualquier forma, guardas el teléfono (`finally`, limpieza). Nunca te quedas pegado al teléfono esperando sin hacer nada: eso sería la espera bloqueante, y en la web no nos la podemos permitir.

Ahora, el encadenamiento (chaining), que es donde las promesas muestran su verdadero poder. `then` devuelve OTRA promesa. Esto significa que puedes encadenar varios `then` uno tras otro, y cada uno recibe el resultado del anterior:

```
pedirUsuario()
  .then(function(usuario) {
    return pedirPedidos(usuario.id);
  })
  .then(function(pedidos) {
    return calcularTotal(pedidos);
  })
  .then(function(total) {
    console.log("Total:", total);
  })
  .catch(function(error) {
    console.log("Falló algo en la cadena:", error);
  });
```

Lee esto como una historia: pido el usuario; cuando llega, pido sus pedidos; cuando llegan, calculo el total; cuando está listo, lo muestro. Si ALGO falla en cualquier punto de la cadena, el `catch` del final lo atrapa. Esa es la belleza: un solo `catch` al final maneja errores de toda la cadena. No necesitas verificar errores en cada paso.

Hay dos reglas de oro del encadenamiento que te van a ahorrar muchos dolores de cabeza:

Regla de oro uno: si dentro de un then inicias otra operación asíncrona, DEVUÉVELA con return. Si no la devuelves, la cadena no espera: el siguiente then se ejecuta de inmediato con undefined. Mira la diferencia:

```
// MAL: sin return, la cadena no espera
pedirUsuario().then(function(usuario) {
  pedirPedidos(usuario.id); // se pierde, nadie espera esto
}).then(function() {
  console.log("Esto se ejecuta antes de que lleguen los pedidos");
});

// BIEN: con return, la cadena espera
pedirUsuario().then(function(usuario) {
  return pedirPedidos(usuario.id); // la cadena espera esta promesa
}).then(function(pedidos) {
  console.log("Pedidos:", pedidos); // esto sí espera
});
```

Este es probablemente el error número uno con promesas: olvidar el return dentro del then. El código "funciona" (no truena), pero los datos llegan tarde o nunca, y te vuelves loco buscando el bug. Cada vez que un then te dé undefined inesperado, revisa: ¿devolví la promesa con return?

Regla de oro dos: un then sin catch es una promesa de problemas. Si una promesa se rechaza y nadie la captura con catch, el error se pierde en silencio (o aparece como advertencia en la consola que nadie lee). Es como pedir comida a domicilio y no contestar cuando te llaman para decirte que no hay: el restaurante se queda con la comida fría y tú con hambre, y nadie sabe qué pasó. Acostúmbrate: toda cadena de promesas termina en catch. Siempre.

¿Sabías que? Antes de las promesas, JavaScript manejaba la asincronía con callbacks anidados, y eso producía el famoso "callback hell" o "pirámide de la perdición": código con tantos niveles de sangría que parecía una pirámide inclinada hacia la derecha, imposible de leer y de depurar. Las promesas se inventaron precisamente para aplanar esa pirámide y convertirla en una cadena vertical legible. Cuando veas código antiguo con callbacks anidados cinco niveles, agradecerás vivir en la era de las promesas.

Cuidado con... pensar que then "pausa" el código. No lo pausa. then registra una reacción futura y el código sigue. Si necesitas el valor de la promesa, TODO lo que dependa de ese valor tiene que estar dentro del then (o en thens encadenados). Este es el error conceptual más común:


```
// MAL: usar el valor fuera del then
let datos;
fetch(url).then(function(respuesta) {
  datos = respuesta;
});
console.log(datos); // undefined, porque esto se ejecuta ANTES de que llegue la respuesta
```

No hay forma de "sacar" el valor de una promesa al código síncrono. El valor vive en el futuro, y tu código que lo necesita tiene que vivir en el futuro también, dentro del then. Cuando en la unidad 10 veas `async/await`, verás una sintaxis que PARECE síncrona pero respeta esta misma regla por debajo.

Otro cuidado: las promesas se ejecutan una sola vez. Si llamas a then dos veces sobre la misma promesa, ambas reacciones reciben el mismo resultado; la operación no se repite. La promesa recuerda su resultado. Esto es útil (puedes reaccionar al mismo dato en varios lugares), pero significa que si quieres "reintentar" una operación, necesitas crear una promesa nueva, no reusar la vieja.

Pregunta frecuente: "¿Las promesas son multihilo? ¿JavaScript hace dos cosas a la vez?" No exactamente. JavaScript tiene un solo hilo de ejecución: hace una cosa a la vez. Lo que pasa es que mientras espera (por ejemplo, a que el servidor responda), no se queda de brazos cruzados: atiende otras cosas (clics, temporizadores, otras promesas). Es como un cocinero que pone a hervir el agua y mientras tanto pica verduras: no está haciendo dos cosas literalmente al mismo tiempo, pero aprovecha los tiempos muertos. Ese modelo se llama "event loop", y las promesas son la forma elegante de trabajar con él.

Pregunta frecuente: "¿Cuándo se ejecuta el código dentro de `new Promise`?" De inmediato. El ejecutor se corre en cuanto creas la promesa, de forma síncrona. Lo que es asíncrono es la reacción (`then/catch`). Mucha gente cree que todo lo de adentro de la promesa es "futuro", pero no: el trabajo empieza ya; lo que llega después es el resultado.

Fetch API: pidiendo datos de verdad

Con JSON y promesas en tu caja de herramientas, ya puedes entender `fetch`: la función del navegador para hacer peticiones por internet. `fetch` es, en el fondo, una forma elegante de decir "ve a esta dirección, tráeme lo que haya, y avísame con una promesa".

La forma más simple de usar `fetch` es para pedir datos (petición GET):

```
fetch("https://api.ejemplo.com/usuarios")
  .then(function(respuesta) {
    return respuesta.json();
  })
  .then(function(datos) {
    console.log(datos);
  })
  .catch(function(error) {
    console.log("Error de red:", error);
  });
```

Vamos a desarmar esto línea por línea, porque cada línea esconde un concepto.

`fetch(url)` inicia la petición y devuelve una promesa INMEDIATAMENTE (en estado pendiente). No espera a que el servidor responda. Tu código sigue. Cuando el servidor responde (con éxito o con error de red), la promesa cambia de estado.

El primer `then` recibe un objeto `Response` (respuesta). Ojo: este objeto NO son los datos todavía. Es el "sobre" que contiene los datos: trae información sobre la respuesta (código de estado, cabeceras) y el cuerpo sin leer. Para leer el cuerpo como JSON usas `respuesta.json()`, que a su vez devuelve OTRA promesa (porque leer el cuerpo también toma tiempo). Por eso necesitas el segundo `then`: el primero convierte, el segundo recibe los datos ya convertidos.

¿Por qué dos pasos? Porque la respuesta llega en dos tiempos: primero llegan los encabezados ("sí, te voy a mandar datos, son de tipo JSON, pesan tanto"), y luego llega el cuerpo con los datos. `fetch` te avisa en cuanto llegan los encabezados (para que puedas decidir rápido, por ejemplo, si el servidor dijo "no encontrado"), y `respuesta.json()` espera a que llegue el cuerpo completo y lo convierte. Es eficiente, pero al principio sorprende. Grábatelo: `fetch` te da la respuesta; `respuesta.json()` te da los datos.

El `catch` del final atrapa errores de red: si no hay internet, si el dominio no existe, si el servidor no responde. Esos son los errores que rechazan la promesa de `fetch`.

Pero aquí viene una trampa famosa, una de las más famosas de todo JavaScript: `fetch` NO rechaza la promesa cuando el servidor responde con un error HTTP, como un 404 (no encontrado) o un 500 (error del servidor). Para `fetch`, "el servidor respondió" = "la operación de red funcionó", aunque la respuesta sea un error. La promesa se cumple con un objeto `Response` cuyo código de estado es 404. Si no verificas el estado, tu código intentará leer datos que no existen.

¿Cómo se verifica? Con `respuesta.ok`, una propiedad booleana que es `true` si el código de estado está entre 200 y 299 (éxito), y `false` si no. El patrón profesional es este:

```
fetch("https://api.ejemplo.com/usuarios/999")
  .then(function(respuesta) {
    if (!respuesta.ok) {
      throw new Error("Error del servidor: " + respuesta.status);
    }
    return respuesta.json();
  })
  .then(function(datos) {
    console.log(datos);
  })
  .catch(function(error) {
    console.log("Falló:", error.message);
  });
```

Fíjate en el `throw` dentro del `then`: lanzar un error dentro de un `then` hace que la promesa encadenada se rechace, y el `catch` del final lo atrapa. Es el mecanismo estándar para convertir "el servidor respondió con error" en "la cadena se va por el camino del `catch`". Este patrón (verificar `ok`, lanzar error si no) deberías usarlo en cada `fetch` que escribas. Hazlo un hábito, como ponerte el cinturón de seguridad.

Además de `json()`, el objeto `Response` tiene otros métodos para leer el cuerpo según lo que esperes: `respuesta.text()` para texto plano, `respuesta.blob()` para archivos binarios (imágenes, por ejemplo). Cada uno devuelve una promesa. El 95% de las veces usarás `json()`, porque las APIs hablan JSON, pero es bueno saber que existen.

Hablemos de GET, el tipo de petición que estamos haciendo. GET significa "dame datos": es para leer, no para modificar. Cuando pones una URL en el navegador y presionas Enter, el navegador hace un GET. `fetch` sin opciones extra también hace un GET. Existen otros métodos (POST para enviar datos, PUT, DELETE), pero en esta unidad nos concentramos en GET: pedir y leer. (En la unidad 10 verás cómo enviar datos si el proyecto lo necesita, pero el 80% del trabajo con APIs es GET.)

Otro concepto clave: las APIs y sus URLs. Una API es un servicio en internet que responde a peticiones con datos. Sus direcciones suelen tener una estructura lógica. Por ejemplo, una API de usuarios podría tener: `/usuarios` (lista todos), `/usuarios/5` (el usuario con id 5). A veces necesitas pasar parámetros en la URL, como `?limite=10` (dame solo 10). Esos parámetros se llaman "query strings" y son parte normal de trabajar con APIs: no te asustes si ves URLs largas con `?` y `&`; son solo instrucciones para el servidor.

Para practicar, existen APIs públicas gratuitas que no piden registro. La más famosa para aprender es JSONPlaceholder (jsonplaceholder.typicode.com): una API falsa con usuarios, publicaciones y comentarios, diseñada exactamente para practicar fetch. Puedes pedir <https://jsonplaceholder.typicode.com/users> y recibirás un arreglo de 10 usuarios falsos en JSON. Es el campo de entrenamiento perfecto: datos reales viajando por internet de verdad, sin riesgo y sin registro. Úsala para todos los ejercicios de esta unidad.

Cuidado con... el CORS. Algún día harás fetch a una API y verás un error de "CORS policy" en la consola. CORS es un mecanismo de seguridad del navegador: por defecto, una página solo puede pedir datos a su mismo origen (mismo dominio), a menos que el servidor diga explícitamente "permite peticiones de otros orígenes". Las APIs públicas como JSONPlaceholder ya permiten todo, así que para practicar no tendrás problemas. Pero si algún día consumes una API privada y ves error de CORS, sabrás que no es tu código el que falla: es el servidor el que no te dio permiso. Es un tema de seguridad, no un bug tuyo.

Cuidado con... olvidar que fetch es asíncrono. Todo lo que ya aprendiste de promesas aplica: no puedes usar los datos fuera del then, necesitas return en los thens encadenados, y toda cadena termina en catch. El error clásico del principiante con fetch es este:

```
// MAL
let usuarios = [];
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function(r) { return r.json(); })
  .then(function(datos) { usuarios = datos; });
console.log(usuarios.length); // 0, porque la respuesta aún no llega
mostrarUsuarios(usuarios); // muestra nada
```

La solución es mover el trabajo dentro de la cadena:

```
// BIEN
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function(r) { return r.json(); })
  .then(function(usuarios) {
    console.log(usuarios.length); // 10
    mostrarUsuarios(usuarios); // muestra los 10
  })
  .catch(function(error) { console.log("Error:", error); });
```

Todo lo que dependa de los datos vive dentro del then. Sin excepciones. Esta regla te va a perseguir (en el buen sentido) por el resto de tu carrera.

¿Sabías que? `fetch` es relativamente nuevo en la historia de la web. Antes se usaba `XMLHttpRequest`, un objeto con un nombre larguísimo y una API torpe que parecía diseñada para hacerte sufrir (y en parte lo era: nació en los 2000 para Outlook Web). `fetch` llegó para reemplazarlo con una interfaz basada en promesas, mucho más limpia. Todavía verás `XMLHttpRequest` en código viejo y en tutoriales antiguos; ahora sabes que `fetch` es su sucesor moderno. Si un tutorial te enseña `XMLHttpRequest` en 2026, busca otro tutorial.

Anécdota de error típico, de esas que te pasan una vez y no se te olvidan: un estudiante hizo `fetch` a una API, convirtió con `.json()`, y en el siguiente `then` intentó hacer `datos.map(...)`, pero le tronó diciendo que `map` no es una función. ¿Qué pasó? La API, cuando no encontraba resultados, devolvía un OBJETO con un mensaje de error (`{"error": "no encontrado"}`), no un arreglo. El código asumía que siempre llegaba un arreglo. Moraleja: nunca asumas la forma de los datos; cuando algo falle, imprime los datos crudos con `console.log` y mira qué te mandó realmente el servidor. Las APIs cambian, los errores pasan, y tu código debe estar preparado para lo inesperado. Verificar con un `console.log` antes de operar con los datos te ahorra horas.

Otra anécdota clásica: alguien escribía `fetch(url).then(respuesta.json())`. ¿Ves el error? Falta la función intermediaria: escribió `respuesta.json()` directamente como argumento del `then`, pero `respuesta` no existe fuera del `then`. Lo correcto es `.then(function(respuesta) {return respuesta.json();})`. El `then` necesita una FUNCIÓN que reciba la respuesta; no puedes llamar a `.json()` sobre algo que aún no tienes. Es un error de "querer correr antes de caminar" con la asincronía, y es muy común en las primeras semanas.

Buena práctica general para esta unidad: cuando trabajes con `fetch`, escribe siempre la cadena completa desde el principio, con el `catch` incluido, aunque creas que "esta API nunca falla". Las APIs fallan. El internet se cae. Los servidores se actualizan a las 3 de la mañana. El código profesional no es el que funciona cuando todo sale bien; es el que se comporta con dignidad cuando algo sale mal. Un mensaje de error amable para el usuario ("No pudimos cargar los datos, intenta de nuevo") vale más que mil líneas de código perfecto que truenan en silencio.

Y una última idea para cerrar la explicación: nota cómo las tres piezas de esta unidad se conectan. `fetch` pide el texto por internet (promesa pendiente -> cumplida). `respuesta.json()` convierte ese texto JSON en objeto (otra promesa). Tu código recibe el objeto y lo pinta en el DOM (que ya dominas de unidades anteriores). JSON es el idioma, las promesas son la paciencia, `fetch` es el mensajero. Juntas, te dan el poder de construir páginas vivas, conectadas al mundo. Eso es lo que hacen las páginas de verdad. Y ahora tú sabes cómo.

Ejemplos paso a paso

En esta sección vas a ver todo lo aprendido en acción. Cada ejemplo viene con código comentado que puedes copiar, pegar en la consola del navegador (o en un archivo) y probar. Te recomiendo que no solo los leas: ejecútalos, rompelos a propósito, cámbiales valores y observa qué pasa. Así es como realmente se aprende.

Ejemplo 1: Convertir un objeto a JSON y de vuelta

Este es el ciclo fundamental: objeto -> texto -> objeto. Lo vas a repetir toda tu vida como programador.

```
// Paso 1: tenemos un objeto normal de JavaScript en memoria
let producto = {
  nombre: "Laptop",
  precio: 12999,
  enStock: true,
  colores: ["negro", "gris"]
};

// Paso 2: lo convertimos a texto JSON para poder enviarlo o guardarlo
let textoJSON = JSON.stringify(producto);
console.log(textoJSON);

// Muestra:
{"nombre":"Laptop","precio":12999,"enStock":true,"colores":["negro","gris"]}
console.log(typeof textoJSON); // "string" -> ya es texto, no objeto

// Paso 3: simulamos que ese texto viajó por internet y nos llegó de vuelta
// Lo convertimos de nuevo a un objeto usable
let productoRecuperado = JSON.parse(textoJSON);
console.log(productoRecuperado.nombre); // "Laptop"
console.log(typeof productoRecuperado); // "object" -> otra vez es objeto

// Paso 4: verificamos que los datos sobrevivieron el viaje
console.log(productoRecuperado.precio); // 12999
console.log(productoRecuperado.colores[0]); // "negro"
```

Fíjate en el detalle del `typeof`: antes de `stringify` es `"object"`, después es `"string"`, y después de `parse` vuelve a ser `"object"`. Ese cambio de tipo es la prueba de que la conversión realmente ocurrió. Si alguna vez dudas si una variable es texto JSON o un objeto, `typeof` te saca de la duda en un segundo.

Ejemplo 2: JSON.stringify con formato legible para depurar

Cuando el JSON sale en una sola línea es difícil de leer. Para depurar, usa los parámetros extra de stringify.

```
let pedido = {
  cliente: "María",
  items: [
    { nombre: "Taza", cantidad: 2 },
    { nombre: "Plato", cantidad: 4 }
  ],
  total: 850
};

// Sin formato: todo en una línea, difícil de leer
console.log(JSON.stringify(pedido));

// Con formato: el tercer parámetro (2) indica espacios de sangría
console.log(JSON.stringify(pedido, null, 2));
// Muestra algo así:
// {
//   "cliente": "María",
//   "items": [
//     { "nombre": "Taza", "cantidad": 2 },
//     ...
//   ],
//   "total": 850
// }
```

El segundo parámetro (null) es un "replacer" que casi nunca usarás al principio; el tercero es la sangría. Úsalo cada vez que quieras inspeccionar una estructura grande en la consola. Para enviar datos al servidor, en cambio, usa stringify sin parámetros: ocupa menos espacio y viaja más rápido.

Ejemplo 3: Los errores de formato más comunes (y cómo se ven)

Vamos a provocar los errores a propósito para que los reconozcas cuando te salgan de verdad.

```

// ERROR 1: coma colgando al final
try {
  JSON.parse('{"nombre": "Luis", "edad": 30,}');
} catch (e) {
  console.log("Error 1:", e.message);
  // "Unexpected token } in JSON at position..." -> la coma antes de } sobra
}

// ERROR 2: comillas simples en lugar de dobles
try {
  JSON.parse('{\"nombre\": \"Luis\"}');
} catch (e) {
  console.log("Error 2:", e.message);
  // "Unexpected token ' in JSON..." -> JSON exige comillas dobles
}

// ERROR 3: texto que no es JSON (por ejemplo, un HTML de error del servidor)
try {
  JSON.parse("<html>Error 404</html>");
} catch (e) {
  console.log("Error 3:", e.message);
  // "Unexpected token < in JSON..." -> el servidor no mandó JSON
}

// ERROR 4: parsear algo que ya es objeto
try {
  let yaEsObjeto = { nombre: "Ana" };
  JSON.parse(yaEsObjeto);
} catch (e) {
  console.log("Error 4:", e.message);
  // Falla porque parse espera texto, no objetos
}

```

Nota que usé try/catch para atrapar los errores y que el programa no se detenga. En la unidad 10 verás try/catch a fondo; por ahora solo observa el patrón: intentas algo riesgoso dentro de try, y si falla, el catch lo maneja con elegancia en lugar de tronar todo.

El error 3 merece atención especial: "Unexpected token <" es la señal clásica de que el servidor te respondió con una página HTML (normalmente de error) en lugar de JSON. Cuando lo veas, no revises tu parse: revisa la URL, revisa si el servidor está caído, imprime el texto crudo. El problema está en la respuesta, no en tu código.

Ejemplo 4: Tu primera promesa desde cero

Vamos a crear una promesa manualmente para entender `resolve` y `reject` por dentro. Usaremos `setTimeout` para simular una operación que tarda.

```
console.log("Inicio del programa");

// Creamos una promesa que simula pedir datos a un servidor
let pedirDatos = new Promise(function(resolve, reject) {
  console.log("Pidiendo datos al servidor...");

  setTimeout(function() {
    let exito = true; // cambia a false para simular un fallo

    if (exito) {
      // Todo salió bien: cumplimos la promesa con los datos
      resolve({ nombre: "Carlos", edad: 30 });
    } else {
      // Algo falló: rechazamos la promesa con el motivo
      reject(new Error("El servidor no respondió"));
    }
  }, 2000); // simulamos que tarda 2 segundos
});

// Reaccionamos a la promesa: qué hacer si se cumple y si se rechaza
pedirDatos
  .then(function(datos) {
    console.log("Datos recibidos:", datos.nombre, datos.edad);
  })
  .catch(function(error) {
    console.log("Hubo un problema:", error.message);
  })
  .finally(function() {
    console.log("Operación terminada (sea como sea)");
  });

console.log("El programa sigue sin esperar a los datos");
```

El orden de los mensajes será: "Inicio del programa", "Pidiendo datos al servidor...", "El programa sigue sin esperar a los datos", y dos segundos después: "Datos recibidos: Carlos 30" y "Operación terminada". Cambia `exit` a `false` y verás el camino del `catch`. Juega con este ejemplo hasta que el orden de ejecución te parezca obvio: ese es el momento en que las promesas "hacen clic".

Ejemplo 5: Encadenar promesas paso a paso

Ahora encadenamos varias operaciones donde cada una depende del resultado de la anterior.
Simulamos: pedir un usuario, luego sus pedidos, luego el total.

```

// Función que devuelve una promesa: simula buscar un usuario
function buscarUsuario(id) {
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve({ id: id, nombre: "María" });
    }, 1000);
  });
}

// Función que devuelve una promesa: simula buscar los pedidos del usuario
function buscarPedidos(usuarioId) {
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve([
        { producto: "Libro", precio: 300 },
        { producto: "Taza", precio: 150 }
      ]);
    }, 1000);
  });
}

// Encadenamos: cada then devuelve la siguiente promesa con return
buscarUsuario(7)
  .then(function(usuario) {
    console.log("Usuario:", usuario.nombre);
    return buscarPedidos(usuario.id); // CLAVE: devolvemos la promesa
  })
  .then(function(pedidos) {
    console.log("Pedidos encontrados:", pedidos.length);
    let total = 0;
    for (let p of pedidos) {
      total = total + p.precio;
    }
    return total; // devolvemos un valor normal, también funciona
  })
  .then(function(total) {
    console.log("Total a pagar: $" + total);
  })
  .catch(function(error) {
    console.log("Algo falló en la cadena:", error.message);
  });

```

Observa que en el segundo then devolvemos un número normal (total), no una promesa. Eso también funciona: el siguiente then lo recibe directamente. La regla es: lo que devuelvas (con return) en un then, lo recibe el siguiente then. Si devuelves una promesa, la cadena espera a que se cumpla; si devuelves un valor normal, la cadena sigue de inmediato con ese valor.

Prueba romperlo a propósito: quita el return de return buscarPedidos(usuario.id) y observa cómo el siguiente then recibe undefined y el total sale mal (NaN). Luego vuelve a ponerlo. Ese experimento de 30 segundos te enseñará más que mil palabras sobre la importancia del return.

Ejemplo 6: fetch real a una API pública (GET)

Ahora sí, datos de verdad viajando por internet de verdad. Usaremos JSONPlaceholder, la API gratuita para practicar.

```
// Pedimos la lista de usuarios a la API de práctica
fetch("https://jsonplaceholder.typicode.com/users")
  .then(function(respuesta) {
    // Verificamos que el servidor respondió con éxito (código 200-299)
    if (!respuesta.ok) {
      throw new Error("El servidor respondió con error: " + respuesta.status);
    }
    // Convertimos el cuerpo de la respuesta de texto JSON a objeto
    // OJO: .json() devuelve otra promesa, por eso el return
    return respuesta.json();
  })
  .then(function(usuarios) {
    // Aquí ya tenemos el arreglo de usuarios como objetos de JavaScript
    console.log("Llegaron " + usuarios.length + " usuarios");

    // Mostramos los primeros 3 en la consola
    for (let i = 0; i < 3; i++) {
      console.log(usuarios[i].name + " - " + usuarios[i].email);
    }
  })
  .catch(function(error) {
    // Atrapa errores de red Y los errores que lanzamos con throw
    console.log("No se pudieron cargar los datos:", error.message);
  });
```

Pega esto en la consola del navegador (necesitas internet) y verás 10 usuarios con nombres y correos. Son datos falsos, pero el viaje es real: salieron de un servidor, cruzaron internet,

llegaron a tu navegador y tu código los convirtió en objetos. La primera vez que ves esto funcionar se siente como magia, y es normal que quieras probarlo diez veces.

Ejemplo 7: fetch con verificación de error HTTP (404)

Vamos a pedir algo que no existe para ver el camino del error y entender por qué hay que verificar `respuesta.ok`.

```
// Pedimos el usuario 999, que no existe en la API de práctica
fetch("https://jsonplaceholder.typicode.com/users/999")
  .then(function(respuesta) {
    console.log("Código de estado:", respuesta.status); // 404
    console.log("¿Fue exitosa?", respuesta.ok); // false

    if (!respuesta.ok) {
      // Lanzamos un error para que la cadena se vaya al catch
      throw new Error("No se encontró el usuario (código " + respuesta.status +
        ")");
    }
    return respuesta.json();
  })
  .then(function(usuario) {
    // Este then NO se ejecuta, porque lanzamos el error arriba
    console.log(usuario);
  })
  .catch(function(error) {
    console.log("Manejo del error:", error.message);
    // "No se encontró el usuario (código 404)"
  });
```

Sin el `if (!respuesta.ok)`, la promesa de `fetch` se habría "cumplido" con un 404, el código habría intentado leer datos inexistentes, y el bug habría sido confuso. Con la verificación, el error es claro y el `catch` lo maneja. Este patrón de cuatro líneas (`if!ok -> throw`) es uno de los hábitos más valiosos que te puedes llevar de esta unidad.

Ejemplo 8: Mostrar datos de una API en la página (DOM + fetch)

Juntemos todo: pedimos datos con `fetch` y los pintamos en el HTML usando el DOM que ya dominas. Este es el ejemplo más parecido a una aplicación real.

Supón este HTML:

```
<h1>Usuarios</h1>
<button id="btnCargar">Cargar usuarios</button>
<p id="estado"></p>
<ul id="lista"></ul>
```

Y este JavaScript:

```
let boton = document.getElementById("btnCargar");
let estado = document.getElementById("estado");
let lista = document.getElementById("lista");

boton.addEventListener("click", function() {
  // Avisamos al usuario que estamos cargando
  estado.textContent = "Cargando...";
  lista.innerHTML = ""; // limpiamos la lista anterior

  fetch("https://jsonplaceholder.typicode.com/users")
    .then(function(respuesta) {
      if (!respuesta.ok) {
        throw new Error("Error del servidor: " + respuesta.status);
      }
      return respuesta.json();
    })
    .then(function(usuarios) {
      estado.textContent = "Se cargaron " + usuarios.length + " usuarios:";

      // Por cada usuario, creamos un <li> y lo agregamos a la lista
      usuarios.forEach(function(usuario) {
        let item = document.createElement("li");
        item.textContent = usuario.name + " (" + usuario.email + ")";
        lista.appendChild(item);
      });
    })
    .catch(function(error) {
      // Si algo falla, se lo decimos al usuario en la página, no solo en consola
      estado.textContent = "No se pudieron cargar los usuarios. Intenta de nuevo.";
      console.log("Detalle del error:", error.message);
    });
});
```

Este ejemplo combina tres unidades del curso: eventos (clase 22), DOM (clases 20-21) y fetch (esta unidad). Nota los detalles profesionales: el mensaje "Cargando..." para que el usuario

sepa que algo está pasando, la limpieza de la lista anterior, y el mensaje de error visible en la página. Una app que no avisa qué está pasando se siente rota aunque funcione. Prueba este ejemplo en un archivo HTML real: es el mini-proyecto perfecto para esta unidad.

Ejemplo 9: Enviar datos con JSON.stringify (preparar un POST)

Aunque en esta unidad nos enfocamos en GET (pedir), es útil ver cómo se preparan los datos para enviar, porque usa JSON.stringify y completa el ciclo.

```
// Datos que el usuario escribió en un formulario (por ejemplo)
let nuevoUsuario = {
  nombre: "Lucía",
  email: "lucia@ejemplo.com",
  edad: 27
};

// Los convertimos a texto JSON para enviarlos al servidor
let cuerpoJSON = JSON.stringify(nuevoUsuario);
console.log(cuerpoJSON);
// {"nombre":"Lucía","email":"lucia@ejemplo.com","edad":27}
console.log(typeof cuerpoJSON); // "string"

// Así se vería la petición POST completa (para referencia futura):
/*
fetch("https://jsonplaceholder.typicode.com/users", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: cuerpoJSON
})
  .then(function(respuesta) { return respuesta.json(); })
  .then(function(datos) { console.log("Servidor respondió:", datos); })
  .catch(function(error) { console.log("Error:", error.message); });
*/
```

El header "Content-Type: application/json" le dice al servidor "lo que te mando es JSON, trátalo como tal". Sin ese header, muchos servidores no saben cómo interpretar el cuerpo. No necesitas memorizar el POST ahora; solo entiende la idea: para enviar, conviertes con stringify y mandas el texto. Para recibir, pides con GET y conviertes con parse (o respuesta.json()).

Ejemplo 10: Mini-reto guiado -- cadena con finally y limpieza

Un patrón muy común: mostrar un "cargando", hacer la petición, y quitar el "cargando" pase lo que pase. finally es perfecto para eso.

```
function cargarPublicaciones() {
  let aviso = document.getElementById("aviso");
  aviso.textContent = "Cargando publicaciones...";
  aviso.style.display = "block";

  fetch("https://jsonplaceholder.typicode.com/posts")
    .then(function(respuesta) {
      if (!respuesta.ok) {
        throw new Error("Error: " + respuesta.status);
      }
      return respuesta.json();
    })
    .then(function(posts) {
      console.log("Primer título:", posts[0].title);
      aviso.textContent = "Listo: " + posts.length + " publicaciones.";
    })
    .catch(function(error) {
      aviso.textContent = "Falló la carga. Revisa tu conexión.";
      console.log(error.message);
    })
    .finally(function() {
      // Esto se ejecuta SIEMPRE: éxito o fallo
      // Aquí iría, por ejemplo, ocultar un spinner animado
      console.log("Limpieza: la petición terminó");
    });
}
```

¿Sabías que? Muchos programadores olvidan el finally y duplican la limpieza en el then y en el catch (ocultar el spinner en ambos lugares). Funciona, pero si mañana agregas otro camino (otro catch, otro then), tienes que acordarte de duplicarlo otra vez. finally elimina esa duplicación: la limpieza vive en un solo lugar y siempre se ejecuta. Es el equivalente asíncrono de "apaga la luz al salir, pase lo que pase".

Práctica

Es hora de ensuciarte las manos. Estos ejercicios van de lo básico a lo retador. Intenta resolver cada uno por tu cuenta antes de mirar la solución: la tentación de espiar es grande, pero el aprendizaje está en el esfuerzo. Si te atorras más de 15 minutos, mira la solución, entiéndela, ciérrala y vuelve a escribirla tú solo.

Ejercicio 1: Convierte un objeto a JSON y de vuelta

Enunciado: Crea un objeto llamado libro con las propiedades titulo ("Cien años de soledad"), autor ("Gabriel García Márquez"), paginas (432) y disponible (true). Convierte el objeto a texto JSON con JSON.stringify, guárdalo en una variable llamada libroJSON e imprime el texto y su tipo con typeof. Después convierte el texto de vuelta a objeto con JSON.parse, guárdalo en libroRecuperado e imprime el título y el autor desde el objeto recuperado.

Solución:

```
let libro = {
  titulo: "Cien años de soledad",
  autor: "Gabriel García Márquez",
  paginas: 432,
  disponible: true
};

// Objeto -> texto JSON
let libroJSON = JSON.stringify(libro);
console.log(libroJSON);
console.log(typeof libroJSON); // "string"

// Texto JSON -> objeto
let libroRecuperado = JSON.parse(libroJSON);
console.log(libroRecuperado.titulo); // "Cien años de soledad"
console.log(libroRecuperado.autor); // "Gabriel García Márquez"
```

Explicación paso a paso: Primero creamos el objeto con sus cuatro propiedades, mezclando textos, un número y un booleano para practicar los tipos que JSON admite. Luego JSON.stringify lo convierte en una sola cadena de texto; el typeof confirma que ya no es un objeto sino un string. Finalmente JSON.parse lee ese texto y reconstruye un objeto equivalente, del cual podemos leer titulo y autor con la notación de punto normal. Si los valores impresos coinciden con los originales, el ciclo funcionó: los datos sobrevivieron la conversión de ida y vuelta.

Ejercicio 2: Detecta y corrige JSON inválido

Enunciado: Las siguientes tres cadenas intentan ser JSON pero tienen errores de formato.

Para cada una: (a) identifica qué regla de JSON se está rompiendo, (b) escribe la versión corregida, y (c) comprueba con `JSON.parse` que la versión corregida sí funciona.

Cadena A: `'{"nombre": "Pedro", "edad": 25,}'` Cadena B: `'{"ciudad": 'Lima', 'pais': 'Perú'}'` Cadena C: `'{"activo": true, "puntos": 100'`

Solución:

```
// Cadena A: el error es la coma colgando después de 25
// Regla rota: no se permiten comas al final en JSON
let corregidaA = '{"nombre": "Pedro", "edad": 25}';
console.log(JSON.parse(corregidaA)); // { nombre: "Pedro", edad: 25 }

// Cadena B: el error son las comillas simples
// Regla rota: JSON exige comillas DOBLES en claves y textos
let corregidaB = '{"ciudad": "Lima", "pais": "Perú"}';
console.log(JSON.parse(corregidaB)); // { ciudad: "Lima", pais: "Perú" }

// Cadena C: el error es que falta la llave de cierre }
// Regla rota: los objetos deben cerrarse correctamente
let corregidaC = '{"activo": true, "puntos": 100}';
console.log(JSON.parse(corregidaC)); // { activo: true, puntos: 100 }
```

Explicación paso a paso: La cadena A rompe la regla de "sin comas al final": esa coma antes de `}` es válida en JavaScript pero inválida en JSON, y produce el clásico `"Unexpected token"`. La cadena B usa comillas simples, que JavaScript acepta pero JSON no: todo en JSON lleva comillas dobles, sin excepción. La cadena C simplemente está incompleta: le falta cerrar el objeto, algo que pasa mucho cuando el texto se corta (por ejemplo, una respuesta de red interrumpida). El ejercicio te entrena el ojo para los tres errores más comunes; cuando veas `"Unexpected token"` en tu consola, tu primer instinto ya será revisar comas, comillas y llaves.

Ejercicio 3: Crea una promesa que a veces falla

Enunciado: Escribe una función llamada `simularDescarga` que devuelva una promesa. La promesa debe esperar 1.5 segundos (con `setTimeout`) y luego: si un número aleatorio (`Math.random()`) es mayor que 0.3, cumplir la promesa con el texto "Descarga completa"; si no, rechazarla con un `Error("Conexión interrumpida")`. Luego usa la promesa con `then`, `catch` y `finally`, imprimiendo mensajes adecuados en cada caso. Ejecuta el código varias veces para ver ambos caminos.

Solución:

```
function simularDescarga() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      if (Math.random() > 0.3) {
        resolve("Descarga completa");
      } else {
        reject(new Error("Conexión interrumpida"));
      }
    }, 1500);
  });
}

simularDescarga()
  .then(function(mensaje) {
    console.log("Éxito:", mensaje);
  })
  .catch(function(error) {
    console.log("Falló:", error.message);
  })
  .finally(function() {
    console.log("Intento terminado");
  });
```

Explicación paso a paso: La función devuelve `new Promise` con un ejecutor que programa un `setTimeout` de 1.5 segundos. Dentro del temporizador, `Math.random()` decide el destino: el 70% de las veces se cumple con `resolve`, el 30% se rechaza con `reject`. Como la decisión es aleatoria, cada ejecución puede tomar un camino distinto: corre el código varias veces y verás a veces "Éxito" y a veces "Falló". El `then` maneja el éxito, el `catch` maneja el fallo, y el `finally` se ejecuta siempre. Este ejercicio simula la realidad de la red: a veces funciona, a veces no, y tu código debe estar listo para ambas. Nota que devolvemos la promesa con `return` desde la función: sin ese `return`, quien llame a `simularDescarga` no podría encadenar `then`.

Ejercicio 4: Encadena dos operaciones dependientes

Enunciado: Crea dos funciones que devuelvan promesas: `obtenerUsuario()` que después de 1 segundo cumple con `{id: 5, nombre: "Sofía"}`, y `obtenerMascotas(usuarioId)` que después de 1 segundo cumple con el arreglo `["Firulais", "Michi"]` (ignora el id, solo simula). Encadena ambas con `then` de modo que: primero obtengas el usuario, luego (usando su id) obtengas sus mascotas, y al final imprimas "Sofía tiene 2 mascotas: Firulais, Michi". Agrega un `catch` al final. Ojo: no olvides el `return` dentro del primer `then`.

Solución:

```
function obtenerUsuario() {
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve({ id: 5, nombre: "Sofía" });
    }, 1000);
  });
}

function obtenerMascotas(usuarioId) {
  return new Promise(function(resolve) {
    setTimeout(function() {
      resolve(["Firulais", "Michi"]);
    }, 1000);
  });
}

obtenerUsuario()
  .then(function(usuario) {
    console.log("Usuario:", usuario.nombre);
    return obtenerMascotas(usuario.id); // sin este return, la cadena no espera
  })
  .then(function(mascotas) {
    console.log("Sofía tiene " + mascotas.length + " mascotas: " +
mascotas.join(", "));
  })
  .catch(function(error) {
    console.log("Error en la cadena:", error.message);
  });
```

Explicación paso a paso: Cada función simula una operación que tarda 1 segundo y devuelve su promesa con `return`. En la cadena, el primer `then` recibe el usuario e inicia la segunda operación, devolviéndola con `return` para que la cadena espere su resultado. El segundo `then`

recibe el arreglo de mascotas y construye el mensaje final con `length` y `join`. El `catch` al final atraparía cualquier fallo en cualquier punto. El punto crítico del ejercicio es el `return`: quítalo y el segundo `then` se ejecuta de inmediato con `undefined`, imprimiendo un mensaje roto. Este patrón (pedir A, con el resultado pedir B) es exactamente lo que harás con APIs reales: pedir el usuario y luego sus publicaciones, pedir la ciudad y luego su clima.

Ejercicio 5: fetch real con manejo completo de errores

Enunciado: Escribe código que pida los datos del usuario con id 3 a la API

<https://jsonplaceholder.typicode.com/users/3>. Verifica `respuesta.ok` y lanza un error si no es exitosa. Convierte la respuesta a JSON y muestra en consola el nombre, el email y la ciudad (viene en `address.city`) del usuario. Maneja los errores con `catch` mostrando un mensaje amable. Después, modifica la URL para pedir el usuario 999 (que no existe) y observa cómo tu código maneja el 404.

Solución:

```
function cargarUsuario(id) {
  fetch("https://jsonplaceholder.typicode.com/users/" + id)
    .then(function(respuesta) {
      if (!respuesta.ok) {
        throw new Error("El servidor respondió con código " + respuesta.status);
      }
      return respuesta.json();
    })
    .then(function(usuario) {
      console.log("Nombre:", usuario.name);
      console.log("Email:", usuario.email);
      console.log("Ciudad:", usuario.address.city);
    })
    .catch(function(error) {
      console.log("No se pudo cargar el usuario. Intenta más tarde.");
      console.log("Detalle técnico:", error.message);
    });
}

cargarUsuario(3); // existe: muestra los datos
// cargarUsuario(999); // no existe: muestra el mensaje amable del catch
```

Explicación paso a paso: La función recibe el id y construye la URL. El primer `then` verifica `respuesta.ok`: si el servidor respondió con error (como el 404 del usuario 999), lanzamos un

Error con throw, lo que manda la cadena directo al catch sin pasar por el segundo then. Si todo va bien, respuesta.json() convierte el cuerpo y el segundo then recibe el objeto usuario, del cual leemos name, email y la ciudad anidada en address.city (nota cómo navegamos un objeto dentro de otro, algo típico de APIs reales). El catch muestra primero un mensaje amable pensado para el usuario final y luego el detalle técnico para el programador. Al probar con 999 verás el camino del error en acción: sin la verificación de ok, el código habría intentado leer propiedades de una respuesta vacía y el fallo habría sido confuso.

Ejercicio 6 (reto): Lista de publicaciones con estado de carga

Enunciado: Crea una página con un botón "Cargar publicaciones", un párrafo para mensajes y una lista vacía. Al hacer clic: muestra "Cargando..." en el párrafo, pide <https://jsonplaceholder.typicode.com/posts>, verifica respuesta.ok, convierte a JSON y muestra en la lista los títulos de las primeras 5 publicaciones (cada uno en un li). Si algo falla, muestra en el párrafo "Error al cargar. Revisa tu conexión." Usa finally para escribir en consola "Petición finalizada".

Solución:

HTML:

```
<button id="btnPosts">Cargar publicaciones</button>
<p id="mensaje"></p>
<ul id="listaPosts"></ul>
```

JavaScript:

```

let btn = document.getElementById("btnPosts");
let mensaje = document.getElementById("mensaje");
let lista = document.getElementById("listaPosts");

btn.addEventListener("click", function() {
  mensaje.textContent = "Cargando...";
  lista.innerHTML = "";

  fetch("https://jsonplaceholder.typicode.com/posts")
    .then(function(respuesta) {
      if (!respuesta.ok) {
        throw new Error("Código " + respuesta.status);
      }
      return respuesta.json();
    })
    .then(function(posts) {
      mensaje.textContent = "Se encontraron " + posts.length + " publicaciones. Mostrando 5:";
      for (let i = 0; i < 5; i++) {
        let item = document.createElement("li");
        item.textContent = posts[i].title;
        lista.appendChild(item);
      }
    })
    .catch(function(error) {
      mensaje.textContent = "Error al cargar. Revisa tu conexión.";
      console.log("Detalle:", error.message);
    })
    .finally(function() {
      console.log("Petición finalizada");
    });
});

```

Explicación paso a paso: Este ejercicio junta todo lo de la unidad con el DOM de unidades anteriores. Al hacer clic, primero damos feedback inmediato ("Cargando...") y limpiamos resultados anteriores, dos detalles que separan una app profesional de una amateur. La cadena fetch sigue el patrón completo: verificación de ok con throw, conversión con respuesta.json(), y en el segundo then recorremos solo los primeros 5 con un for clásico, creando un li por cada título con createElement y appendChild (como aprendiste en la clase 21). El catch muestra el mensaje de error en la página (no solo en consola), y el finally registra en consola que la petición terminó, se haya cumplido o rechazado. Si lo lograste sin mirar la solución, ya sabes hacer lo esencial de cualquier app moderna: pedir datos y pintarlos.

Resumen de la unidad

JSON es un formato de texto para representar datos estructurados: el idioma con el que los servidores y tu código se comunican. Sus reglas son estrictas: claves y textos con comillas dobles, sin comas al final, sin comentarios, sin funciones, y solo admite textos, números, booleanos, null, arreglos y objetos.

JSON.stringify convierte un valor de JavaScript a texto JSON (lo usas para enviar datos o guardarlos). JSON.parse convierte texto JSON a un valor de JavaScript (lo usas para recibir datos). Parse es entrada, stringify es salida: no los apliques al revés ni dos veces sobre lo mismo.

Los errores de formato más comunes son la coma colgando al final, las comillas simples donde van dobles, y el texto incompleto. El mensaje "Unexpected token" casi siempre apunta a uno de esos tres. Cuando algo falle al parsear, imprime el texto crudo antes de buscar el bug en tu código.

Al convertir con stringify, las funciones y los undefined desaparecen en silencio y las fechas se vuelven texto. Del otro lado, al hacer parse, esas fechas llegan como texto, no como objetos Date. Tenlo presente cuando trabajes con APIs que mandan fechas.

Una promesa representa un valor futuro y vive en tres estados: pendiente (esperando), cumplida (éxito, con un valor) y rechazada (fallo, con un motivo). Solo cambia de estado una vez y nunca vuelve atrás.

Reaccionas a las promesas con then (se ejecuta si se cumple), catch (si se rechaza) y finally (siempre, para limpieza). Las reacciones se registran ahora y se ejecutan después: el código no se pausa, sigue avanzando mientras la operación termina en el fondo.

Puedes encadenar promesas porque then devuelve otra promesa: lo que devuelvas con return en un then lo recibe el siguiente then. Si olvidas el return al iniciar otra operación asíncrona, la cadena no espera y recibes undefined. Un solo catch al final de la cadena atrapa errores de cualquier punto.

Todo lo que dependa del valor de una promesa debe vivir dentro de un then (o en la cadena). No hay forma de "sacar" el valor al código síncrono: el código que necesita el dato tiene que esperar en el futuro también.

fetch(url) hace una petición GET y devuelve una promesa que se cumple con un objeto Response. Ese objeto no son los datos todavía: usa respuesta.json() (que devuelve otra promesa) para convertir el cuerpo de texto JSON a objetos de JavaScript.

fetch no rechaza su promesa ante errores HTTP como el 404 o el 500: considera que la red funcionó. Verifica siempre `respuesta.ok` y lanza un error con `throw` si no lo es, para que tu `catch` maneje el fallo. El `catch` sí atrapa errores de red reales (sin internet, servidor caído).

El patrón profesional completo es: `fetch` -> verificar `ok` -> convertir con `.json()` -> trabajar con los datos dentro del `then` -> `catch` con mensaje amable -> `finally` para limpieza. Úsalo siempre, incluso cuando creas que la API "nunca falla".

Para practicar con datos reales sin registro, usa la API gratuita JSONPlaceholder (jsonplaceholder.typicode.com): usuarios, publicaciones y comentarios falsos servidos de verdad por internet.

Nunca asumas la forma de los datos que te manda un servidor: cuando algo falle, imprime la respuesta cruda con `console.log` antes de operar con ella. Las APIs cambian, devuelven errores con formas distintas, y un vistazo al dato real te ahorra horas de depuración a ciegas.

El error de CORS no es un bug de tu código: es el navegador protegiéndote cuando un servidor no autoriza peticiones de otros orígenes. Si lo ves con una API privada, el permiso lo tiene que dar el servidor; con APIs públicas de práctica no te va a pasar.

Las promesas se crearon para reemplazar el "callback hell": código con callbacks anidados que se volvía una pirámide ilegible. La cadena vertical de `then` es su reemplazo moderno y legible; si ves código viejo con `XMLHttpRequest` o callbacks anidados, ahora sabes qué era y por qué `fetch` con promesas es mejor.

Y la idea que une todo: JSON es el idioma, las promesas son la paciencia para esperar sin bloquearse, y `fetch` es el mensajero. Con las tres piezas sabes pedir datos a un servidor, convertirlos y mostrarlos en tu página: eso es lo que hacen las aplicaciones web de verdad, y ahora tú también.

UNIDAD 10: Código moderno y tu primer proyecto

Llegaste a la última unidad del curso. Tómate un segundo para dimensionar lo que eso significa: empezaste sin saber qué era JavaScript y ahora sabes pedir datos a un servidor, convertirlos, encadenar promesas y pintar resultados en una página. Eso no es poco. Es, de hecho, el conjunto de habilidades con el que miles de personas consiguen su primer trabajo como desarrolladoras. Esta unidad tiene dos misiones. La primera: enseñarte `async/await`, la forma moderna de escribir código asíncrono, la sintaxis que vas a ver en casi todos los proyectos profesionales actuales. La segunda, y la más importante: guiarte en la construcción de tu primer proyecto completo desde cero, un contador funcional con botones de sumar, restar y reiniciar, explicado línea por línea. Y al final, haremos el cierre del curso: repasaremos el camino recorrido, hablaremos de qué sigue, y te diré exactamente cómo continuar tu formación en lucasegaacademy.com.

Empecemos por `async/await`, porque es la evolución natural de todo lo que aprendiste en la unidad 9. Las promesas con `then/catch` funcionan, pero tienen un costo: el código se escribe en vertical, con callbacks anidados, y cuando la lógica se complica (condicionales dentro de la cadena, ciclos que esperan, varios pasos con decisiones), las cadenas de `then` se vuelven difíciles de seguir. Los creadores de JavaScript lo notaron y agregaron una sintaxis que te deja escribir código asíncrono como si fuera síncrono: lees de arriba hacia abajo, en orden, sin callbacks. Eso es `async/await`. No es magia nueva por debajo: son promesas con un traje más elegante. Pero ese traje cambia por completo la legibilidad de tu código, y en el mundo profesional la legibilidad vale oro.

Mira la diferencia con un ejemplo rápido. Con promesas clásicas:

```
fetch("https://jsonplaceholder.typicode.com/users/1")
  .then(function(respuesta) { return respuesta.json(); })
  .then(function(usuario) { console.log(usuario.name); })
  .catch(function(error) { console.log(error); });
```

Con `async/await`:

```
async function mostrarUsuario() {
  try {
    let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/1");
    let usuario = await respuesta.json();
    console.log(usuario.name);
  } catch (error) {
    console.log(error);
  }
}
```

¿Ves? El segundo se lee como una receta de cocina: haz esto, luego esto, luego esto. No hay callbacks, no hay anidación. Cada await dice "espera aquí a que termine esta promesa, y luego sigue con la siguiente línea". Es el mismo comportamiento que las promesas, pero escrito de la forma en que tu cerebro ya piensa. En esta unidad vas a dominar esta sintaxis: funciones async, cómo await pausa sin bloquear, y cómo manejar errores con try/catch.

La segunda misión de la unidad es el proyecto guiado: un contador. Podrías pensar "¿un contador? ¿eso es todo?". Y te entiendo, suena modesto. Pero escúchame: un contador bien hecho, explicado línea por línea, con HTML completo y JavaScript completo, que tú entiendes al cien por ciento, vale más que un proyecto "impresionante" copiado de internet que no entiendes. El contador usa todo lo fundamental: variables para el estado, DOM para mostrar el número, eventos para los botones, condicionales para las reglas, funciones para organizar. Es el "hola mundo" de las aplicaciones interactivas: pequeño, completo y honesto. Cuando lo termines, no solo tendrás un proyecto funcionando: tendrás la plantilla mental para construir cualquier app interactiva, porque todas siguen el mismo patrón: estado -> mostrar -> escuchar -> actualizar -> volver a mostrar.

Y luego viene el cierre. Vamos a repasar las 30 clases, a conectar los puntos entre unidades, a hablar de qué sigue: cómo practicar con proyectos, qué son los frameworks y cuándo aprenderlos, y cómo seguir tu camino en lucasegaacademy.com, donde te esperan más cursos. Graduarte de este curso no significa que ya lo sabes todo; significa que tienes los cimientos sólidos para construir lo que sea. Los cimientos no se ven en el edificio terminado, pero sin ellos todo se cae. Tú ya tienes los tuyos.

Un consejo para esta última unidad: baja la velocidad. Sé que quieres terminar, sé que la meta está cerca, pero `async/await` merece que lo entiendas de verdad, no que lo memorices. Y el proyecto merece que lo escribas tú, línea por línea, no que copies y pegues. La diferencia entre "terminé el curso" y "aprendí del curso" está en estas últimas horas. Haz los ejercicios, rompe el contador a propósito y arréglalo, cambia los textos, agrega un botón extra por tu cuenta. El curso termina, pero tu práctica no.

Explicación detallada

`async/await`: las promesas con traje elegante

Para entender `async/await` necesitas una sola idea central: es azúcar sintáctica sobre las promesas. "Azúcar sintáctica" es el término técnico para "una forma más dulce de escribir lo mismo". Por debajo, todo sigue siendo promesas: los mismos estados, las mismas reglas, el mismo comportamiento. Lo único que cambia es cómo se ve tu código. Y ese cambio visual es tan grande que la industria entera migró a esta sintaxis.

Empecemos con `async`. Cuando marcas una función con la palabra `async`, pasan dos cosas. Primera: la función siempre devuelve una promesa, aunque su código devuelva un valor normal. Si tu función `async` devuelve 42, quien la llame recibe una promesa cumplida con el valor 42. Segunda: dentro de esa función puedes usar la palabra `await`. Mira:

```
async function saludar() {  
  return "Hola";  
}  
  
saludar().then(function(mensaje) {  
  console.log(mensaje); // "Hola"  
});
```

La función devuelve "Hola", pero como es `async`, lo que realmente entrega es una promesa cumplida con "Hola". Por eso podemos usar `.then` sobre su resultado. Esto confunde al principio: "¿por qué devuelve una promesa si yo devolví un texto?". Respuesta: porque `async` lo promete en su contrato. Toda función `async` devuelve promesa, sin excepción. Si dentro lanzas un error (`throw`), la promesa se rechaza. Es el mismo modelo de las promesas, solo que expresado con `return` y `throw` normales.

Ahora `await`, la estrella del show. `await` solo puede usarse dentro de una función `async` (regla inquebrantable, salvo en módulos, pero no te preocupes por eso ahora). Lo que hace es: toma

una promesa, pausa la ejecución de la función hasta que la promesa se cumple, y luego continúa con la siguiente línea, entregándote el valor directamente. Mira la diferencia mental:

Con `then`, tú le dices a la promesa: "cuando termines, ejecuta esta función". Con `await`, tú le dices a tu propio código: "detente aquí hasta que esto termine, y dame el resultado en esta misma línea".

```
async function ejemplo() {  
  console.log("Pido los datos...");  
  let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/1");  
  console.log("Ya llegaron los encabezados");  
  let usuario = await respuesta.json();  
  console.log("Ya tengo al usuario:", usuario.name);  
  console.log("Sigo con mi vida");  
}
```

Léelo en voz alta: pido los datos, ESPERO la respuesta, aviso que llegaron los encabezados, ESPERO la conversión, muestro al usuario, sigo. Es lineal. Es como lo pensarías si lo explicarás a un colega. Compáralo con la versión de `then` de la introducción y dime cuál preferirías leer a las 2 de la mañana cuando algo falla en producción. Por eso la industria migró.

Aquí viene la pregunta que todo el mundo hace y que es clave: "si `await` pausa, ¿no estamos volviendo a la espera bloqueante? ¿No se congela la página?". Y la respuesta es lo más bonito del diseño: NO. `await` pausa solo la función `async`, no todo el programa. Mientras tu función espera, el resto de JavaScript sigue funcionando: otros clics se atienden, otras funciones corren, la página sigue viva. Espera sin bloquear. ¿Cómo es posible? Porque por debajo sigue siendo el modelo de promesas y el event loop: la función se "suspende" y el motor aprovecha para hacer otras cosas, y cuando la promesa se cumple, la función "despierta" donde se quedó. Es como pausar una película: la película se detiene, pero el resto de la casa sigue funcionando. Solo pausaste esa historia, no el mundo.

Analogía de la vida real: imagina que eres chef y tienes ayudantes. Con `then` (callbacks), le dices a tu ayudante: "cuando hierva el agua, avísame y yo hago X". Tú sigues con otras cosas y cuando te avisa, interrumpes lo que hacías para hacer X. Con `await`, es como si pudieras decir "voy a esperar a que hierva el agua" y, mientras tanto, tu cuerpo hiciera otras cosas en automático y tu atención volviera justo cuando el agua hierve, continuando exactamente donde te quedaste. La segunda forma se parece más a cómo pensamos naturalmente: "primero esto, luego esto".

Vamos con las reglas y los errores típicos, porque aquí es donde la sintaxis bonita muerde si no la respetas.

Error típico uno: usar await fuera de una función async. Esto truena de inmediato:

```
// MAL: await fuera de async
let datos = await fetch(url); // SyntaxError
```

JavaScript te dice "await is only valid in async functions". La solución es envolver tu código en una función async. Este error es tan común que los programadores lo cometen incluso con experiencia, cuando pegan código de un ejemplo dentro de otro contexto. Si ves ese error, revisa: ¿estoy dentro de una función marcada con async?

Error típico dos: olvidar el await. Si llamas a una función async (o a fetch) sin await, no obtienes el valor: obtienes la promesa pendiente.

```
// MAL: sin await, usuario es una promesa, no el usuario
async function mostrar() {
  let usuario = fetch("https://jsonplaceholder.typicode.com/users/1");
  console.log(usuario.name); // undefined: usuario es una promesa pendiente
}

// BIEN: con await, usuario es el valor ya resuelto
async function mostrar() {
  let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/1");
  let usuario = await respuesta.json();
  console.log(usuario.name); // nombre real
}
```

El síntoma clásico del await olvidado es ver "Promise {<pending>}" en la consola, o undefined donde esperabas un dato. Cuando eso pase, busca el await faltante. Es el equivalente moderno del "olvidé el return en el then" de la unidad 9: el mismo tipo de olvido, en la sintaxis nueva.

Error típico tres: pensar que await hace todo secuencial cuando quieres paralelismo. Mira:

```
// Secuencial: tarda la SUMA de los tiempos (2 + 2 = 4 segundos)
async function secuencial() {
  let a = await operacionLenta1(); // espera 2s
  let b = await operacionLenta2(); // espera otros 2s
}

// Paralelo: tarda el MÁXIMO de los tiempos (2 segundos)
async function paralelo() {
  let promesaA = operacionLenta1(); // inicia sin esperar
  let promesaB = operacionLenta2(); // inicia sin esperar
  let a = await promesaA; // espera a que ambas terminen
  let b = await promesaB;
}
```

Si las dos operaciones son independientes (no necesitas el resultado de una para iniciar la otra), inícialas sin await y espera después: se ejecutan en paralelo y tu código tarda la mitad. Si son dependientes (necesitas el usuario antes de pedir sus pedidos), el await secuencial es lo correcto. Saber cuándo secuenciar y cuándo paralelizar es una habilidad de programador senior; ahora ya conoces la técnica.

Cuidado con... el try/catch con await, que es OBLIGATORIO en código serio. Cuando una promesa esperada con await se rechaza, el await lanza el error como si fuera un throw normal, y si no hay try/catch, la función async devuelve una promesa rechazada que nadie maneja. El patrón estándar es:

```
async function cargarDatos() {
  try {
    let respuesta = await fetch(url);
    if (!respuesta.ok) {
      throw new Error("Error del servidor: " + respuesta.status);
    }
    let datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.log("Falló la carga:", error.message);
  }
}
```

El try envuelve todo lo riesgoso; si CUALQUIER await falla (la red, el servidor, la conversión), el control salta al catch. Es más limpio que el .catch() de las cadenas porque puedes usar un solo try/catch para muchas líneas, con condicionales y ciclos adentro. De hecho, esa es otra

ventaja de `async/await`: puedes mezclar código asíncrono con `if`, `for` y `while` de forma natural, cosa que con `then` era un dolor de cabeza.

¿Sabías que? `async/await` llegó a JavaScript en 2017 (ES2017), pero la idea es más vieja: C# lo tenía desde 2012 y Python lo adoptó también. Cuando un concepto aparece en varios lenguajes, es señal de que resuelve un problema real y profundo. Aprender `async/await` no solo te sirve para JavaScript: te entrena un modelo mental que te servirá en otros lenguajes.

Pregunta frecuente: "¿Debo usar `then` o `async/await`?" En código nuevo, `async/await` casi siempre. Es más legible, maneja errores con `try/catch` natural, y se mezcla bien con el resto del lenguaje. `then` sigue siendo útil para casos puntuales (reaccionar a una promesa sin crear función `async`, o encadenamientos muy simples de una línea). Lo importante es que entiendas ambos, porque vas a leer código viejo con `then` toda tu vida. Pero para escribir, prefiere `async/await`.

Pregunta frecuente: "¿`await` funciona con cualquier promesa, o solo con `fetch`?" Con cualquiera. `await` espera a cualquier promesa: las que devuelve `fetch`, las que tú creas con `new Promise`, las que devuelven funciones `async`. Incluso si le pasas un valor que NO es promesa, `await` lo envuelve y lo devuelve tal cual (`await 42` devuelve 42). Es universal.

Otra pregunta frecuente: "¿Puedo usar `await` en un ciclo `for`?" Sí, y es uno de los superpoderes de `async/await`. Con `then`, hacer "para cada elemento, espera una operación" era un enredo. Con `await`, es natural:

```
async function procesar(lista) {  
  for (let item of lista) {  
    let resultado = await operacionLenta(item);  
    console.log(resultado);  
  }  
}
```

Esto procesa los elementos uno por uno, esperando cada operación. Claro, legible y correcto. (Si quisieras procesarlos en paralelo, usarías el truco de iniciar todas las promesas primero, como vimos arriba.)

El proyecto: un contador completo, línea por línea

Ahora la segunda misión: tu primer proyecto guiado completo. Vamos a construir un contador con tres botones: sumar (+1), restar (-1) y reiniciar (volver a 0). Lo haremos con el HTML completo y el JavaScript completo, explicado línea por línea. Nada de "y aquí pasa la magia": cada línea tiene su porqué.

Antes del código, el diseño mental. Todo proyecto interactivo, del más simple al más complejo, sigue el mismo ciclo: (1) tienes un ESTADO (datos en memoria), (2) MUESTRAS ese estado en pantalla, (3) ESCUCHAS acciones del usuario, (4) ACTUALIZAS el estado según la acción, (5) vuelves a MOSTRAR. Estado -> mostrar -> escuchar -> actualizar -> mostrar. El contador es ese ciclo en su forma más pura: el estado es un número, mostrarlo es ponerlo en un elemento, escuchar es atender clics, actualizar es sumar/restar/reiniciar. Cuando entiendas este ciclo en el contador, lo reconocerás en un carrito de compras, en un juego, en un formulario: siempre es lo mismo, solo cambia la complejidad del estado.

El HTML completo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mi contador</title>
</head>
<body>
  <h1>Contador</h1>
  <p id="numero">0</p>
  <button id="btnSumar">Sumar</button>
  <button id="btnRestar">Restar</button>
  <button id="btnReiniciar">Reiniciar</button>
  <script src="contador.js"></script>
</body>
</html>
```

Línea por línea del HTML: DOCTYPE y html/head/body son la estructura básica que ya conoces. El h1 es el título visible. El p con id="numero" es donde se muestra el valor; empieza en 0 porque el contador inicia en cero. Los tres botones tienen ids únicos para poder seleccionarlos desde JavaScript. El script al final del body carga contador.js: va al final para que cuando el JavaScript se ejecute, los elementos ya existan en el DOM (si lo pusieras en el head, getElementById no encontraría nada porque la página aún no se construye; este detalle muerde a muchos principiantes).

El JavaScript completo (contador.js):

```

// 1. ESTADO: el número que el contador recuerda
let cuenta = 0;

// 2. SELECCIÓN: agarramos los elementos del DOM una sola vez
let numero = document.getElementById("numero");
let btnSumar = document.getElementById("btnSumar");
let btnRestar = document.getElementById("btnRestar");
let btnReiniciar = document.getElementById("btnReiniciar");

// 3. MOSTRAR: función que pinta el estado actual en pantalla
function mostrar() {
    numero.textContent = cuenta;
}

// 4. ESCUCHAR Y ACTUALIZAR: cada botón cambia el estado y vuelve a mostrar
btnSumar.addEventListener("click", function() {
    cuenta = cuenta + 1;
    mostrar();
});

btnRestar.addEventListener("click", function() {
    cuenta = cuenta - 1;
    mostrar();
});

btnReiniciar.addEventListener("click", function() {
    cuenta = 0;
    mostrar();
});

// 5. Mostrar el valor inicial al cargar la página
mostrar();

```

Vamos línea por línea del JavaScript, sin prisa.

Línea "let cuenta = 0": declaramos el estado. Es una variable normal, con let porque va a cambiar. Vive fuera de las funciones para que todos los botones la compartan: es el "cerebro" del contador, el único lugar donde vive la verdad del valor actual. Todo lo demás (lo que se ve en pantalla) es un reflejo de esta variable. Este concepto, "una sola fuente de verdad", es fundamental en programación: el estado manda, la pantalla obedece.

Líneas de getElementById: seleccionamos los cuatro elementos que necesitamos y los guardamos en variables. Lo hacemos UNA vez, al inicio, no dentro de cada clic. ¿Por qué?

Porque buscar elementos en el DOM cuesta trabajo; si lo hicieras en cada clic, funcionarían igual, pero estarías repitiendo búsquedas innecesarias. Guardarlos una vez es más eficiente y más limpio. Es una buena práctica que debes llevar a todos tus proyectos.

Función `mostrar()`: toma el valor de cuenta y lo pone como texto del párrafo. Es pequeña a propósito: su único trabajo es sincronizar la pantalla con el estado. Cada vez que el estado cambia, llamamos a `mostrar()`. Esta separación (una función dedicada a pintar) es lo que hace que el código sea fácil de mantener: si mañana quieres que el número se vea más grande o cambie de color, solo tocas esta función.

Los tres `addEventListener`: cada botón escucha clics. Cuando el usuario hace clic en Sumar, la función suma 1 a cuenta y llama a `mostrar()` para reflejarlo. Restar resta 1. Reiniciar pone cuenta en 0. Nota el patrón idéntico en los tres: actualizar estado -> mostrar. Ese patrón es el ciclo del que hablamos. Si lo ves repetido tres veces, ya lo tienes grabado.

La llamada final `mostrar()`: pinta el valor inicial (0) cuando la página carga. Sin esto, el párrafo ya dice 0 por el HTML, así que parecería innecesario; pero es una buena costumbre: la pantalla siempre debe reflejar el estado al inicio, no depender de que el HTML inicial coincida. En proyectos grandes, el estado inicial viene de un servidor o de un cálculo, y esta llamada es la que lo pinta.

Ahora, mejoras que puedes hacer tú para practicar (mini-retos del proyecto):

Mini-reto 1: evita que el contador baje de 0. En el botón Restar, solo resta si cuenta es mayor que 0. Es un condicional simple, de la clase 6, aplicado a un proyecto real.

Mini-reto 2: cambia el color del número según su valor: verde si es positivo, rojo si es negativo, negro si es cero. Usa `numero.style.color` dentro de `mostrar()`. Practicas el DOM de la clase 21.

Mini-reto 3: agrega un botón "Sumar 5" que suma de 5 en 5. Es copiar el patrón del botón Sumar con otro valor. Si puedes hacer esto sin mirar, entendiste el patrón.

Mini-reto 4: guarda el contador en `localStorage` para que no se pierda al recargar la página. Al iniciar, lee el valor guardado (con `JSON.parse`, ¡mira quién apareció!); cada vez que cambie, guárdalo (con `JSON.stringify`). Este reto conecta la unidad 9 con el proyecto: persistencia real de datos.

Mini-reto 5: muestra un mensaje cuando el contador llegue a 10: "¡Llegaste a 10!". Un condicional dentro de `mostrar()`. Pequeño, pero te enseña a reaccionar a estados específicos, que es la base de la lógica de juegos.

Cada mini-reto te toma unos minutos y cada uno te enseña algo. Haz al menos dos. El proyecto deja de ser "el ejemplo del libro" y se convierte en "mi proyecto" cuando le pones tus propias mejoras.

Cuidado con... los errores típicos del proyecto. Uno: olvidar llamar a `mostrar()` después de actualizar cuenta. El estado cambia pero la pantalla no, y parece que el botón "no hace nada". Si un botón no refleja cambios, revisa que llamaste a `mostrar()`. Dos: poner el script en el head sin esperar a que cargue el DOM: `getElementById` devuelve null y luego `numero.textContent` truena. Script al final del body, siempre (o usa el evento `DOMContentLoaded`, pero al final del body es más simple). Tres: usar el mismo id en dos elementos: `getElementById` agarra el primero y el segundo botón "no funciona". Los ids son únicos, como tu CURP: uno por elemento.

¿Sabías que? Este patrón (estado -> mostrar -> escuchar -> actualizar) es, en esencia, lo que hacen los frameworks modernos como React. React lo automatiza: tú describes cómo se ve cada estado y él se encarga de actualizar la pantalla. Pero por debajo, la idea es la misma que acabas de programar a mano. Cuando llegues a React, no estarás aprendiendo algo alienígena: estarás viendo una versión con esteroides de tu contador. Todo lo que aprendiste aquí se transfiere.

Cómo probar y depurar tu proyecto como un profesional

Escribir el código es la mitad del trabajo; la otra mitad es comprobar que funciona y encontrar por qué no cuando falla. Aquí tienes un método simple de pruebas para el contador (y para cualquier proyecto futuro), más los errores más comunes y cómo cazarlos.

El método de pruebas tiene cinco pasos. Paso uno: la prueba feliz. Abre la página, haz clic en Sumar tres veces y verifica que el número diga 3. Haz clic en Restar una vez: debe decir 2. Haz clic en Reiniciar: debe decir 0. Si esto funciona, la base está sana. Paso dos: la prueba de límites. Haz clic en Restar diez veces seguidas desde 0. ¿El contador muestra números negativos? Si tu diseño dice que no debe bajar de 0, ahí tienes un bug que arreglar con un condicional. Haz clic en Sumar cincuenta veces rápido: ¿el número se actualiza bien o se traba? Paso tres: la prueba de recarga. Suma 7, recarga la página: si implementaste `localStorage`, debe seguir diciendo 7; si no, vuelve a 0 (y eso está bien si así lo diseñaste, pero debe ser una decisión consciente, no un accidente). Paso cuatro: la prueba de consola. Abre la consola del navegador y recarga: ¿hay errores en rojo? Un proyecto "que funciona" pero con errores en consola es un proyecto con una bomba de tiempo. Paso cinco: la prueba del extraño. Pídele a alguien que use tu contador sin explicarle nada y observa dónde se confunde: si no entiende qué hace un botón, el problema es tu diseño, no su inteligencia.

Ahora los bugs clásicos del proyecto y su diagnóstico. Síntoma: "hago clic y no pasa nada". Diagnóstico paso a paso: primero abre la consola, ¿hay un error rojo? Si dice "Cannot read properties of null", tu `getElementById` no encontró el elemento: revisa que el id en el HTML sea idéntico al del JavaScript (mayúsculas incluidas, sin espacios). Si no hay error en consola, el listener no se está ejecutando o no está registrado: verifica que el script esté al final del body y que el id del botón coincida. Segundo síntoma: "el número cambia en la variable pero no en pantalla" (lo verificas con `console.log(cuenta)`). Diagnóstico: olvidaste llamar a `mostrar()` después de actualizar. Es el bug más común del patrón estado -> mostrar: el estado cambió, pero nadie le avisó a la pantalla. Tercer síntoma: "funciona la primera vez, pero después de recargar se rompe". Diagnóstico: probablemente el `localStorage` tiene un valor viejo con formato inesperado; abre las herramientas del navegador, revisa lo guardado y ajusta tu lectura inicial. Cuarto síntoma: "los botones funcionan pero el diseño se ve raro". Eso ya no es JavaScript, es CSS, y está fuera del alcance de este curso, pero que no te detenga: un proyecto feo que funciona vale más que un diseño bonito que no hace nada.

Una técnica de depuración que te servirá toda la vida: los `console.log` estratégicos. Cuando algo no funciona, imprime el estado en cada paso: `console.log("antes:", cuenta)` antes de actualizar, `console.log("después:", cuenta)` después. Verás exactamente dónde el valor deja de ser el esperado. Los programadores senior usan `console.log` todos los días; no es una herramienta de principiantes, es una herramienta de profesionales que no tienen tiempo para adivinar. Y cuando el bug esté resuelto, quita los `console.log` de depuración o déjalos solo donde aporten información útil: una consola limpia es señal de código cuidado.

Por último, el hábito más valioso: prueba en otro navegador. Tu código puede funcionar perfecto en Chrome y fallar en otro por detalles de compatibilidad. No necesitas probar en diez navegadores; con dos basta para atrapar la mayoría de los problemas. Y si tu proyecto usa `fetch`, Pruébalo también con la red lenta (las herramientas de desarrollo te dejan simular conexión lenta): verás cómo se comporta tu "Cargando..." y descubrirás si tu manejo de la espera es digno o si la página se siente rota durante dos segundos. Los usuarios reales tienen conexiones reales, no la fibra de tu casa.

Cierre del curso: el camino recorrido y lo que sigue

Respira. Llegaste al final del curso completo de JavaScript de Lucas EGA Academy. Vamos a repasar el camino, porque cuando ves todo junto, te das cuenta de cuánto avanzaste.

Empezaste en la unidad 1 preguntándote qué era JavaScript y cómo se incluía en una página. Aprendiste variables con `let` y `const`, y entendiste por qué `const` es tu amiga para valores que no cambian. En la unidad 2 dominaste los tipos de datos y los operadores: textos, números,

booleanos, aritmética, comparaciones y lógica. Ahí nació tu capacidad de tomar decisiones en código con `if/else`. En la unidad 3 llegaron los ciclos `for` y `while`, y con ellos tu primer superpoder real: hacer que la computadora repita trabajo por ti. En la unidad 4 las funciones te enseñaron a organizar: parámetros, `return`, y las funciones flecha que hoy ves en todo código moderno.

La unidad 5 te dio las estructuras de datos: arreglos y objetos, y los métodos para manipularlos (`forEach`, `map` y compañía). Ahí tu código empezó a manejar información de verdad, no solo números sueltos. En la unidad 6 el DOM te abrió la puerta a la página: seleccionar elementos, modificarlos, escuchar clics, leer formularios, y jugar con el tiempo con `setTimeout` y `setInterval`. Tu código dejó de vivir en la consola y empezó a vivir en el navegador. La unidad 7 (que en este libro se integra en el flujo) reforzó eventos y formularios; la unidad 8 te habría hablado de persistencia y buenas prácticas. En la unidad 9 rompiste la burbuja: JSON, promesas y `fetch` te conectaron con servidores de verdad. Y en esta unidad 10, `async/await` te dio la sintaxis moderna y el proyecto del contador te demostró que puedes construir una aplicación completa tú solo, desde cero, entendiendo cada línea.

Treinta clases. Diez unidades. Un lenguaje completo en tus manos. Si hoy te sientas frente a una página en blanco, ya no ves un vacío: ves variables, funciones, eventos, promesas y un plan. Esa transformación es el verdadero graduado del curso, más que cualquier certificado.

Ahora, ¿qué sigue? Te lo digo directo, como colega:

Primero, practica con proyectos. El conocimiento sin proyectos se oxida. Ideas en orden de dificultad: una lista de tareas (agregar, tachar, borrar), una calculadora, un juego de adivinar el número, una página que muestre el clima usando una API real con `fetch`, un quiz de preguntas con puntaje. Cada proyecto te va a doler un poco y te va a enseñar mucho. Esa es la fórmula.

Segundo, profundiza en lo que más te costó. Si las promesas te costaron, haz tres mini-proyectos solo con `fetch`. Si el DOM te costó, clona la interfaz de una página simple. Los programadores profesionales no son los que entendieron todo a la primera: son los que volvieron sobre lo difícil hasta dominarlo.

Tercero, cuando domines los proyectos pequeños, aprende un framework. React es la apuesta más segura hoy: lo piden en la mayoría de los trabajos y su filosofía (componentes, estado) es la evolución natural de lo que aprendiste aquí. Pero no corras: un framework sin bases sólidas de JavaScript es una casa sin cimientos. Tú ya tienes los cimientos; el framework será el siguiente piso, no un parche.

Cuarto, aprende a leer documentación y a buscar errores. La mitad del trabajo profesional es leer docs (MDN es tu biblia: developer.mozilla.org) y buscar soluciones. No es trampa: es la habilidad. Los mejores programadores no memorizan todo; saben encontrar respuestas rápido y distinguir las buenas de las malas.

Quinto, sigue en lucasegaacademy.com. Este curso de JavaScript es uno de los 23 cursos de la academia, y el camino natural sigue con HTML y CSS a fondo (si aún no los dominas), y luego con cursos de desarrollo web completo, donde unirás todo para construir sitios reales. La academia está diseñada como ecosistema: cada curso conecta con los demás, y el "telaraña" de contenido te lleva de principiante a desarrollador completo. Visita lucasegaacademy.com, explora los cursos gratuitos, y sigue construyendo.

Y un último consejo, personal: guarda tus proyectos. Súbelos a GitHub cuando aprendas Git (que debería ser pronto). Dentro de un año vas a volver a ver tu contador y te vas a reír de lo simple que era, y esa risa será la medida exacta de cuánto creciste. Todos los programadores tenemos ese primer proyecto que nos da ternura. El tuyo ya existe. Felicidades: terminaste el curso. Ahora empieza lo bueno.

Más errores típicos, preguntas frecuentes y anécdotas de `async/await`

Error típico cuatro: el `try/catch` que no envuelve lo suficiente. Mira este caso:

```
async function cargar() {
  let respuesta = await fetch(url); // si esto falla, ¿quién lo atrapa?
  try {
    let datos = await respuesta.json();
    console.log(datos);
  } catch (e) {
    console.log("Error:", e.message);
  }
}
```

El `try` solo protege la conversión a JSON, pero el `fetch` está fuera: si la red falla, el error no lo atrapa nadie y la función devuelve una promesa rechazada sin manejar. En la consola verás la advertencia de "unhandled promise rejection", que es JavaScript diciéndote "algo falló y nadie se hizo responsable". La regla es simple: el `try` debe envolver TODAS las líneas riesgosas, desde el primer `await` hasta el último. Cuando escribas una función `async`, primero escribe el `try/catch` vacío y luego mete el código adentro. Ese hábito te ahorra la advertencia más confusa del lenguaje.

Error típico cinco: devolver el resultado de await innecesariamente enredado. Esto funciona pero es feo:

```
async function obtener() {  
  let datos = await fetch(url).then(function(r) { return r.json(); });  
  return datos;  
}
```

Mezclar await con then en la misma línea funciona, pero combina dos estilos y confunde a quien lo lee. Elige uno: o todo then, o todo await. La consistencia de estilo no es vanidad: cuando un equipo entero escribe igual, los errores se detectan más rápido porque el ojo sabe qué esperar.

Cuidado con... las funciones async en eventos y su "silencio". Cuando pones una función async como listener de un clic y algo falla sin try/catch, el error se pierde en la promesa rechazada y el usuario solo ve que "el botón no hizo nada". Siempre envuelve el contenido de un listener async en try/catch y muestra algo en la página cuando falle. El usuario no tiene la consola abierta: si no le dices qué pasó, para él tu app está rota.

¿Sabías que? La advertencia "unhandled promise rejection" existe porque en los primeros años de las promesas, los errores se perdían en silencio total y nadie sabía por qué su app fallaba. Los navegadores agregaron la advertencia como red de seguridad. Si la ves, no la ignores: es el lenguaje haciéndote un favor, señalándote exactamente la promesa que nadie atendió.

Pregunta frecuente: "¿async/await es más lento que then?" No, de forma relevante. Por debajo es el mismo mecanismo; la diferencia de rendimiento es insignificante frente al tiempo que tarda la red o el disco. La legibilidad que ganas vale infinitamente más que los microsegundos que podrías perder. Los programadores senior eligen el código que se entiende, no el que ahorra nanosegundos invisibles.

Pregunta frecuente: "¿Puedo usar await con setTimeout directamente?" setTimeout no devuelve una promesa, así que await setTimeout(...) no espera nada útil. Pero puedes envolverlo, como hicimos en los ejemplos: await new Promise(function(resolve) { setTimeout(resolve, 1000);}). Ese truco (una promesa que se cumple cuando termina el temporizador) es tan común que muchos proyectos definen una función esperar(ms) reutilizable. Es un buen ejemplo de cómo las promesas te dejan convertir cualquier cosa "que tarda" en algo "esperable".

Anécdota real de esas que enseñan: un desarrollador escribió un ciclo que pedía 50 usuarios a una API, uno por uno con `await`, y tardaba casi un minuto. Alguien le mostró el truco del paralelo (iniciar las 50 promesas y esperarlas juntas) y bajó a 3 segundos. Misma cantidad de trabajo, veinte veces más rápido, cambiando solo DÓNDE ponía los `await`. La moraleja no es "el paralelo siempre gana" (cuidado: 50 peticiones juntas pueden saturar un servidor o toparse con límites de la API), sino que entender dónde esperan tus `awaits` te da control real sobre el rendimiento. El código asíncrono bien pensado no solo funciona: vuela.

Otra anécdota: un estudiante me dijo "async/await me parece trampa, es como si el código mintiera diciendo que es síncrono". Y tiene un punto: la sintaxis OCULTA la complejidad. Por eso esta unidad insistió tanto en lo que pasa por debajo. La regla de oro es: usa la sintaxis dulce, pero piensa en promesas. Cuando algo se comporte raro, pregúntate "¿qué promesa está pendiente aquí? ¿dónde está mi `await`? ¿quién atrapa el rechazo?". Ese pensamiento te saca de cualquier atolladero, porque la sintaxis bonita nunca deja de ser promesas por dentro.

Y un último "cuidado con..." para cerrar la explicación: no conviertas TODO a `async` por moda. Hay código que no necesita esperar nada, y marcarlo `async` solo porque "se ve moderno" agrega promesas innecesarias y confunde. `async` es para funciones que esperan operaciones asíncronas. Si tu función solo suma dos números, que sea una función normal. La elegancia está en usar la herramienta correcta, no la más nueva.

Ejemplos paso a paso

Ejemplo 1: De `then` a `async/await` (la misma operación, dos estilos)

Vamos a ver la misma tarea escrita en los dos estilos, para que sientas la diferencia en tus propias manos.

```

// ESTILO CLÁSICO: cadena de promesas con then
function cargarUsuarioClasico() {
  fetch("https://jsonplaceholder.typicode.com/users/1")
    .then(function(respuesta) {
      return respuesta.json();
    })
    .then(function(usuario) {
      console.log("Clásico:", usuario.name);
    })
    .catch(function(error) {
      console.log("Error:", error.message);
    });
}

// ESTILO MODERNO: async/await, se lee de arriba hacia abajo
async function cargarUsuarioModerno() {
  try {
    let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/1");
    let usuario = await respuesta.json();
    console.log("Moderno:", usuario.name);
  } catch (error) {
    console.log("Error:", error.message);
  }
}

cargarUsuarioClasico();
cargarUsuarioModerno();

```

Las dos hacen exactamente lo mismo: piden el usuario 1, lo convierten a objeto y muestran su nombre. Pero la versión moderna no tiene callbacks anidados: cada paso es una línea, en orden. El try/catch envuelve todo y atrapa cualquier fallo de cualquier línea. Prueba ejecutar ambas y confirma que imprimen lo mismo. Luego pregúntate cuál preferirías depurar.

Ejemplo 2: await pausa la función, no el programa

Este ejemplo demuestra la idea más importante de await: la espera no bloquea.

```

async function tareaLenta() {
  console.log("A: Empiezo la tarea lenta");
  // Simulamos una operación que tarda 3 segundos
  await new Promise(function(resolve) {
    setTimeout(resolve, 3000);
  });
  console.log("C: Terminé la tarea lenta (3 segundos después)");
}

console.log("1: Antes de llamar a la tarea");
tareaLenta(); // la llamamos SIN await: no esperamos a que termine
console.log("2: Después de llamar a la tarea (esto sale de inmediato)");
console.log("B: El programa sigue vivo mientras la tarea espera");

```

El orden de salida será: 1, A, 2, B, y tres segundos después C. Fíjate: la función `tareaLenta` se pausó en el `await`, pero el programa siguió con "2" y "B" sin esperar. La pausa fue local a la función. Si hubiéramos escrito `await tareaLenta()`, entonces sí habríamos esperado a que terminara antes de seguir. Tú decides cuándo esperar y cuándo no, con o sin `await` al llamar. Ese control fino es poderosísimo.

Ejemplo 3: El `await` olvidado (el error que todos cometemos)

Vamos a provocar el error clásico para reconocerlo al instante cuando te pase de verdad.

```

async function ejemploRoto() {
  // OJO: falta el await antes de fetch
  let respuesta = fetch("https://jsonplaceholder.typicode.com/users/1");
  console.log("¿Qué es respuesta?", respuesta);
  // Muestra: Promise { <pending> } -> ¡es una promesa, no la respuesta!
  console.log("¿Y respuesta.ok?", respuesta.ok);
  // undefined -> las promesas no tienen .ok
}

async function ejemploArreglado() {
  let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/1");
  console.log("¿Qué es respuesta?", respuesta.ok);
  // true -> ahora sí es el objeto Response
  let usuario = await respuesta.json();
  console.log("Nombre:", usuario.name);
}

```

Cuando veas "Promise {<pending>}" en tu consola donde esperabas un dato, ya sabes el diagnóstico: falta un `await`. Es el equivalente moderno del `then` sin `return` de la unidad 9. Con el tiempo, tu ojo lo detectará en segundos.

Ejemplo 4: Secuencial vs paralelo (cuándo esperar y cuándo no)

Dos operaciones independientes: pedir dos usuarios. Versión lenta y versión rápida.

```
// VERSIÓN SECUENCIAL: espera una, luego la otra (tarda la suma)
async function secuencial() {
  console.log("Secuencial: inicio");
  let r1 = await fetch("https://jsonplaceholder.typicode.com/users/1");
  let u1 = await r1.json(); // esperamos a terminar con el usuario 1...
  let r2 = await fetch("https://jsonplaceholder.typicode.com/users/2");
  let u2 = await r2.json(); // ...y luego pedimos el usuario 2
  console.log("Secuencial:", u1.name, "y", u2.name);
}

// VERSIÓN PARALELA: inicia ambas y espera al final (tarda el máximo)
async function paralelo() {
  console.log("Paralelo: inicio");
  let promesa1 = fetch("https://jsonplaceholder.typicode.com/users/1");
  let promesa2 = fetch("https://jsonplaceholder.typicode.com/users/2");
  // Las dos peticiones viajan al mismo tiempo por internet
  let u1 = await (await promesa1).json();
  let u2 = await (await promesa2).json();
  console.log("Paralelo:", u1.name, "y", u2.name);
}
```

La versión paralela es notablemente más rápida porque las dos peticiones viajan juntas. Regla práctica: si la operación B no necesita el resultado de la A, inícialas juntas. Si B necesita el resultado de A (como pedir los pedidos de un usuario), el orden secuencial es obligatorio. Nota el doble `await` en la versión paralela: `await promesa1` espera la respuesta, y el segundo `await` espera la conversión a JSON. Se lee raro al principio, pero es correcto.

Ejemplo 5: try/catch con varias operaciones riesgosas

Un solo `try/catch` puede proteger una secuencia completa de operaciones, con lógica normal en medio.

```

async function reporteUsuario(id) {
  try {
    // Operación 1: pedir el usuario
    let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/" +
id);
    if (!respuesta.ok) {
      throw new Error("Usuario no encontrado (código " + respuesta.status + ")");
    }
    let usuario = await respuesta.json();

    // Operación 2: pedir sus publicaciones (depende del usuario)
    let respuestaPosts = await fetch(
      "https://jsonplaceholder.typicode.com/posts?userId=" + usuario.id
    );
    if (!respuestaPosts.ok) {
      throw new Error("No se pudieron cargar las publicaciones");
    }
    let posts = await respuestaPosts.json();

    // Lógica normal mezclada con lo asíncrono, sin problema
    console.log("Reporte de " + usuario.name + ":");
    console.log("- Email: " + usuario.email);
    console.log("- Publicaciones: " + posts.length);
    console.log("- Primer título: " + posts[0].title);

  } catch (error) {
    // Atrapa fallos de CUALQUIERA de las operaciones de arriba
    console.log("No se pudo generar el reporte:", error.message);
  }
}

reporteUsuario(1); // funciona: muestra el reporte
// reporteUsuario(999); // falla: el catch muestra el mensaje amable

```

Observa la URL con?userId=: es un "query string" que le pide a la API solo las publicaciones de ese usuario. Y observa cómo el código mezcla awaits, ifs y console.log con total naturalidad: eso es lo que then no podía hacer sin enredarse. Prueba con 999 para ver el camino del error: el throw del primer if manda todo directo al catch.

Ejemplo 6: await dentro de un ciclo for

Procesar una lista esperando cada operación, de forma legible.

```

async function saludarUsuarios(ids) {
  for (let id of ids) {
    try {
      let respuesta = await fetch("https://jsonplaceholder.typicode.com/users/" +
id);
      let usuario = await respuesta.json();
      console.log("Hola, " + usuario.name);
    } catch (error) {
      console.log("No se pudo saludar al usuario " + id);
    }
  }
  console.log("Terminamos de saludar a todos");
}

saludarUsuarios([1, 2, 3]);

```

Con `then`, esto habría requerido recursión o utilidades raras. Con `await`, es un `for` normal con un `await` adentro: se procesa el 1, luego el 2, luego el 3, en orden. El `try/catch` dentro del ciclo hace que si un usuario falla, se reporte y se siga con el siguiente, sin romper todo el ciclo. Este patrón (recorrer + esperar + manejar errores por elemento) aparece en montones de aplicaciones reales: migraciones, reportes, sincronizaciones.

Ejemplo 7: El contador completo, listo para copiar y probar

Aquí está el proyecto de la unidad en su forma final, con comentarios que explican cada bloque. Guárdalo como `contador.js` junto al HTML de la explicación y ábrelo en el navegador.

```

// ===== ESTADO =====
// La única fuente de verdad: el valor actual del contador
let cuenta = 0;

// ===== SELECCIÓN DEL DOM =====
// Guardamos los elementos una sola vez, al inicio
let numero = document.getElementById("numero");
let btnSumar = document.getElementById("btnSumar");
let btnRestar = document.getElementById("btnRestar");
let btnReiniciar = document.getElementById("btnReiniciar");

// ===== MOSTRAR =====
// Pinta el estado actual en pantalla. Se llama después de cada cambio.
function mostrar() {
    numero.textContent = cuenta;
}

// ===== EVENTOS: escuchar, actualizar, mostrar =====
btnSumar.addEventListener("click", function() {
    cuenta = cuenta + 1; // actualizamos el estado
    mostrar();           // reflejamos el cambio
});

btnRestar.addEventListener("click", function() {
    cuenta = cuenta - 1;
    mostrar();
});

btnReiniciar.addEventListener("click", function() {
    cuenta = 0;
    mostrar();
});

// ===== INICIO =====
// Pintamos el valor inicial al cargar la página
mostrar();

```

Y el HTML que lo acompaña (guárdalo como contador.html en la misma carpeta):

```

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Mi contador</title>
</head>
<body>
  <h1>Contador</h1>
  <p id="numero">0</p>
  <button id="btnSumar">Sumar</button>
  <button id="btnRestar">Restar</button>
  <button id="btnReiniciar">Reiniciar</button>
  <script src="contador.js"></script>
</body>
</html>

```

Pruébalo: abre el HTML, haz clic en los botones, observa el número cambiar. Luego haz los mini-retos de la explicación: que no baje de 0, colores según el valor, botón de +5, guardar en localStorage. Cada mejora que agregues tú solo es una victoria real.

Ejemplo 8: Contador con límite y mensaje (mini-reto resuelto)

Una versión mejorada del contador que no baja de 0, no sube de 10, y avisa cuando llega a los límites. Así se ve un proyecto "terminado" con reglas.

```

let cuenta = 0;
let numero = document.getElementById("numero");
let mensaje = document.getElementById("mensaje"); // un <p id="mensaje"></p> en el
HTML

let btnSumar = document.getElementById("btnSumar");
let btnRestar = document.getElementById("btnRestar");
let btnReiniciar = document.getElementById("btnReiniciar");

function mostrar() {
  numero.textContent = cuenta;

  // Cambiamos el color según el valor
  if (cuenta > 0) {
    numero.style.color = "green";
  } else if (cuenta < 0) {
    numero.style.color = "red";
  } else {
    numero.style.color = "black";
  }
}

```



```

    }
}

btnSumar.addEventListener("click", function() {
    if (cuenta < 10) {
        cuenta = cuenta + 1;
        mensaje.textContent = "";
    } else {
        mensaje.textContent = "¡Llegaste al máximo de 10!";
    }
    mostrar();
});

btnRestar.addEventListener("click", function() {
    if (cuenta > 0) {
        cuenta = cuenta - 1;
        mensaje.textContent = "";
    } else {
        mensaje.textContent = "El contador no puede bajar de 0.";
    }
    mostrar();
});

btnReiniciar.addEventListener("click", function() {
    cuenta = 0;
    mensaje.textContent = "Contador reiniciado.";
    mostrar();
});

mostrar();

```

Este ejemplo combina condicionales (clase 6), DOM (clases 20-21) y eventos (clase 22) en un solo proyecto coherente. Nota cómo `mostrar()` creció: ahora también decide el color. Y los botones ahora tienen reglas: sumar solo si no llegamos al máximo, restar solo si no estamos en cero. Los mensajes le dicen al usuario por qué un botón "no hizo nada". Esa comunicación es lo que separa una app que se siente rota de una que se siente bien diseñada.

¿Sabías que? En la industria, a este tipo de reglas se les llama "validaciones" o "casos borde", y gran parte del trabajo profesional consiste en pensar en ellos: ¿qué pasa si el usuario hace clic 100 veces? ¿qué pasa si el valor llega a un límite? El programador junior hace que funcione el caso feliz; el senior hace que no se rompa en los casos raros. Con este ejemplo ya estás pensando como senior.

Ejemplo 9: Reintentar una operación que falla (lógica real con await)

En el mundo real, las peticiones a veces fallan por causas temporales (la red parpadeó, el servidor estaba ocupado). Una técnica profesional es reintentar unas cuantas veces antes de rendirse. Con `await`, esto se escribe con un ciclo `for` normal: algo impensable con `then`.

```
// Función que simula una operación que falla las primeras veces
let intentos = 0;
function operacionFragil() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      intentos++;
      if (intentos < 3) {
        reject(new Error("Fallo temporal (intento " + intentos + ")"));
      } else {
        resolve("Éxito en el intento " + intentos);
      }
    }, 500);
  });
}

async function conReintentos(maxIntentos) {
  for (let i = 1; i <= maxIntentos; i++) {
    try {
      let resultado = await operacionFragil();
      console.log(resultado);
      return resultado; // ¡éxito! salimos de la función
    } catch (error) {
      console.log("Intento " + i + " falló: " + error.message);
      if (i === maxIntentos) {
        console.log("Se acabaron los intentos. Nos rendimos.");
        throw error; // avisamos a quien nos llamó
      }
      // si no era el último intento, el ciclo sigue y reintentamos
    }
  }
}

conReintentos(5).catch(function(e) {
  console.log("Fallo definitivo:", e.message);
});
```

Lee la lógica: intentamos la operación; si funciona, devolvemos el resultado y terminamos; si falla, registramos el fallo y el ciclo sigue con el siguiente intento; solo cuando se acaban los intentos lanzamos el error para que lo maneje quien llamó. Con `then`, este "intenta hasta 5 veces" requería recursión enredada. Con `await`, es un `for` con `try/catch` adentro: lógica que cualquier programador entiende al primer vistazo. Este patrón (reintentos con límite) aparece en pagos, envíos de formularios y sincronizaciones: es oro profesional.

Ejemplo 10: Convertir una función con callback en promesa (puente entre mundos)

Todavía existe código viejo basado en callbacks (funciones que reciben otra función para avisar cuando terminan). Para usarlo con `await`, lo envuelves en una promesa. Esta técnica te salva cuando trabajas con librerías antiguas.

```
// Función estilo antiguo: avisa con un callback, no devuelve promesa
function leerDatoViejo(callback) {
  setTimeout(function() {
    callback("dato antiguo");
  }, 1000);
}

// La envolvemos: creamos una promesa que se cumple cuando el callback avisa
function leerDatoModerno() {
  return new Promise(function(resolve) {
    leerDatoViejo(function(dato) {
      resolve(dato); // el callback ahora cumple la promesa
    });
  });
}

// Y ahora sí podemos usar await con ella
async function usarDato() {
  let dato = await leerDatoModerno();
  console.log("Recibido:", dato);
}

usarDato();
```

El truco es siempre el mismo: `new Promise`, y dentro, cuando el mecanismo antiguo avise (callback, evento, lo que sea), llamas a `resolve` o `reject`. Esta envoltura se llama "promisificar", y es una de las habilidades más útiles para trabajar con código heredado: no reescribes la

librería vieja, la envuelves y la usas con sintaxis moderna. Si algún día te topas con una API que solo habla callbacks, ya sabes qué hacer.

Práctica

Última ronda de ejercicios del curso. Hazlos con calma y con orgullo: cada uno que resuelvas es una prueba de que lo aprendido se quedó.

Ejercicio 1: Convierte una cadena then a async/await

Enunciado: Toma esta cadena de promesas y reescríbela usando async/await con try/catch.

Debe hacer exactamente lo mismo: pedir el post con id 1 a

<https://jsonplaceholder.typicode.com/posts/1>, convertirlo a JSON, mostrar su título, y manejar errores.

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(function(respuesta) { return respuesta.json(); })
  .then(function(post) { console.log(post.title); })
  .catch(function(error) { console.log("Error:", error.message); });
```

Solución:

```
async function mostrarTitulo() {
  try {
    let respuesta = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    let post = await respuesta.json();
    console.log(post.title);
  } catch (error) {
    console.log("Error:", error.message);
  }
}

mostrarTitulo();
```

Explicación paso a paso: Envolvemos todo en una función marcada con `async`, porque `await` solo vive dentro de funciones `async`. Cada `then` se convierte en una línea con `await`: primero esperamos la respuesta, luego esperamos la conversión a JSON. El `catch()` final se convierte en el bloque `catch` del `try/catch` que envuelve todo. No olvides llamar a la función al final: definir una `async` no la ejecuta. Si comparas ambas versiones, verás que la lógica es idéntica pero la lectura es lineal, sin `callbacks`.

Ejercicio 2: Función `async` que suma dos números de una API

Enunciado: Escribe una función `async` llamada `sumarPrecios` que pida dos "productos" a la API de práctica (usa <https://jsonplaceholder.typicode.com/posts/1> y [/posts/2](https://jsonplaceholder.typicode.com/posts/2), e interpreta el id de cada post como su "precio"). La función debe pedir ambos en paralelo (iniciar las dos peticiones sin `await` y esperar después), sumar los dos ids, y devolver el total. Luego llama a la función y muestra el resultado. Maneja errores con `try/catch`.

Solución:

```
async function sumarPrecios() {
  try {
    // Iniciamos ambas peticiones SIN esperar: viajan en paralelo
    let promesa1 = fetch("https://jsonplaceholder.typicode.com/posts/1");
    let promesa2 = fetch("https://jsonplaceholder.typicode.com/posts/2");

    // Ahora sí esperamos cada una y convertimos
    let producto1 = await (await promesa1).json();
    let producto2 = await (await promesa2).json();

    let total = producto1.id + producto2.id;
    return total; // al ser async, esto llega como promesa cumplida
  } catch (error) {
    console.log("Error:", error.message);
    return 0;
  }
}

sumarPrecios().then(function(total) {
  console.log("Total:", total); // 3 (1 + 2)
});
```

Explicación paso a paso: Iniciamos los dos `fetch` sin `await` para que las peticiones viajen al mismo tiempo por internet: eso es paralelismo real y tarda roughly lo de una sola petición en

lugar de la suma. Luego esperamos cada promesa y convertimos su cuerpo a JSON con el doble `await`. Sumamos los ids ($1 + 2 = 3$) y devolvemos el total con `return`; como la función es `async`, quien la llame recibe una promesa, por eso la llamamos con `.then` para mostrar el resultado. El `catch` devuelve 0 como valor seguro si algo falla, para que la función siempre devuelva un número. Este ejercicio practica las tres ideas clave: paralelismo, doble `await`, y que `async` siempre devuelve promesa.

Ejercicio 3: Contador con paso personalizado

Enunciado: Modifica el contador del proyecto para que, además de los tres botones originales, tenga un campo de texto donde el usuario escriba un número y un botón "Sumar cantidad" que sume esa cantidad al contador. Si el campo está vacío o no es un número válido, muestra el mensaje "Escribe un número válido." (Pista: usa `parseInt` y `isNaN`, de las clases de tipos de datos.)

Solución:

HTML (agregar dentro del `body`):

```
<input type="text" id="cantidad" placeholder="Cantidad">
<button id="btnSumarCantidad">Sumar cantidad</button>
<p id="mensaje"></p>
```

JavaScript (agregar al `contador.js`):

```
let cantidad = document.getElementById("cantidad");
let btnSumarCantidad = document.getElementById("btnSumarCantidad");
let mensaje = document.getElementById("mensaje");

btnSumarCantidad.addEventListener("click", function() {
  let valor = parseInt(cantidad.value);

  if (isNaN(valor)) {
    mensaje.textContent = "Escribe un número válido.";
  } else {
    cuenta = cuenta + valor;
    mensaje.textContent = "Se sumaron " + valor + ".";
    cantidad.value = ""; // limpiamos el campo para la próxima
  }
  mostrar();
});
```

Explicación paso a paso: Agregamos un input para que el usuario escriba la cantidad y un botón nuevo, más un párrafo para mensajes. En el evento del botón, leemos `cantidad.value` (que siempre es texto) y lo convertimos con `parseInt`. Si el resultado es `NaN` (no es un número), mostramos el mensaje de error y no tocamos el contador: esa es la validación, el caso borde. Si es válido, sumamos al estado, avisamos cuánto se sumó y limpiamos el campo para que el usuario no tenga que borrarlo a mano (detalle de experiencia de usuario). Finalmente llamamos a `mostrar()` para reflejar el cambio, igual que los otros botones. Este ejercicio conecta formularios (clase 23) con el patrón del proyecto: leer input -> validar -> actualizar estado -> mostrar.

Ejercicio 4: Contador persistente con localStorage

Enunciado: Haz que el contador no se pierda al recargar la página usando `localStorage`. Al iniciar, lee el valor guardado (si existe) y úsalo como cuenta inicial. Cada vez que la cuenta cambie, guarda el nuevo valor. Usa `JSON.stringify` para guardar y `JSON.parse` para leer (aunque sea un número simple, practica el ciclo de la unidad 9).

Solución:

```

// Al iniciar: intentamos leer el valor guardado
let guardado = localStorage.getItem("miContador");
let cuenta = guardado === null ? 0 : JSON.parse(guardado);

let numero = document.getElementById("numero");
let btnSumar = document.getElementById("btnSumar");
let btnRestar = document.getElementById("btnRestar");
let btnReiniciar = document.getElementById("btnReiniciar");

function mostrar() {
    numero.textContent = cuenta;
}

// Función nueva: guarda el estado actual en localStorage
function guardar() {
    localStorage.setItem("miContador", JSON.stringify(cuenta));
}

btnSumar.addEventListener("click", function() {
    cuenta = cuenta + 1;
    guardar(); // persistimos cada cambio
    mostrar();
});

btnRestar.addEventListener("click", function() {
    cuenta = cuenta - 1;
    guardar();
    mostrar();
});

btnReiniciar.addEventListener("click", function() {
    cuenta = 0;
    guardar();
    mostrar();
});

mostrar();

```

Explicación paso a paso: Al cargar, leemos "miContador" del localStorage. Si es null (primera visita, nada guardado), empezamos en 0; si existe, lo convertimos de texto a número con JSON.parse. Creamos la función guardar() que convierte la cuenta a JSON y la almacena: centralizar el guardado en una función evita repetir la línea en cada botón y hace el código más fácil de mantener. Cada botón ahora hace tres cosas: actualizar, guardar, mostrar.

Prueba: suma 5, recarga la página, y el 5 sigue ahí. Acabas de darle memoria permanente a tu app: el estado ya no vive solo en la memoria volátil, sino en el disco del navegador. Este es el mismo principio que usan las apps reales para recordar tus preferencias.

Ejercicio 5: Mini-proyecto final -- lista de tareas

Enunciado: Construye una mini app de lista de tareas (to-do list) aplicando el patrón del contador: un campo de texto, un botón "Agregar", y una lista donde aparece cada tarea con un botón "Borrar" junto a ella. Al agregar, si el campo está vacío muestra un mensaje de aviso y no agregues nada. Este es tu proyecto de graduación: si lo logras, dominas el ciclo estado -> mostrar -> escuchar -> actualizar.

Solución:

HTML:

```
<h1>Mis tareas</h1>
<input type="text" id="nuevaTarea" placeholder="Escribe una tarea">
<button id="btnAgregar">Agregar</button>
<p id="aviso"></p>
<ul id="listaTareas"></ul>
```

JavaScript:

```

// ESTADO: arreglo con las tareas
let tareas = [];

let input = document.getElementById("nuevaTarea");
let btnAgregar = document.getElementById("btnAgregar");
let aviso = document.getElementById("aviso");
let lista = document.getElementById("listaTareas");

// MOSTRAR: pinta el estado actual en la lista
function mostrar() {
    lista.innerHTML = ""; // limpiamos antes de pintar

    tareas.forEach(function(tarea, indice) {
        let item = document.createElement("li");
        item.textContent = tarea + " ";

        let btnBorrar = document.createElement("button");
        btnBorrar.textContent = "Borrar";
        btnBorrar.addEventListener("click", function() {
            tareas.splice(indice, 1); // quitamos esa tarea del estado
            mostrar(); // volvemos a pintar
        });

        item.appendChild(btnBorrar);
        lista.appendChild(item);
    });
}

// ESCUCHAR Y ACTUALIZAR: agregar tareas
btnAgregar.addEventListener("click", function() {
    let texto = input.value.trim(); // trim quita espacios al inicio y final

    if (texto === "") {
        aviso.textContent = "Escribe una tarea antes de agregar.";
    } else {
        tareas.push(texto); // actualizamos el estado
        input.value = "";
        aviso.textContent = "";
        mostrar(); // reflejamos el cambio
    }
});

mostrar();

```

Explicación paso a paso: El estado ahora es un arreglo (tareas), no un número: el patrón es el mismo, solo cambió la forma del estado. La función mostrar() limpia la lista y la reconstruye desde el arreglo: por cada tarea crea un li con su texto y un botón Borrar. Cada botón Borrar conoce su índice (gracias al forEach) y al hacer clic elimina esa posición con splice, luego vuelve a pintar. Al agregar, validamos con trim() que el texto no esté vacío (ni lleno de espacios) antes de hacer push al arreglo. Si lo construiste y funciona --agregar, borrar, validación--, felicidades: acabas de programar el proyecto clásico con el que miles de desarrolladores practican, y lo hiciste entendiendo cada línea. Ese es tu graduado real, más que cualquier diploma.

Ejercicio 6 (reto de graduación): tareas persistentes + async

Enunciado: Combina todo: haz que la lista de tareas del ejercicio 5 se guarde en localStorage (como el ejercicio 4), y agrega un botón "Cargar tareas de ejemplo" que use async/await con fetch para pedir 3 tareas de https://jsonplaceholder.typicode.com/todos?_limit=3 y agregar sus títulos a tu lista. Maneja errores con try/catch mostrando un aviso si falla la carga.

Solución (solo la parte nueva, se agrega al ejercicio 5):

```

// Al iniciar: leer tareas guardadas
let guardadas = localStorage.getItem("misTareas");
if (guardadas !== null) {
  tareas = JSON.parse(guardadas);
}

function guardarTareas() {
  localStorage.setItem("misTareas", JSON.stringify(tareas));
}

// Llamar a guardarTareas() dentro de mostrar(), al final:
// function mostrar() { ... todo lo anterior ...; guardarTareas(); }

// Botón nuevo en el HTML:
// <button id="btnEjemplos">Cargar tareas de ejemplo</button>

let btnEjemplos = document.getElementById("btnEjemplos");

btnEjemplos.addEventListener("click", async function() {
  try {
    aviso.textContent = "Cargando ejemplos...";

    let respuesta = await
fetch("https://jsonplaceholder.typicode.com/todos?_limit=3");
    if (!respuesta.ok) {
      throw new Error("Error del servidor: " + respuesta.status);
    }
    let ejemplos = await respuesta.json();
    ejemplos.forEach(function(ej) {
      tareas.push(ej.title);
    });
    aviso.textContent = "Se agregaron " + ejemplos.length + " tareas de
ejemplo.";
    mostrar();
  } catch (error) {
    aviso.textContent = "No se pudieron cargar los ejemplos. Revisa tu
conexión.";
    console.log("Detalle:", error.message);
  }
});

```

Explicación paso a paso: Primero la persistencia: al iniciar leemos el arreglo guardado (o empezamos vacío), y guardamos en cada mostrar() para que ningún cambio se pierda. Luego el botón de ejemplos: su función de clic es async (sí, los listeners pueden ser async), y dentro

usa `await` con `fetch`, verifica `respuesta.ok` con `throw`, convierte con `await respuesta.json()`, y agrega cada título al arreglo con `forEach + push`. La URL usa `?_limit=3` para pedir solo 3 tareas. El `try/catch` envuelve todo y muestra avisos amables en ambos caminos. Si completaste este reto, usaste en un solo proyecto: estado, DOM, eventos, validación, `localStorage` con JSON, `fetch`, `async/await`, `try/catch` y query strings. Eso es, sin exagerar, el 80% del trabajo diario de un desarrollador web junior. Guárdalo, presúmelo, y sigue construyendo.

Ejercicio 7: Reintenta una petición hasta 3 veces

Enunciado: Escribe una función `async` llamada `fetchConReintentos(url, maxIntentos)` que intente hacer `fetch` a la url. Si la petición tiene éxito (`respuesta.ok`), devuelve los datos ya convertidos a JSON. Si falla (error de red o respuesta no ok), reintenta hasta `maxIntentos` veces. Si se acaban los intentos, lanza el último error. Pruébala con una URL válida (`https://jsonplaceholder.typicode.com/users/1`) y con una inválida (`https://jsonplaceholder.typicode.com/usuarios/1`, que da 404) para ver ambos caminos.

Solución:

```

async function fetchConReintentos(url, maxIntentos) {
  let ultimoError;

  for (let i = 1; i <= maxIntentos; i++) {
    try {
      let respuesta = await fetch(url);
      if (!respuesta.ok) {
        throw new Error("Código " + respuesta.status);
      }
      return await respuesta.json(); // éxito: devolvemos los datos
    } catch (error) {
      ultimoError = error;
      console.log("Intento " + i + " falló: " + error.message);
    }
  }

  throw ultimoError; // se acabaron los intentos
}

// Prueba 1: URL válida, funciona al primer intento
fetchConReintentos("https://jsonplaceholder.typicode.com/users/1", 3)
  .then(function(usuario) { console.log("OK:", usuario.name); })
  .catch(function(e) { console.log("Falló todo:", e.message); });

// Prueba 2: URL inválida (404), reintentamos 3 veces y luego falla
// fetchConReintentos("https://jsonplaceholder.typicode.com/usuarios/1", 3)
//   .then(function(d) { console.log(d); })
//   .catch(function(e) { console.log("Falló todo:", e.message); });

```

Explicación paso a paso: El ciclo for intenta la operación hasta maxIntentos veces. Dentro del try hacemos el fetch completo con verificación de ok: si todo sale bien, el return devuelve los datos y la función termina (los intentos restantes ya no importan). Si algo falla, el catch guarda el error y registra el intento fallido, y el ciclo continúa. Al salir del ciclo sin haber devuelto nada, significa que todos los intentos fallaron, así que lanzamos el último error con throw para que quien llamó lo maneje con catch. Nota que guardamos ultimoError fuera del ciclo: sin esa variable, el error se perdería al terminar el for. Con la URL válida verás un solo intento exitoso; con la inválida verás 3 intentos fallidos y el mensaje final. Este es el patrón de reintentos que usan las apps serias, y ahora lo escribes en 20 líneas legibles.

Ejercicio 8: Quiz de 3 preguntas con puntaje

Enunciado: Construye un mini quiz: muestra una pregunta con 3 botones de respuesta; al hacer clic en una, muestra si fue correcta o no, suma el puntaje si acertó, y pasa a la siguiente pregunta. Al terminar las 3, muestra el puntaje final y un botón "Reiniciar" que empieza de nuevo. Usa el patrón del contador: estado (índice actual + puntaje), mostrar, escuchar, actualizar.

Solución:

HTML:

```
<h1>Quiz relámpago</h1>
<p id="pregunta"></p>
<div id="opciones"></div>
<p id="resultado"></p>
<p id="puntaje"></p>
<button id="btnReiniciar" style="display:none">Reiniciar</button>
```

JavaScript:

```
// ESTADO: preguntas, posición actual y puntaje
let preguntas = [
  { texto: "¿Cuánto es 2 + 2?", opciones: ["3", "4", "5"], correcta: 1 },
  { texto: "¿Capital de México?", opciones: ["Lima", "Bogotá", "Ciudad de México"],
correcta: 2 },
  { texto: "¿Qué significa HTML?", opciones: ["HyperText Markup Language", "High
Tech Machine Learning", "Home Tool Manual Language"], correcta: 0 }
];
let indice = 0;
let puntaje = 0;

let preguntaEl = document.getElementById("pregunta");
let opcionesEl = document.getElementById("opciones");
let resultadoEl = document.getElementById("resultado");
let puntajeEl = document.getElementById("puntaje");
let btnReiniciar = document.getElementById("btnReiniciar");

// MOSTRAR: pinta la pregunta actual y sus botones
function mostrar() {
  resultadoEl.textContent = "";
  puntajeEl.textContent = "Puntaje: " + puntaje + " / " + preguntas.length;

  if (indice >= preguntas.length) {
```

```

        // Se acabaron las preguntas: mostramos el final
        preguntaEl.textContent = "¡Quiz terminado!";
        opcionesEl.innerHTML = "";
        resultadoEl.textContent = "Tu puntaje final: " + puntaje + " de " +
preguntas.length;
        btnReiniciar.style.display = "inline";
        return;
    }

    let actual = preguntas[indice];
    preguntaEl.textContent = actual.texto;
    opcionesEl.innerHTML = "";

    actual.opciones.forEach(function(opcion, i) {
        let boton = document.createElement("button");
        boton.textContent = opcion;
        boton.addEventListener("click", function() {
            // ACTUALIZAR: revisamos la respuesta
            if (i === actual.correcta) {
                puntaje = puntaje + 1;
                resultadoEl.textContent = "¡Correcto!";
            } else {
                resultadoEl.textContent = "Incorrecto. La respuesta era: " +
actual.opciones[actual.correcta];
            }
            indice = indice + 1;
            // Pequeña pausa para leer el resultado antes de la siguiente pregunta
            setTimeout(mostrar, 1200);
        });
        opcionesEl.appendChild(boton);
    });
}

btnReiniciar.addEventListener("click", function() {
    indice = 0;
    puntaje = 0;
    btnReiniciar.style.display = "none";
    mostrar();
});

mostrar();

```

Explicación paso a paso: El estado ahora tiene tres piezas: el arreglo de preguntas (datos fijos), el índice de la pregunta actual y el puntaje. La función mostrar() pinta la pregunta actual

creando un botón por cada opción con `forEach`; cada botón lleva su propio listener que compara el índice elegido con el índice correcto. Si acierta, suma puntaje; si no, muestra cuál era la correcta (feedback educativo, no solo "mal"). Después de responder, avanzamos el índice y usamos `setTimeout` para dar 1.2 segundos de lectura antes de pintar la siguiente: fíjate cómo conviven el temporizador (clase 24) con el patrón del proyecto. Cuando el índice supera el total, `mostrar()` detecta el final, limpia las opciones y revela el botón Reiniciar, que restablece el estado y vuelve a empezar. Si construiste esto entendiendo cada parte, ya no eres "alguien que terminó un curso": eres alguien que construye aplicaciones interactivas con estado, eventos y lógica. Bienvenido al club.

Resumen de la unidad

`async/await` es la forma moderna de escribir código asíncrono: por debajo son promesas, pero el código se lee de arriba hacia abajo como si fuera síncrono, sin callbacks anidados. En código nuevo, prefiere `async/await`; entiende `then` porque lo verás en código existente toda tu vida.

Una función marcada con `async` siempre devuelve una promesa: si devuelves un valor normal, llega como promesa cumplida; si lanzas un error, llega como promesa rechazada. Solo dentro de funciones `async` puedes usar `await`.

`await` pausa la ejecución de SU función hasta que la promesa se cumple, y luego continúa con el valor en la misma línea. La pausa es local: el resto del programa sigue funcionando. Espera sin bloquear: esa es toda la magia, bien entendida.

Los errores típicos: usar `await` fuera de una función `async` (error de sintaxis inmediato), y olvidar el `await` (recibes "Promise {<pending>}" donde esperabas un dato). Ante un `pending` inesperado, busca el `await` faltante.

Si dos operaciones son independientes, inícialas sin `await` y espera después: viajan en paralelo y tu código tarda menos. Si una depende del resultado de la otra, el `await` secuencial es lo correcto. `await` también funciona dentro de ciclos `for`, lo que hace triviales las tareas repetidas con espera.

Maneja los errores de `await` con `try/catch`: si cualquier promesa esperada se rechaza, el control salta al `catch`. Un solo `try/catch` puede proteger una secuencia completa de operaciones, con condicionales y ciclos adentro, algo que con `then` era mucho más enredado.

Todo proyecto interactivo sigue el mismo ciclo: estado (datos en memoria, una sola fuente de verdad) -> mostrar (pintar el estado en pantalla) -> escuchar (atender acciones del usuario) ->

actualizar (cambiar el estado) -> volver a mostrar. El contador es ese ciclo en su forma más pura.

En el contador: la variable cuenta es el estado; `getElementByld` una sola vez al inicio es eficiente; la función `mostrar()` sincroniza pantalla y estado; cada botón actualiza y luego muestra; y se llama a `mostrar()` al cargar para pintar el valor inicial. Los ids deben ser únicos y el script va al final del body.

Las mejoras separan lo amateur de lo profesional: validar entradas (que no baje de 0, que el texto sea número), dar feedback (mensajes cuando algo no se puede hacer), y persistir con `localStorage` usando `JSON.stringify` y `JSON.parse` para que los datos sobrevivan la recarga.

Repaso del curso: 30 clases en 10 unidades, de "¿qué es JavaScript?" a aplicaciones conectadas a servidores. Variables, tipos, operadores, condicionales, ciclos, funciones, arreglos, objetos, DOM, eventos, JSON, promesas, `fetch` y `async/await`: ya no son temas sueltos, son tu caja de herramientas.

Lo que sigue: practica con proyectos (lista de tareas, calculadora, clima con API real), profundiza en lo que más te costó, aprende Git para guardar tu trabajo, lee documentación en MDN como hábito, y cuando domines lo básico, da el salto a React. Y sigue tu camino en lucasegaacademy.com, donde te esperan los demás cursos del ecosistema: HTML, CSS y desarrollo web completo, gratuitos, en español, hechos para que sigas creciendo.

Guarda tus proyectos y vuelve a verlos en un año: la risa que te den será la medida exacta de cuánto avanzaste. Felicidades por terminar el Curso Completo de JavaScript. Ahora empieza lo bueno: construir.

Un recordatorio final sobre `async/await` que vale oro: la sintaxis es dulce, pero piensa en promesas. Cuando algo se comporte raro, pregúntate qué promesa está pendiente, dónde falta un `await` y quién atrapa el rechazo. Esa pregunta te saca de cualquier atolladero.

Y sobre el manejo de errores: el código profesional no es el que funciona cuando todo sale bien, sino el que se comporta con dignidad cuando algo falla. `try/catch` en cada función `async`, verificación de `respuesta.ok` en cada `fetch`, mensajes amables para el usuario: esos hábitos te distinguen desde tu primer proyecto.

No conviertas todo a `async` por moda: si una función no espera nada asíncrono, que sea una función normal. La elegancia está en usar la herramienta correcta, no la más nueva.

Por último: enseña lo que aprendiste. Explicarle a alguien más cómo funciona una promesa o cómo se arma el contador es la prueba definitiva de que lo dominas, y es como la comunidad

de programadores crece. Nos vemos en lucasegaacademy.com, en el siguiente curso. A construir.

EXAMEN FINAL

Llegaste al final del libro. Este examen tiene 50 preguntas que cubren las 10 unidades: opción múltiple, verdadero o falso, y preguntas abiertas donde tendrás que escribir o analizar código. Hazlo con calma, sin ver las respuestas. Al terminar, revisa la sección "## Respuestas y soluciones" y cuenta tu puntaje: 45 o más es excelente, 40 a 44 es bueno, 35 a 39 significa que debes repasar un par de unidades, y menos de 35 te recomiendo releer el libro completo antes de seguir.

Preguntas

PREGUNTA 1 (Opción múltiple): ¿Cuál es la forma correcta de incluir un archivo JavaScript externo al final del body para que no bloquee la carga de la página?

a) `<script href="app.js"></script>` b) `<script src="app.js" defer></script>` c) `<javascript src="app.js"></javascript>` d) `<script import="app.js"></script>`

PREGUNTA 2 (Verdadero/Falso): Con `const` puedes cambiar el valor de una variable cuantas veces quieras.

PREGUNTA 3 (Abierta): Escribe el código para declarar una variable llamada `nombre` que pueda cambiar de valor, y una constante llamada `PI` con valor 3.1416.

PREGUNTA 4 (Opción múltiple): ¿Qué imprime `typeof "42"`?

a) number b) string c) text d) undefined

PREGUNTA 5 (Opción múltiple): ¿Cuál es el resultado de `10% 3`?

a) 3 b) 1 c) 0 d) 3.33

PREGUNTA 6 (Abierta): Escribe un `if...else` que imprima "Puedes votar" si la variable `edad` es mayor o igual a 18, y "Aún no puedes votar" en caso contrario.

PREGUNTA 7 (Verdadero/Falso): En JavaScript, `"5" === 5` devuelve `true`.

PREGUNTA 8 (Opción múltiple): ¿Cuál operador lógico representa "Y" (ambas condiciones deben cumplirse)?

a) `||` b) `&&` c) `!` d) `??`

PREGUNTA 9 (Opción múltiple): ¿Cuántas veces se ejecuta el cuerpo de `for (let i = 0; i < 5; i++)`?

a) 4 b) 5 c) 6 d) Infinitas

PREGUNTA 10 (Abierta): Escribe un ciclo `while` que imprima los números del 1 al 10.

PREGUNTA 11 (Verdadero/Falso): Una función declarada con `function saludar() {}` puede ser llamada antes de su declaración en el código.

PREGUNTA 12 (Opción múltiple): ¿Qué palabra clave usa una función para devolver un valor a quien la llamó?

a) `send` b) `return` c) `output` d) `give`

PREGUNTA 13 (Abierta): Escribe una función flecha llamada `doble` que reciba un número y devuelva el doble.

PREGUNTA 14 (Opción múltiple): Dado `let frutas = ["manzana", "pera", "uva"]`, ¿qué valor tiene `frutas[1]`?

a) manzana b) pera c) uva d) undefined

PREGUNTA 15 (Verdadero/Falso): El método `push()` agrega un elemento al inicio de un array.

PREGUNTA 16 (Opción múltiple): ¿Cuál es la diferencia principal entre `forEach` y `map`?

a) No hay diferencia b) `forEach` devuelve un array nuevo y `map` no c) `map` devuelve un array nuevo transformado y `forEach` no devuelve nada útil d) `map` solo funciona con números

PREGUNTA 17 (Abierta): Dado el objeto `let persona = {nombre: "Lucas", edad: 30}` ;, escribe el código para imprimir su nombre usando la notación de punto.

PREGUNTA 18 (Verdadero/Falso): Dentro de un método de un objeto, `this` se refiere al propio objeto.

PREGUNTA 19 (Opción múltiple): ¿Cuál es la sintaxis correcta de un template literal que incluye la variable `nombre`?

a) `"Hola ${nombre}"` b) `'Hola ${nombre}'` c) `Hola ${nombre}` d) `(Hola ${nombre})`

PREGUNTA 20 (Abierta): Escribe el código para seleccionar el elemento con id `"titulo"` y guardarlo en una constante llamada `titulo`.

PREGUNTA 21 (Opción múltiple): ¿Cuál es la diferencia entre `textContent` e `innerHTML` ?

a) No hay diferencia b) `textContent` interpreta etiquetas HTML e `innerHTML` no c) `innerHTML` interpreta etiquetas HTML y `textContent` las trata como texto plano d) `textContent` solo funciona en formularios

PREGUNTA 22 (Abierta): Escribe el código para que un botón con id "miBoton" muestre una alerta que diga "¡Hola!" cuando se le haga clic.

PREGUNTA 23 (Verdadero/Falso): `event.preventDefault()` en un formulario evita que la página se recargue al enviarlo.

PREGUNTA 24 (Opción múltiple): `setTimeout(saludar, 2000)` ejecuta la función `saludar...`

a) Cada 2 segundos para siempre b) Una sola vez después de 2 segundos c) 2000 veces d) Inmediatamente 2 veces

PREGUNTA 25 (Abierta): Escribe el código para convertir el objeto `{nombre: "Ana"}` a texto JSON y guardarlo en una variable llamada `texto`.

PREGUNTA 26 (Verdadero/Falso): Una promesa solo puede estar en estado "pendiente" para siempre; nunca cambia de estado.

PREGUNTA 27 (Opción múltiple): En `fetch(url).then(respuesta =>...).catch(error =>...)`, ¿para qué sirve el `catch`?

a) Para repetir la petición b) Para manejar errores si la promesa se rechaza c) Para convertir la respuesta a JSON automáticamente d) Para cancelar la petición

PREGUNTA 28 (Abierta): Escribe una función `async` llamada `cargarDatos` que use `await fetch()` para pedir datos de `"https://api.ejemplo.com/datos"` dentro de un `try/catch`.

PREGUNTA 29 (Opción múltiple): ¿Qué hace la palabra clave `await`?

a) Detiene todo el navegador hasta que termina b) Pausa la ejecución de la función `async` hasta que la promesa se resuelve, sin bloquear el resto del programa c) Cancela la promesa d) Convierte una función normal en promesa

PREGUNTA 30 (Verdadero/Falso): `let` y `const` tienen alcance de bloque, mientras que `var` tiene alcance de función (o global).

PREGUNTA 31 (Opción múltiple): ¿Qué imprime este código? `console.log(2 + "2");`

a) 4 b) "22" c) Error d) NaN

PREGUNTA 32 (Abierta): Explica con tus palabras la diferencia entre `==` y `===` y da un ejemplo de cada uno.

PREGUNTA 33 (Opción múltiple): ¿Qué valor devuelve `!true`?

a) true b) false c) 0 d) Error

PREGUNTA 34 (Abierta): Escribe un ciclo `for` que imprima solo los números pares del 2 al 20.

PREGUNTA 35 (Verdadero/Falso): Los parámetros son los valores que le pasas a una función al llamarla, y los argumentos son los nombres que aparecen en la declaración.

PREGUNTA 36 (Opción múltiple): ¿Qué hace `array.pop()`?

a) Agrega un elemento al final b) Elimina el último elemento y lo devuelve c) Elimina el primer elemento d) Vacía el array

PREGUNTA 37 (Abierta): Dado `let numeros = [1, 2, 3, 4];`, usa `map` para crear un nuevo array con el doble de cada número.

PREGUNTA 38 (Verdadero/Falso): `document.querySelector(".btn")` selecciona todos los elementos con la clase "btn".

PREGUNTA 39 (Opción múltiple): ¿Cómo lees lo que el usuario escribió en un `<input id="nombre">`?

a) `input.text` b) `document.getElementById("nombre").value` c) `input.innerHTML` d) `document.getElementById("nombre").text`

PREGUNTA 40 (Abierta): Escribe el código completo (HTML + JS) de un contador con un párrafo que muestre un número, un botón "Sumar" que lo incremente y un botón "Restar" que lo decremente.

PREGUNTA 41 (Opción múltiple): ¿Qué imprime `JSON.parse('{"edad": 25}')`?

a) Un texto b) Un objeto con la propiedad `edad` igual a 25 c) Error d) undefined

PREGUNTA 42 (Verdadero/Falso): `setInterval` ejecuta una función repetidamente cada cierto número de milisegundos hasta que se detiene con `clearInterval`.

PREGUNTA 43 (Abierta): Escribe una promesa que se resuelva después de 1 segundo con el texto "Listo", y usa `.then` para imprimirlo.

PREGUNTA 44 (Opción múltiple): ¿Cuál de estos es un nombre de variable VÁLIDO en JavaScript?

a) 2nombre b) mi-nombre c) _nombre2 d) var

PREGUNTA 45 (Verdadero/Falso): En una función flecha de una sola expresión como `const f = x => x * 2;`, el valor se devuelve automáticamente sin escribir `return`.

PREGUNTA 46 (Abierta): Encuentra y corrige el error en este código:

```
let total = 0
for (let i = 1; i <= 5; i++) {
  total = total + i;
  console.log("El total es: " + total);
}
```

PREGUNTA 47 (Opción múltiple): ¿Qué método usarías para detener un temporizador creado con `setInterval`?

a) `stopInterval()` b) `clearInterval()` c) `deleteInterval()` d) `killTimer()`

PREGUNTA 48 (Abierta): Escribe una función llamada `esMayorDeEdad` que reciba una edad y devuelva `true` si es 18 o más, y `false` en caso contrario. Usa una función flecha.

PREGUNTA 49 (Verdadero/Falso): `innerHTML` es seguro para mostrar texto escrito por usuarios porque escapa automáticamente las etiquetas.

PREGUNTA 50 (Abierta): Explica con tus palabras qué es el DOM y menciona dos cosas que puedes hacer con él usando JavaScript.

Respuestas y soluciones

RESPUESTA 1: b) `<script src="app.js" defer></script>`. El atributo `src` indica la ruta del archivo externo (no `href`, que es para enlaces y hojas de estilo), y `defer` le dice al navegador que descargue el script sin bloquear el HTML y lo ejecute hasta que el documento esté listo. La opción c) usa una etiqueta inexistente y la d) un atributo inventado.

RESPUESTA 2: Falso. `const` crea una constante: una vez asignado el valor no puedes reasignarlo. Si necesitas que el valor cambie, usa `let`.

RESPUESTA 3:

```
let nombre = "Esteban";  
const PI = 3.1416;
```

`let` porque el nombre puede cambiar; `const` porque el valor de PI nunca cambia. Los nombres en mayúsculas son una convención para constantes.

RESPUESTA 4: b) string. Las comillas hacen que "42" sea texto, aunque parezca número. `typeof` siempre devuelve el tipo real del valor, no lo que parece.

RESPUESTA 5: b) 1. El operador `%` devuelve el residuo de la división: 10 entre 3 es 3 con residuo 1. Es muy útil para saber si un número es par (`n% 2 === 0`) o para ciclos.

RESPUESTA 6:

```
if (edad >= 18) {  
  console.log("Puedes votar");  
} else {  
  console.log("Aún no puedes votar");  
}
```

El `if` evalúa la condición entre paréntesis; si es verdadera se ejecuta el primer bloque, si no, el del `else`. Fíjate en las llaves y en el `>=` (mayor o igual).

RESPUESTA 7: Falso. `"5" === 5` devuelve `false` porque el triple igual compara valor Y tipo: texto contra número son tipos distintos. Con doble igual (`==`) sí daría `true` porque solo compara el valor haciendo conversión.

RESPUESTA 8: b) `&&`. El doble ampersand es el Y lógico: ambas condiciones deben ser verdaderas. `||` es el O, y `!` es la negación.

RESPUESTA 9: b) 5. El ciclo va con `i` desde 0 hasta 4 (se detiene cuando `i < 5` es falso, es decir, cuando `i` llega a 5): 0, 1, 2, 3, 4. Son cinco iteraciones. Este es el error más común de los principiantes: contar una de más o una de menos.

RESPUESTA 10:

```
let i = 1;
while (i <= 10) {
  console.log(i);
  i++;
}
```

Tres partes indispensables: inicializar antes del ciclo, la condición, y modificar la variable DENTRO del ciclo. Si olvidas el `i++`, el ciclo nunca termina.

RESPUESTA 11: Verdadero. Las declaraciones de función con `function` se "elevan" (hoisting): JavaScript las registra antes de ejecutar el código, así que puedes llamarlas antes de la línea donde están escritas. Esto no funciona igual con funciones guardadas en variables.

RESPUESTA 12: b) return. `return` termina la función y entrega el valor a quien la llamó. Sin `return`, la función devuelve `undefined`.

RESPUESTA 13:

```
const doble = (numero) => numero * 2;
```

Como es una sola expresión, no necesita llaves ni `return`: la flecha devuelve el resultado automáticamente. Los paréntesis alrededor del parámetro son opcionales con un solo parámetro, pero ponerlos siempre es más claro.

RESPUESTA 14: b) pera. Los arrays empiezan en la posición 0: `frutas[0]` es "manzana", `frutas[1]` es "pera", `frutas[2]` es "uva".

RESPUESTA 15: Falso. `push()` agrega al FINAL del array. El que agrega al inicio es `unshift()`. Para quitar del final se usa `pop()` y para quitar del inicio `shift()`.

RESPUESTA 16: c) map devuelve un array nuevo transformado y `forEach` no devuelve nada útil. `forEach` sirve para hacer algo con cada elemento (como imprimirlos); `map` sirve para transformar cada elemento y obtener una lista nueva del mismo tamaño.

RESPUESTA 17:

```
console.log(persona.nombre);
```

La notación de punto es la forma más común de leer una propiedad: objeto, punto, nombre de la propiedad.

RESPUESTA 18: Verdadero. Dentro de un método, `this` apunta al objeto dueño del método. Por eso `this.nombre` dentro de `persona.saludar()` lee el nombre de `persona`.

RESPUESTA 19: c) Backticks (comillas invertidas). Solo con la comilla invertida (``) la sintaxis `${variable}` funciona. Con comillas simples o dobles, ``${nombre}`` se imprimiría literalmente como texto.

RESPUESTA 20:

```
const titulo = document.getElementById("titulo");
```

Sin el símbolo `#`: `getElementById` recibe el id puro. Usamos `const` porque la referencia al elemento no va a cambiar.

RESPUESTA 21: c) `innerHTML` interpreta etiquetas HTML y `textContent` las trata como texto plano. Si haces `elemento.textContent = "<b>hola</b>"`, el usuario verá literalmente `<b>hola</b>`; con `innerHTML` vería "hola" en negritas. Por seguridad, para texto de usuarios usa `textContent`.

RESPUESTA 22:

```
const miBoton = document.getElementById("miBoton");
miBoton.addEventListener("click", () => {
  alert("¡Hola!");
});
```

Primero seleccionas el botón, luego le agregas un "escuchador" del evento "click" con una función que se ejecuta cuando ocurre el clic.

RESPUESTA 23: Verdadero. Los formularios por defecto recargan la página al enviarse; `event.preventDefault()` cancela ese comportamiento para que JavaScript maneje los datos sin recargar.

RESPUESTA 24: b) Una sola vez después de 2 segundos. `setTimeout` es de una sola ejecución; el que repite es `setInterval`. El tiempo siempre va en milisegundos: `2000 ms = 2 segundos`.

RESPUESTA 25:

```
const texto = JSON.stringify({ nombre: "Ana" });
```

JSON.stringify convierte un objeto a texto JSON. Su contraparte, JSON.parse, hace lo inverso: de texto a objeto.

RESPUESTA 26: Falso. Una promesa empieza "pendiente" pero siempre termina en uno de dos estados: "cumplida" (se resolvió bien) o "rechazada" (falló). Nunca se queda pendiente para siempre en un programa correcto.

RESPUESTA 27: b) Para manejar errores si la promesa se rechaza. then maneja el caso de éxito y catch el de fracaso. Sin catch, un error de red quedaría sin manejar y sería difícil de depurar.

RESPUESTA 28:

```
async function cargarDatos() {  
  try {  
    const respuesta = await fetch("https://api.ejemplo.com/datos");  
    const datos = await respuesta.json();  
    console.log(datos);  
  } catch (error) {  
    console.log("Algo falló:", error);  
  }  
}
```

La función debe llevar async para poder usar await dentro. El try/catch atrapa errores de red o de respuesta. await pausa la función hasta que llega la respuesta, sin congelar el resto de la página.

RESPUESTA 29: b) Pausa la ejecución de la función async hasta que la promesa se resuelve, sin bloquear el resto del programa. Esa es la magia de async/await: el código se lee como si fuera secuencial, pero por dentro sigue siendo asíncrono y no congela la página.

RESPUESTA 30: Verdadero. let y const viven dentro del bloque {} donde se declaran; var se escapa del bloque y solo respeta funciones, lo que causa errores difíciles de rastrear. Por eso ya no se usa var.

RESPUESTA 31: b) "22". Cuando sumas un número con un texto, JavaScript convierte el número a texto y los concatena: "2" + "2" = "22". Es la famosa coerción de tipos; por eso hay que tener cuidado con valores que vienen de formularios (siempre son texto).

RESPUESTA 32: `==` compara solo el valor haciendo conversión de tipos automática (`"5" == 5` es `true`), mientras que `===` compara valor Y tipo sin conversiones (`"5" === 5` es `false`). Ejemplo: `0 == false` da `true`, pero `0 === false` da `false`. La buena práctica es usar siempre `===` para evitar sorpresas por conversiones invisibles.

RESPUESTA 33: b) `false`. El operador `!` niega el valor: `!true` es `false` y `!false` es `true`. También convierte a booleano: `!0` es `true` porque 0 es un valor "falsy".

RESPUESTA 34:

```
for (let i = 2; i <= 20; i += 2) {  
  console.log(i);  
}
```

En vez de incrementar de 1 en 1 y filtrar, incrementamos de 2 en 2 desde el 2: así solo tocamos pares. Es más eficiente y más claro.

RESPUESTA 35: Falso, está al revés. Los **PARÁMETROS** son los nombres que aparecen en la declaración de la función (`function sumar(a, b)`), y los **ARGUMENTOS** son los valores que le pasas al llamarla (`sumar(3, 5)`). Truco para recordarlo: el parámetro es el "lugar" y el argumento es el "valor que ocupa el lugar".

RESPUESTA 36: b) Elimina el último elemento y lo devuelve. `push` y `pop` trabajan al final del array; `unshift` y `shift` al inicio. Que `pop` devuelva el elemento eliminado es útil: `let ultimo = frutas.pop();`.

RESPUESTA 37:

```
const dobles = numeros.map(n => n * 2);
```

`map` recorre cada número, le aplica la función y arma un array nuevo `[2, 4, 6, 8]`. El array original `numeros` no se modifica.

RESPUESTA 38: Falso. `querySelector` devuelve solo el **PRIMER** elemento que coincida. Para obtener todos se usa `querySelectorAll`, que devuelve una lista de elementos.

RESPUESTA 39: b) `document.getElementById("nombre").value`. Los inputs guardan lo que escribe el usuario en su propiedad `value`, no en `textContent` ni `innerHTML`.

RESPUESTA 40:

```
<p id="numero">0</p>
<button id="btnSumar">Sumar</button>
<button id="btnRestar">Restar</button>
```

```
let contador = 0;
const numero = document.getElementById("numero");
document.getElementById("btnSumar").addEventListener("click", () => {
  contador++;
  numero.textContent = contador;
});
document.getElementById("btnRestar").addEventListener("click", () => {
  contador--;
  numero.textContent = contador;
});
```

La variable `contador` vive fuera de las funciones para que conserve su valor entre clics. Cada clic actualiza la variable Y el texto en pantalla: si olvidas actualizar el `textContent`, el número interno cambia pero el usuario no lo ve.

RESPUESTA 41: b) Un objeto con la propiedad `edad` igual a 25. `JSON.parse` convierte texto JSON en un valor real de JavaScript: aquí un objeto `{edad: 25}` donde `edad` ya es número, no texto.

RESPUESTA 42: Verdadero. `setInterval(fn, 1000)` ejecuta `fn` cada segundo indefinidamente. Para detenerlo guardas su identificador (`const id = setInterval(...)`) y luego llamas `clearInterval(id)`.

RESPUESTA 43:

```
const miPromesa = new Promise((resolve) => {
  setTimeout(() => resolve("Listo"), 1000);
});
miPromesa.then((mensaje) => {
  console.log(mensaje);
});
```

La promesa envuelve algo que tarda (aquí un `setTimeout` de 1 segundo). Cuando termina, llama a `resolve("Listo")`, y ese valor llega al `.then` como `mensaje`.

RESPUESTA 44: c) `_nombre2`. Las reglas: no puede empezar con número (a falla), no puede llevar guiones (b falla, el guion se interpreta como resta), y no puede ser palabra

reservada como `var` (d falla). El guion bajo y los números después del primer carácter sí están permitidos.

RESPUESTA 45: Verdadero. Con la sintaxis de una sola expresión, la flecha devuelve el resultado implícitamente. Solo necesitas llaves y `return` cuando la función tiene varias líneas.

RESPUESTA 46: Falta cerrar la llave del `for`. El código correcto:

```
let total = 0;
for (let i = 1; i <= 5; i++) {
  total = total + i;
}
console.log("El total es: " + total);
```

Faltaba la `}` que cierra el ciclo antes del `console.log`. Además, el `console.log` debe ir FUERA del ciclo: si lo dejas adentro, imprime el total parcial en cada vuelta (1, 3, 6, 10, 15) en vez del total final (15). Siempre cuenta tus llaves de apertura y cierre.

RESPUESTA 47: b) `clearInterval()`. El patrón completo es guardar el id (`const id = setInterval(...)`) y luego `clearInterval(id)`. Su pareja para `setTimeout` es `clearTimeout`.

RESPUESTA 48:

```
const esMayorDeEdad = (edad) => edad >= 18;
```

La comparación `edad >= 18` ya devuelve `true` o `false` por sí sola, así que no necesitas un `if`: la devuelves directo. Es el ejemplo perfecto de `return` implícito en una función flecha.

RESPUESTA 49: Falso. Es al revés: `innerHTML` interpreta las etiquetas, así que si un usuario escribe `<script>` o HTML malicioso, se ejecutaría. Para mostrar texto de usuarios usa siempre `textContent`, que lo trata como texto inofensivo.

RESPUESTA 50: El DOM (Document Object Model) es la representación en forma de árbol que el navegador crea de tu página HTML, donde cada etiqueta es un "nodo" que JavaScript puede leer y modificar. Dos cosas que puedes hacer con él: seleccionar elementos (con `getElementById` o `querySelector`) y modificarlos (cambiar su `textContent`, su `innerHTML`, sus estilos o sus atributos), además de reaccionar a eventos del usuario con `addEventListener`.

¡Examen terminado! Cuenta tus aciertos: 45 o más, excelente; 40 a 44, buen nivel; 35 a 39, repasa las unidades donde fallaste; menos de 35, relee el libro con calma y vuelve a intentarlo. Recuerda: el objetivo no es la calificación, es que el conocimiento se quede contigo. Sigue practicando en lucasegaacademy.com.

Colibrí Editorial EGA

Curso Completo de JavaScript

Curso Completo de JavaScript nació del curso en video de 30 clases de lucasegaacademy.com. En 10 unidades explica cada tema a fondo —desde variables y funciones hasta objetos, arreglos, DOM y asincronía— con ejemplos que puedes copiar y correr, ejercicios para practicar de verdad y un examen final de 50 preguntas de opción múltiple, verdadero o falso y código abierto, con todas las respuestas explicadas.

lucasegaacademy.com