

Ejercicios resueltos de Desarrollo Web en Entorno Cliente

Los 10 ejercicios de Desarrollo Web en Entorno Cliente de la web, tema a tema: cada uno con su enunciado, los datos que necesitas (código de partida, ejemplos de entrada y salida o la base de datos) y, al final, la solución explicada.

Índice

Fundamentos de JavaScript

1. Encuentra el fallo: los turnos que se repiten

2. Encuentra el fallo: el pago que nunca cuadra

Objetos Predefinidos y BOM

3. Encuentra el fallo: las notas mal ordenadas

Funciones, Objetos y Clases

4. Encuentra el fallo: la copia que no es una copia

5. Encuentra el fallo: el contador que no cuenta

Asincronía, AJAX y Frameworks

6. Encuentra el fallo: el total que es una promesa

Ejercicios largos

7. Calendario de un mes con festivos y puentes

8. Reservas de pistas de pádel con clases

9. Validador de formularios de registro

10. Cargador de datos asíncrono con reintentos y dependencias

Fundamentos de JavaScript

1. Encuentra el fallo: los turnos que se repiten

Medio · Fundamentos de JavaScript · apuntesdam.com/subject/dwec/topic/fundamentos-javascript

El programa programa tres avisos con `setTimeout` y cada uno debería decir su número: «Turno 1», «Turno 2» y «Turno 3». En lugar de eso, los tres dicen «Turno 4». Corrígelo cambiando una sola palabra.

Código de partida (javascript)

```
for (var i = 1; i <= 3; i++) {  
  setTimeout(() => console.log(`Turno ${i}`), 10 * i);  
}
```

Ejemplo: Tres turnos

SALIDA ESPERADA

```
Turno 1  
Turno 2  
Turno 3
```

2. Encuentra el fallo: el pago que nunca cuadra

Medio · Fundamentos de JavaScript · apuntesdam.com/subject/dwec/topic/fundamentos-javascript

El carrito tiene dos artículos de 0,10 € y 0,20 €, y el cliente paga 0,30 €. El programa dice que falta dinero. Corrige la comparación para que funcione con importes en euros.

Código de partida (javascript)

```
const carrito = [0.1, 0.2];  
const total = carrito.reduce((a, b) => a + b, 0);  
const pagado = 0.3;  
console.log(total === pagado ? "Pago correcto" : "Falta dinero");
```

Ejemplo: 0,10 + 0,20 pagando 0,30

SALIDA ESPERADA

```
Pago correcto
```

Objetos Predefinidos y BOM

3. Encuentra el fallo: las notas mal ordenadas

Fácil · Objetos Predefinidos y BOM · apuntesdam.com/subject/dwec/topic/objetos-predefinidos

El programa ordena unas puntuaciones de menor a mayor, pero el 10 y el 25 salen en mal sitio: «10, 2, 25, 7, 9». Corrígelo.

Código de partida (javascript)

```
const notas = [10, 9, 2, 7, 25];  
notas.sort();  
console.log(notas.join(", "));
```

Ejemplo: Cinco puntuaciones

SALIDA ESPERADA

2, 7, 9, 10, 25

Funciones, Objetos y Clases

4. Encuentra el fallo: la copia que no es una copia

Fácil · Funciones, Objetos y Clases · apuntesdam.com/subject/dwec/topic/funciones-objetos

El programa hace una copia de la lista para añadir un elemento sin tocar la original. Al final las dos listas tienen 4 elementos, cuando la original debería seguir con 3. Corrígelo.

Código de partida (javascript)

```
const original = ["lunes", "martes", "miércoles"];
const copia = original;
copia.push("jueves");
console.log(`Original: ${original.length}, copia: ${copia.length}`);
```

Ejemplo: Longitudes

SALIDA ESPERADA

Original: 3, copia: 4

5. Encuentra el fallo: el contador que no cuenta

Difícil · Funciones, Objetos y Clases · apuntesdam.com/subject/dwec/topic/funciones-objetos

El método sumarTodos debería acumular en this.cuenta los números recibidos y devolver 6 para [1, 2, 3], pero devuelve 0. Corrígelo.

Código de partida (javascript)

```
const contador = {
  cuenta: 0,
  sumarTodos(numeros) {
    numeros.forEach(function (n) {
      this.cuenta += n;
    });
    return this.cuenta;
  },
};
console.log(contador.sumarTodos([1, 2, 3]));
```

Ejemplo: Sumar 1, 2 y 3

SALIDA ESPERADA

6

Asincronía, AJAX y Frameworks

6. Encuentra el fallo: el total que es una promesa

Medio · Asincronía, AJAX y Frameworks · apuntesdam.com/subject/dwec/topic/asincronia-ajax

precioConIva es una función asíncrona. El programa debería mostrar «Total: 121 €», pero muestra «Total: [object Promise] €». Corrígelo.

Código de partida (javascript)

```
async function precioConIva(precio) {
  return precio * 1.21;
}

async function main() {
  const total = precioConIva(100);
  console.log(`Total: ${total} €`);
}

main();
```

Ejemplo: 100 € con IVA

SALIDA ESPERADA

Total: 121 €

Ejercicios largos

7. Calendario de un mes con festivos y puentes

Medio · JavaScript · 50 minutos · apuntesdam.com/ejercicios/javascript/calendario-con-festivos-y-puentes

La intranet de una empresa quiere un calendario sencillo en el que cada departamento marque sus festivos y vea de un vistazo cuántos días se trabajan en el mes y qué puentes se pueden pedir. Antes de dibujarlo en la página, la lógica se escribe y se prueba en la consola con Node.js.

En España las semanas empiezan en lunes, pero `getDay()` numera los días con el domingo como 0. Además, `new Date(2026, 1, 1)` crea la fecha en la hora local del ordenador, y según la zona horaria el día puede cambiar al pasarla a UTC. Usar `Date.UTC` y los métodos `getUTC...` evita las dos trampas.

Un puente es un día laborable que queda encerrado entre un festivo y el fin de semana: si un festivo cae en martes, el lunes anterior; si cae en jueves, el viernes siguiente.

Requisitos

- La primera línea es `año mes` (año de 1900 a 2100, mes de 1 a 12); si no es válida, escribe `Fecha no válida` y nada más. Cada línea siguiente es un festivo: `día nombre`. Una línea con un día que no existe en ese mes, o sin nombre, escribe `Aviso: línea N ignorada` (las líneas vacías no avisan). Si un día se repite, vale el último nombre.
- Escribe el título (`Diciembre de 2026`) y la cabecera `L M X J V S D`. Después, una línea por semana: cada día ocupa 3 caracteres, el número alineado a la derecha en 2 y un `*` si es festivo o un espacio si no. La primera semana empieza con 3 espacios por cada día del mes anterior. Ninguna línea termina en espacios.
- Festivos:** y, por orden de día, una línea por festivo con dos espacios delante: `8 Inmaculada Concepción (martes)`. Si no hay ninguno, `Festivos: ninguno`.
- Días laborables:** `N`: los días de lunes a viernes que no son festivos.
- Puentes posibles:** `lunes 7, viernes 24`: los lunes anteriores a un festivo en martes y los viernes posteriores a un festivo en jueves, siempre que sean del mismo mes y no sean festivos, por orden de día y sin repetir. Si no hay ninguno, `Puentes posibles: ninguno`.

Formato de la entrada

- Línea 1: `año mes` . Resto: un festivo por línea, `día nombre` .

Ejemplo: Diciembre con tres festivos

ENTRADA

```
2026 12
6 Día de la Constitución
8 Inmaculada Concepción
25 Navidad
```

SALIDA ESPERADA

```
Diciembre de 2026
L M X J V S D
    1 2 3 4 5 6*
 7 8* 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25*26 27
28 29 30 31
Festivos:
 6 Día de la Constitución (domingo)
 8 Inmaculada Concepción (martes)
25 Navidad (viernes)
Días laborables: 21
Puentes posibles: lunes 7
```

Ejemplo: Mayo con líneas que no valen

ENTRADA

```
2026 5
1 Fiesta del Trabajo
2 Comunidad de Madrid
40 Día imposible
abc
15 San Isidro
```

SALIDA ESPERADA

```
Aviso: línea 4 ignorada
Aviso: línea 5 ignorada
Mayo de 2026
L M X J V S D
    1* 2* 3
 4 5 6 7 8 9 10
11 12 13 14 15*16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
Festivos:
 1 Fiesta del Trabajo (viernes)
 2 Comunidad de Madrid (sábado)
15 San Isidro (viernes)
Días laborables: 19
Puentes posibles: ninguno
```

Código de partida (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const MESES = ["Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto", "Septiembre",
  "Octubre", "Noviembre", "Diciembre"];
const DIAS = ["lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"];

function main() {
  const [anio, mes] = (lineas[0] ?? "").trim().split(/\s+/).map(Number);
  // TODO: valida la fecha, lee los festivos y dibuja el calendario
  // TODO: festivos, días laborables y puentes posibles
}

main();
```

8. Reservas de pistas de pádel con clases

Difícil · JavaScript · 70 minutos · apuntesdam.com/ejercicios/javascript/reservas-de-pistas-de-padel

Un club de pádel quiere una web para reservar pistas desde el móvil. Antes de hacer la interfaz, el equipo escribe el «motor» de reservas como una clase que no sabe nada del HTML: recibe órdenes y devuelve textos. Así se puede probar en la consola con Node.js y después usarla tal cual en la página.

El club tiene 3 pistas y nueve turnos de 90 minutos cada día, de las 09:00 a las 22:30. Por la mañana y a primera hora de la tarde (turnos de las 09:00 a las 15:00) es hora valle y la pista cuesta 12 €; desde las 16:30, hora punta, 18 €. Los socios tienen un 30 % de descuento, y para que todo el mundo pueda jugar nadie puede tener más de dos reservas a la vez.

Requisitos

- Cada línea es una orden; la palabra de la orden puede ir en mayúsculas o minúsculas. Las líneas vacías se ignoran y una orden desconocida escribe `Orden desconocida: palabra`.
- `SOCIOS Ana,Luis Mora` añade socios (nombres separados por comas, sin espacios alrededor) y escribe `Socios: N` con el total de socios.
- `RESERVA pista hora nombre` (el nombre puede tener espacios) comprueba, en este orden: que la pista sea 1, 2 o 3 (`Pista no válida: X`), que la hora sea uno de los turnos (`Hora no válida: X`), que la pista esté libre a esa hora (`La pista P ya está reservada a las HH:MM`) y que esa persona no tenga ya 2 reservas (`Nombre ya tiene 2 reservas`). Si todo va bien, la reserva recibe un número (1, 2, 3... sin reutilizar los cancelados) y se escribe `Reserva N: pista P a las HH:MM para Nombre (X €)`.
- `CANCELA n` escribe `Reserva n cancelada` y libera la pista, o `No existe la reserva n`.
- `LIBRES pista` escribe `Pista P libre a las 09:00, 10:30, ...` con sus turnos libres en orden, o `Pista P sin horas libres`, o `Pista no válida: X`.
- `RESUMEN` escribe una línea por pista, `Pista P: R de 9 horas (Q %)` con el porcentaje redondeado al entero; `Recaudación prevista: X €` con la suma de las reservas vigentes; y `Mejor cliente: Nombre (N reservas)` (la persona con más reservas vigentes; si empatan, la primera alfabéticamente; «reserva» si es 1), o `Mejor cliente: ninguno`.
- Los importes se escriben con dos decimales y coma decimal: `12,60 €`.

Formato de la entrada

- Una orden por línea: `SOCIOS`, `RESERVA`, `CANCELA`, `LIBRES` o `RESUMEN`.

Turnos y precios

Turnos	Tipo	Precio	Socios (-30 %)
09:00, 10:30, 12:00, 13:30, 15:00	valle	12,00 €	8,40 €
16:30, 18:00, 19:30, 21:00	punta	18,00 €	12,60 €

Ejemplo: Una mañana de reservas

ENTRADA

SOCIOS Ana,Luis Mora
 RESERVA 2 18:00 Ana
 RESERVA 1 09:00 Pedro Gil
 RESERVA 2 18:00 Pedro Gil
 RESERVA 3 10:30 Ana
 RESERVA 1 12:00 Ana
 RESERVA 4 12:00 Eva
 RESERVA 1 11:00 Eva
 CANCELA 2
 CANCELA 9
 LIBRES 1
 RESERVA 1 21:00 Luis Mora
 RESUMEN

SALIDA ESPERADA

Socios: 2
 Reserva 1: pista 2 a las 18:00 para Ana (12,60 €)
 Reserva 2: pista 1 a las 09:00 para Pedro Gil (12,00 €)
 La pista 2 ya está reservada a las 18:00
 Reserva 3: pista 3 a las 10:30 para Ana (8,40 €)
 Ana ya tiene 2 reservas
 Pista no válida: 4
 Hora no válida: 11:00
 Reserva 2 cancelada
 No existe la reserva 9
 Pista 1 libre a las 09:00, 10:30, 12:00, 13:30, 15:00, 16:30, 18:00, 19:30, 21:00
 Reserva 4: pista 1 a las 21:00 para Luis Mora (12,60 €)
 Pista 1: 1 de 9 horas (11 %)
 Pista 2: 1 de 9 horas (11 %)
 Pista 3: 1 de 9 horas (11 %)
 Recaudación prevista: 33,60 €
 Mejor cliente: Ana (2 reservas)

Ejemplo: Órdenes en minúsculas

ENTRADA

reserva 3 15:00 Marta
 libres 3
 resumen
 borrar 1

SALIDA ESPERADA

Reserva 1: pista 3 a las 15:00 para Marta (12,00 €)
 Pista 3 libre a las 09:00, 10:30, 12:00, 13:30, 16:30, 18:00, 19:30, 21:00
 Pista 1: 0 de 9 horas (0 %)
 Pista 2: 0 de 9 horas (0 %)
 Pista 3: 1 de 9 horas (11 %)
 Recaudación prevista: 12,00 €
 Mejor cliente: Marta (1 reserva)
 Orden desconocida: borrar

```

const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const HORAS = ["09:00", "10:30", "12:00", "13:30", "15:00", "16:30", "18:00", "19:30", "21:00"];
const PISTAS = 3;

const euros = (centimos) => `${(centimos / 100).toFixed(2).replace(".", ",")} €`;

class Reserva {
  constructor(id, pista, hora, nombre, precio) {
    this.id = id;
    this.pista = pista;
    this.hora = hora;
    this.nombre = nombre;
    this.precio = precio;
  }
}

class Club {
  #reservas = new Map(); // id → Reserva
  #siguiente = 1;
  #socios = new Set();

  // TODO: anadirSocios, reservar, cancelar, libres y resumen (cada uno devuelve el texto que se escribe)
}

const club = new Club();
for (const linea of lineas) {
  // TODO: SOCIOS, RESERVA, CANCELA, LIBRES y RESUMEN
}

```

9. Validador de formularios de registro

Difícil · JavaScript · 65 minutos · apuntesdam.com/ejercicios/javascript/validador-de-formularios-de-registro

Una web de cursos recibe cientos de altas al día. El formulario ya valida en el navegador, pero eso se puede saltar (basta con desactivar JavaScript o enviar la petición a mano), así que el equipo quiere la misma validación en una función que se pueda ejecutar en el cliente y en el servidor con Node.js. La prueba se hace con un fichero de formularios en JSON, uno por línea.

Cada campo tiene sus reglas y el usuario debe recibir un único mensaje por campo, el del primer problema que se encuentre: no tiene sentido decirle a la vez que la contraseña es corta y que le faltan números. Los nombres pueden tener tildes, eñes, apóstrofes (O'Brien) y guiones (Ruiz-Pérez), así que `[a-zA-Z]` no sirve: hace falta la clase Unicode `\p{L}`.

Para que los resultados no dependan del día en que se ejecuten, la edad se calcula a una fecha de referencia fija: el 1 de septiembre de 2026.

Requisitos

- Cada línea no vacía es un formulario en JSON. Si no es JSON válido, o no es un objeto, escribe **Formulario N: JSON no válido** (N cuenta todas las líneas no vacías). Los valores se convierten a texto y se les quitan los espacios de los extremos; un campo que falta o es null cuenta como vacío.
- **nombre**: obligatorio; entre 2 y 40 caracteres; empieza por una letra y solo tiene letras (de cualquier idioma), espacios, apóstrofes y guiones (**solo letras, espacios, apóstrofes y guiones**).
- **email**: obligatorio; algo sin espacios ni arrobas, una `@`, algo sin espacios ni arrobas, un punto y al menos dos letras al final (**formato no válido**).
- **password**: obligatoria; al menos 8 caracteres; con mayúsculas, minúsculas y números (**necesita mayúsculas, minúsculas y números**); y si la primera palabra del nombre tiene 3 letras o más, no puede contenerla sin distinguir mayúsculas (**no puede contener tu nombre**). **confirmacion**: si hay contraseña, tiene que ser igual (**no coincide con la contraseña**).

- **nacimiento** : obligatoria, con formato **AAAA-MM-DD** y una fecha que exista (**fecha no válida**); la persona tiene que haber cumplido 16 años el 1 de septiembre de 2026 (**hay que tener al menos 16 años**).
- **cp** : obligatorio; 5 cifras con las dos primeras entre 01 y 52 (**código postal no válido**). **telefono** : opcional; si se escribe, quitando los espacios deben quedar 9 cifras que empiecen por 6, 7, 8 o 9 (**teléfono no válido**). Un campo vacío obligatorio da **obligatorio** .
- Para cada formulario: **Formulario N: correcto** , o **Formulario N: E errores** («error» si es uno) seguido de una línea por campo con error, en el orden nombre, email, password, confirmacion, nacimiento, cp, telefono: **- campo: mensaje** . Las reglas de cada campo se comprueban en el orden en que aparecen aquí y solo se muestra el primer fallo.
- Al final, **Correctos: C de N** y **Campo con más errores: campo (veces)** (si empatan, el primero en el orden de los campos) o **Campo con más errores: ninguno** .

Formato de la entrada

- Un formulario por línea, como objeto JSON con los campos nombre, email, password, confirmacion, nacimiento, cp y telefono.

Ejemplo: Tres altas

ENTRADA

```
{ "nombre": "Lucía Martín-Ruiz", "email": "lucia@correo.es", "password": "Clave2026", "confirmacion": "Clave2026", "nacimiento": "2004-05-17", "cp": "28013", "telefono": "612 345 678" }
{ "nombre": "Ana", "email": "ana@correo", "password": "anaPerez99", "confirmacion": "anaperez99", "nacimiento": "2011-02-30", "cp": "53001" }
{ "nombre": "O'Brien", "email": "obrien@mail.ie", "password": "Dublin1916", "confirmacion": "Dublin1916", "nacimiento": "2010-09-01", "cp": "8001" }
```

SALIDA ESPERADA

```
Formulario 1: correcto
Formulario 2: 5 errores
- email: formato no válido
- password: no puede contener tu nombre
- confirmacion: no coincide con la contraseña
- nacimiento: fecha no válida
- cp: código postal no válido
Formulario 3: 1 error
- cp: código postal no válido
Correctos: 1 de 3
Campo con más errores: cp (2)
```

Ejemplo: Formularios incompletos o mal escritos

ENTRADA

```
{ "email": " ", "password": "corta" }
{ nombre: "sin comillas" }
[1, 2, 3]
{ "nombre": "Ñuño Álvarez", "email": "nuno@x.com", "password": "abcdefgh", "nacimiento": "2010-08-31", "cp": "01001", "telefono": "91234567" }
```

SALIDA ESPERADA

```
Formulario 1: 6 errores
- nombre: obligatorio
- email: obligatorio
- password: al menos 8 caracteres
- confirmacion: no coincide con la contraseña
- nacimiento: obligatorio
- cp: obligatorio
Formulario 2: JSON no válido
Formulario 3: JSON no válido
Formulario 4: 3 errores
- password: necesita mayúsculas, minúsculas y números
- confirmacion: no coincide con la contraseña
- telefono: teléfono no válido
Correctos: 0 de 4
Campo con más errores: password (2)
```

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const HOY = new Date(Date.UTC(2026, 8, 1)); // fecha de referencia: 1 de septiembre de 2026
const CAMPOS = ["nombre", "email", "password", "confirmacion", "nacimiento", "cp", "telefono"];

/** Errores de un formulario: un objeto campo → mensaje, como mucho uno por campo. */
function validar(datos) {
  const errores = {};
  // TODO: las reglas de cada campo
  return errores;
}

// TODO: lee cada línea como JSON, valida y escribe el resultado y el resumen
```

10. Cargador de datos asíncrono con reintentos y dependencias

Muy difícil · JavaScript · 90 minutos · apuntesdam.com/ejercicios/javascript/cargador-asincrono-con-reintentos

El panel de control de una tienda online pide al arrancar los usuarios, los productos, los pedidos, las estadísticas... cada dato a una API distinta. Pedirlos uno detrás de otro haría esperar al usuario la suma de todos los tiempos; pedirlos a la vez, solo lo que tarde el más lento.

Pero las redes fallan. Una petición puede devolver un error (y quizá funcione si se repite) o quedarse colgada para siempre: por eso cada petición tiene un tiempo límite de 100 ms y se reintenta hasta 3 veces. Además, algunos datos dependen de otros (los pedidos necesitan los usuarios): esos tienen que esperar a sus dependencias, y si alguna falla, no se piden.

Como no hay APIs reales, se simulan con `setTimeout`: cada recurso tarda unos milisegundos y falla las primeras veces que se indique. El informe final va en el orden de la entrada, no en el de llegada, para que sea siempre el mismo.

Requisitos

- Cada línea es `nombre ms fallos [dependencias]`: lo que tarda cada petición en milisegundos, cuántas de las primeras peticiones fallan con `error del servidor` y, opcionalmente, los nombres de los recursos de los que depende separados por comas (sin espacios). Las líneas vacías se ignoran. Una línea con otro formato o con un nombre repetido escribe `Aviso: línea N ignorada`. No hay dependencias circulares.
- Todos los recursos empiezan a cargarse a la vez, salvo los que tienen dependencias, que esperan a que terminen todas ellas. Si alguna dependencia no se ha cargado (o no existe), el recurso no se pide.
- Cada intento dura lo que diga `ms`, pero si pasan 100 ms sin respuesta se da por fallido con `tiempo agotado`. Se hacen hasta 3 intentos, uno detrás de otro, y el motivo de error que se informa es el del último.
- Al terminar todo, una línea por recurso en el orden de la entrada: `nombre: cargado (N intentos)` («intento» si es 1), `nombre: error tras 3 intentos (motivo)` u `nombre: omitido (depende de dep, que no se ha podido cargar)` / `(depende de dep, que no existe)`, indicando la primera dependencia de su lista que ha fallado.
- Para terminar, `Cargados: C de T · Peticiones: P`, donde P es el número total de peticiones simuladas que se han hecho.

Formato de la entrada

- Un recurso por línea: `nombre ms fallos` y, opcionalmente, `dep1,dep2`.

Ejemplo: Un panel con seis recursos

ENTRADA

```
usuarios 30 0
productos 20 2
fotos 150 0
pedidos 10 0 usuarios,productos
galeria 10 0 fotos
informe 10 0 pedidos,stock
```

SALIDA ESPERADA

```
usuarios: cargado (1 intento)
productos: cargado (3 intentos)
fotos: error tras 3 intentos (tiempo agotado)
pedidos: cargado (1 intento)
galeria: omitido (depende de fotos, que no se ha
podido cargar)
informe: omitido (depende de stock, que no
existe)
Cargados: 3 de 6 · Peticiones: 8
```

Ejemplo: Errores y líneas no válidas

ENTRADA

```
a 10 3
b 10 1
b 20 0
c diez 0
d 5 0 a
```

SALIDA ESPERADA

```
Aviso: línea 3 ignorada
Aviso: línea 4 ignorada
a: error tras 3 intentos (error del servidor)
b: cargado (2 intentos)
d: omitido (depende de a, que no se ha podido
cargar)
Cargados: 1 de 3 · Peticiones: 5
```

Código de partida (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const LIMITE_MS = 100;
const INTENTOS = 3;
let peticiones = 0;

/** Petición simulada: tarda «ms» y falla mientras el intento sea menor o igual que «fallos». */
function peticion(recurso, intento) {
  peticiones++;
  return new Promise((resolve, reject) => {
    setTimeout(() => (intento <= recurso.fallos ? reject(new Error("error del servidor")) :
      resolve(recurso.nombre)), recurso.ms);
  });
}

/** La primera promesa que termine gana: la petición o el temporizador. */
function conTiempoLimite(promesa, ms) {
  // TODO
}

async function conReintentos(recurso) {
  // TODO
}

async function main() {
  // TODO: lee los recursos, cárgalos respetando las dependencias y escribe el informe
}

main();
```

Soluciones

1. Encuentra el fallo: los turnos que se repiten

Medio · Fundamentos de JavaScript · apuntesdam.com/subject/dwec/topic/fundamentos-javascript

Solución (javascript)

```
for (let i = 1; i <= 3; i++) {  
  setTimeout(() => console.log(`Turno ${i}`), 10 * i);  
}
```

Con `var` solo existe una variable `i` para todo el bucle. Las funciones de `setTimeout` se ejecutan después de que el bucle termine, y para entonces esa única `i` vale 4, así que las tres muestran «Turno 4».

Con `let`, cada vuelta del bucle tiene su propia `i`, y cada función recuerda (gracias al cierre o *closure*) la de su vuelta. Es una de las razones por las que hoy se usa `let` en lugar de `var`.

2. Encuentra el fallo: el pago que nunca cuadra

Medio · Fundamentos de JavaScript · apuntesdam.com/subject/dwec/topic/fundamentos-javascript

Solución (javascript)

```
const carrito = [0.1, 0.2];  
const total = carrito.reduce((a, b) => a + b, 0);  
const pagado = 0.3;  
// Se comparan céntimos enteros: 0.1 + 0.2 vale 0.30000000000000004 en coma flotante  
console.log(Math.round(total * 100) === Math.round(pagado * 100) ? "Pago correcto" : "Falta dinero");
```

Los números de JavaScript son de coma flotante binaria, y 0,1 y 0,2 no tienen representación exacta: su suma es 0.30000000000000004, que no es exactamente igual a 0,3. Comparar decimales con `===` es casi siempre un error.

Con dinero lo más fiable es trabajar en céntimos enteros: `Math.round(x * 100)` convierte cada importe en un entero de céntimos, y los enteros sí se comparan sin problemas. Otra opción es comprobar que la diferencia sea menor que medio céntimo.

3. Encuentra el fallo: las notas mal ordenadas

Fácil · Objetos Predefinidos y BOM · apuntesdam.com/subject/dwec/topic/objetos-predefinidos

Solución (javascript)

```
const notas = [10, 9, 2, 7, 25];  
notas.sort((a, b) => a - b);  
console.log(notas.join(", "));
```

Sin argumentos, `sort()` convierte los elementos en texto y los ordena alfabéticamente: «10» va antes que «2» porque el carácter «1» va antes que el «2», igual que en un diccionario.

Para ordenar números hay que pasarle una función de comparación: `(a, b) => a - b` devuelve un negativo si `a` va antes, un positivo si va después y 0 si son iguales, que es justo lo que `sort` necesita.

4. Encuentra el fallo: la copia que no es una copia

Fácil · Funciones, Objetos y Clases · apuntesdam.com/subject/dwec/topic/funciones-objetos

Solución (javascript)

```
const original = ["lunes", "martes", "miércoles"];  
const copia = [...original];  
copia.push("jueves");  
console.log(`Original: ${original.length}, copia: ${copia.length}`);
```

Asignar un array a otra variable no lo copia: las dos variables apuntan al mismo array en memoria, y lo que se añade por una se ve por la otra. Con la sintaxis de propagación, `[...original]`, se crea un array nuevo con los mismos elementos.

Es una copia superficial: si los elementos fueran objetos, los dos arrays compartirían esos objetos. Para copias profundas existe `structuredClone`.

5. Encuentra el fallo: el contador que no cuenta

Difícil · Funciones, Objetos y Clases · apuntesdam.com/subject/dwec/topic/funciones-objetos

Solución (javascript)

```
const contador = {
  cuenta: 0,
  sumarTodos(numeros) {
    numeros.forEach((n) => {
      this.cuenta += n;
    });
    return this.cuenta;
  },
};
console.log(contador.sumarTodos([1, 2, 3]));
```

Dentro de una función normal (`function`), `this` depende de cómo se llama la función, y `forEach` la llama sin objeto: `this` no es `contador`, así que el método nunca suma nada en `contador.cuenta`.

Las funciones flecha no tienen su propio `this`: usan el del código que las rodea, que aquí es el método `sumarTodos`, donde `this` sí es `contador`. Por eso basta con cambiar `function (n) { ... }` por `(n) => { ... }`.

6. Encuentra el fallo: el total que es una promesa

Medio · Asincronía, AJAX y Frameworks · apuntesdam.com/subject/dwec/topic/asincronia-ajax

Solución (javascript)

```
async function precioConIva(precio) {
  return precio * 1.21;
}

async function main() {
  const total = await precioConIva(100);
  console.log(`Total: ${total} €`);
}
main();
```

Una función `async` siempre devuelve una promesa, aunque dentro haga un `return` de un número. Sin `await`, `total` es la promesa, y al meterla en el texto se muestra como «[object Promise]».

`await` espera a que la promesa se cumpla y devuelve su valor; por eso solo se puede usar dentro de una función `async`, como `main`.

7. Calendario de un mes con festivos y puentes

Medio · JavaScript · 50 minutos · apuntesdam.com/ejercicios/javascript/calendario-con-festivos-y-puentes

Solución (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const MESES = ["Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto", "Septiembre",
  "Octubre", "Noviembre", "Diciembre"];
const DIAS = ["lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"];

function main() {
  const [anio, mes] = (lineas[0] ?? "").trim().split(/\s+/).map(Number);
  if (!Number.isInteger(anio) || !Number.isInteger(mes) || anio < 1900 || anio > 2100 || mes < 1 || mes > 12) {
    console.log("Fecha no válida");
    return;
  }

  // Con UTC no influye la zona horaria del ordenador. El día 0 del mes siguiente es el último de este
  const diasMes = new Date(Date.UTC(anio, mes, 0)).getUTCDate();
  // getUTCDay da 0 = domingo; se pasa a 0 = lunes ... 6 = domingo
  const diaSemana = (d) => (new Date(Date.UTC(anio, mes - 1, d)).getUTCDay() + 6) % 7;

  const festivos = new Map();
  lineas.slice(1).forEach((linea, i) => {
    const texto = linea.trim();
    if (!texto) return;
    const m = texto.match(/^(d{1,2})\s+(.)$/);
    const dia = m ? Number(m[1]) : 0;
    if (!m || dia < 1 || dia > diasMes) {
      console.log(`Aviso: línea ${i + 2} ignorada`);
      return;
    }
    festivos.set(dia, m[2]);
  });

  console.log(`${MESES[mes - 1]} de ${anio}`);
  console.log(" L M X J V S D");
  let fila = " ".repeat(diaSemana(1));
  for (let d = 1; d <= diasMes; d++) {
    fila += String(d).padStart(2) + (festivos.has(d) ? "*" : " ");
    if (diaSemana(d) === 6 || d === diasMes) {
      console.log(fila.trimEnd());
      fila = "";
    }
  }

  const ordenados = [...festivos].sort((a, b) => a[0] - b[0]);
  if (ordenados.length === 0) {
    console.log("Festivos: ninguno");
  } else {
    console.log("Festivos:");
    ordenados.forEach(([d, nombre]) => console.log(` ${d} ${nombre} (${DIAS[diaSemana(d)]})`));
  }

  let laborables = 0;
  for (let d = 1; d <= diasMes; d++) if (diaSemana(d) < 5 && !festivos.has(d)) laborables++;
  console.log(`Días laborables: ${laborables}`);

  // Un festivo en martes deja «puente» el lunes anterior; uno en jueves, el viernes siguiente
  const puentes = new Set();
  for (const [d] of ordenados) {
    const candidato = diaSemana(d) === 1 ? d - 1 : diaSemana(d) === 3 ? d + 1 : 0;
    if (candidato >= 1 && candidato <= diasMes && !festivos.has(candidato)) puentes.add(candidato);
  }
  const lista = [...puentes].sort((a, b) => a - b).map((d) => `${DIAS[diaSemana(d)]} ${d}`);
  console.log(`Puentes posibles: ${lista.length ? lista.join(", ") : "ninguno"}`);
}

main();
```

Toda la aritmética de fechas la hace `Date` : el último día del mes con el truco del día 0 y el día de la semana con `getUTCDay` . Trabajar en UTC hace que el resultado sea el mismo en Madrid, en Canarias o en el servidor de Wandbox.

La conversión `(getUTCDay() + 6) % 7` reordena la semana para que empiece en lunes. Con ese único cambio, el hueco inicial de la primera semana y el salto de línea del domingo salen solos.

Cada celda mide siempre 3 caracteres (dos para el número y uno para la marca), así que las columnas quedan alineadas sin calcular posiciones. `trimEnd()` quita el relleno sobrante del final de cada línea.

El `Map` de festivos sirve a la vez para marcar el calendario, listarlos, descontarlos de los laborables y descartar puentes que ya son festivo: elegir bien la estructura de datos simplifica el resto del programa.

8. Reservas de pistas de pádel con clases

Difícil · JavaScript · 70 minutos · apuntesdam.com/ejercicios/javascript/reservas-de-pistas-de-padel

Solución (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const HORAS = ["09:00", "10:30", "12:00", "13:30", "15:00", "16:30", "18:00", "19:30", "21:00"];
const PISTAS = 3;

const euros = (centimos) => `${(centimos / 100).toFixed(2).replace(".", ",")} €`;

class Reserva {
  constructor(id, pista, hora, nombre, precio) {
    this.id = id;
    this.pista = pista;
    this.hora = hora;
    this.nombre = nombre;
    this.precio = precio;
  }
}

class Club {
  #reservas = new Map(); // id → Reserva
  #siguiente = 1;
  #socios = new Set();

  anadirSocios(nombres) {
    nombres.forEach((n) => this.#socios.add(n));
    return `Socios: ${this.#socios.size}`;
  }

  #lista() {
    return [...this.#reservas.values()];
  }

  reservar(pista, hora, nombre) {
    if (!Number.isInteger(pista) || pista < 1 || pista > PISTAS) return `Pista no válida: ${pista}`;
    if (!HORAS.includes(hora)) return `Hora no válida: ${hora}`;
    if (this.#lista().some((r) => r.pista === pista && r.hora === hora)) return `La pista ${pista} ya está reservada a las ${hora}`;
    if (this.#lista().filter((r) => r.nombre === nombre).length >= 2) return `${nombre} ya tiene 2 reservas`;

    // Hora valle hasta las 15:00 (12 €) y punta desde las 16:30 (18 €); los socios pagan un 30 % menos
    const base = HORAS.indexOf(hora) < 5 ? 1200 : 1800;
    const precio = this.#socios.has(nombre) ? (base * 70) / 100 : base;
    const reserva = new Reserva(this.#siguiente++, pista, hora, nombre, precio);
    this.#reservas.set(reserva.id, reserva);
    return `Reserva ${reserva.id}: pista ${pista} a las ${hora} para ${nombre} (${euros(precio)})`;
  }

  cancelar(id) {
    return this.#reservas.delete(id) ? `Reserva ${id} cancelada` : `No existe la reserva ${id}`;
  }

  libres(pista) {
    if (!Number.isInteger(pista) || pista < 1 || pista > PISTAS) return `Pista no válida: ${pista}`;
    const ocupadas = this.#lista().filter((r) => r.pista === pista).map((r) => r.hora);
    const libres = HORAS.filter((h) => !ocupadas.includes(h));
    return libres.length ? `Pista ${pista} libre a las ${libres.join(", ")}` : `Pista ${pista} sin horas libres`;
  }

  resumen() {
    const salida = [];
    for (let p = 1; p <= PISTAS; p++) {
      const n = this.#lista().filter((r) => r.pista === p).length;
      salida.push(`Pista ${p}: ${n} de ${HORAS.length} horas (${Math.round((n * 100) / HORAS.length)} %)`);
    }
  }
}
```

```

    salida.push(`Recaudación prevista: ${euros(this.#lista().reduce((suma, r) => suma + r.precio, 0))}`);

    const porCliente = this.#lista().reduce((mapa, r) => mapa.set(r.nombre, (mapa.get(r.nombre) ?? 0) +
    1), new Map());
    const [mejor] = [...porCliente].sort((a, b) => b[1] - a[1] || (a[0] < b[0] ? -1 : 1));
    salida.push(mejor ? `Mejor cliente: ${mejor[0]} (${mejor[1]} ${mejor[1] === 1 ? "reserva" :
    "reservas"})` : "Mejor cliente: ninguno");
    return salida.join("\n");
  }
}

const club = new Club();
for (const linea of lineas) {
  const texto = linea.trim();
  if (!texto) continue;
  const [orden, ...resto] = texto.split(/\s+/);
  switch (orden.toUpperCase()) {
    case "SOCIOS":
      console.log(club.anadirSocios(resto.join(" ").split(",").map((n) => n.trim()).filter(Boolean)));
      break;
    case "RESERVA":
      console.log(club.reservar(Number(resto[0]), resto[1] ?? "", resto.slice(2).join(" ")));
      break;
    case "CANCELA":
      console.log(club.cancelar(Number(resto[0])));
      break;
    case "LIBRES":
      console.log(club.libres(Number(resto[0])));
      break;
    case "RESUMEN":
      console.log(club.resumen());
      break;
    default:
      console.log(`Orden desconocida: ${orden}`);
  }
}
}

```

La clase `Club` encapsula las reglas del negocio: los campos privados impiden saltárselas y cada método es una operación con sentido para el club (reservar, cancelar, consultar). El programa principal solo traduce líneas de texto en llamadas.

Los métodos de arrays expresan cada pregunta en una línea: `some` (¿hay alguna?), `filter` + `length` (¿cuántas?), `map` (dame solo las horas), `reduce` (súmalas). Encadenarlos sobre la lista de reservas evita bucles con contadores y banderas.

Los identificadores se generan con un contador que nunca retrocede: si se reutilizaran los números de las reservas canceladas, un cliente con el justificante antiguo podría cancelar la reserva de otro.

El precio se calcula en céntimos y solo se formatea al escribirlo. `(base * 70) / 100` da 840 y 1260, cantidades exactas: no aparece ningún 8.399999 por el camino.

9. Validador de formularios de registro

Difícil · JavaScript · 65 minutos · apuntesdam.com/ejercicios/javascript/validador-de-formularios-de-registro

Solución (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const HOY = new Date(Date.UTC(2026, 8, 1)); // fecha de referencia: 1 de septiembre de 2026
const CAMPOS = ["nombre", "email", "password", "confirmacion", "nacimiento", "cp", "telefono"];

/** Edad cumplida en la fecha de referencia. */
function edad(nacimiento) {
  let anios = HOY.getUTCFullYear() - nacimiento.getUTCFullYear();
  const antesDelCumple = HOY.getUTCMonth() < nacimiento.getUTCMonth() ||
    (HOY.getUTCMonth() === nacimiento.getUTCMonth() && HOY.getUTCDate() < nacimiento.getUTCDate());
  return antesDelCumple ? anios - 1 : anios;
}

/** «2026-02-30» no es una fecha: Date la convertiría en el 2 de marzo, así que se comprueba que no
  cambie. */
function fecha(texto) {
  const m = texto.match(/^(\\d{4})-(\\d{2})-(\\d{2})$/);
  if (!m) return null;
  const [a, mes, d] = m.slice(1).map(Number);
  const f = new Date(Date.UTC(a, mes - 1, d));
  return f.getUTCFullYear() === a && f.getUTCMonth() === mes - 1 && f.getUTCDate() === d ? f : null;
}

/** Errores de un formulario: un objeto campo → mensaje, como mucho uno por campo. */
function validar(datos) {
  const v = Object.fromEntries(CAMPOS.map((c) => [c, datos[c] == null ? "" : String(datos[c]).trim()]));
  const errores = {};

  if (!v.nombre) errores.nombre = "obligatorio";
  else if (v.nombre.length < 2 || v.nombre.length > 40) errores.nombre = "entre 2 y 40 caracteres";
  else if (!/^\\p{L}[\\p{L} '-]*$/u.test(v.nombre)) errores.nombre = "solo letras, espacios, apóstrofes y guiones";

  if (!v.email) errores.email = "obligatorio";
  else if (!/^\\^\\s@+@\\^\\s@+\\. [a-z]{2,}$/i.test(v.email)) errores.email = "formato no válido";

  const nombrePropio = v.nombre.split(" ")[0].toLowerCase();
  if (!v.password) errores.password = "obligatorio";
  else if (v.password.length < 8) errores.password = "al menos 8 caracteres";
  else if (!/[A-Z]/.test(v.password) || !/[a-z]/.test(v.password) || !/\\d/.test(v.password))
    errores.password = "necesita mayúsculas, minúsculas y números";
  else if (nombrePropio.length >= 3 && v.password.toLowerCase().includes(nombrePropio)) errores.password = "no puede contener tu nombre";

  if (v.password && v.confirmacion !== v.password) errores.confirmacion = "no coincide con la contraseña";

  const nacimiento = fecha(v.nacimiento);
  if (!v.nacimiento) errores.nacimiento = "obligatorio";
  else if (!nacimiento) errores.nacimiento = "fecha no válida";
  else if (edad(nacimiento) < 16) errores.nacimiento = "hay que tener al menos 16 años";

  // Los dos primeros dígitos del código postal son la provincia: del 01 al 52
  if (!v.cp) errores.cp = "obligatorio";
  else if (!/^([01-9]|[1-4]\\d|5[0-2])\\d{3}$/i.test(v.cp)) errores.cp = "código postal no válido";

  if (v.telefono && !/^6789\\d{8}$/i.test(v.telefono.replace(/\\s/g, ""))) errores.telefono = "teléfono no válido";
  return errores;
}

let total = 0;
let correctos = 0;
const fallosPorCampo = new Map();
for (const linea of lineas) {
  if (!linea.trim()) continue;

```

```

total++;
let datos;
try {
  datos = JSON.parse(linea);
} catch {
  console.log(`Formulario ${total}: JSON no válido`);
  continue;
}
if (datos === null || typeof datos !== "object" || Array.isArray(datos)) {
  console.log(`Formulario ${total}: JSON no válido`);
  continue;
}
const errores = validar(datos);
const campos = CAMPOS.filter((c) => c in errores); // en el orden del formulario
if (campos.length === 0) {
  correctos++;
  console.log(`Formulario ${total}: correcto`);
} else {
  console.log(`Formulario ${total}: ${campos.length} ${campos.length === 1 ? "error" : "errores"}`);
  for (const c of campos) {
    console.log(` - ${c}: ${errores[c]}`);
    fallosPorCampo.set(c, (fallosPorCampo.get(c) ?? 0) + 1);
  }
}
}

console.log(`Correctos: ${correctos} de ${total}`);
const peor = CAMPOS.filter((c) => fallosPorCampo.has(c)).sort((a, b) => fallosPorCampo.get(b) - fallosPorCampo.get(a))[0];
console.log(`Campo con más errores: ${peor ? `${peor} (${fallosPorCampo.get(peor)})` : "ninguno"}`);

```

La función `validar` es pura: recibe datos y devuelve errores, sin escribir nada ni tocar el DOM. Por eso se puede usar igual en el navegador (al enviar el formulario) y en el servidor (al recibirlo), que es donde la validación es obligatoria de verdad.

Normalizar todos los campos al principio (texto recortado, vacío si falta) elimina de golpe los casos raros de `undefined`, `null` o números, y cada regla solo tiene que mirar cadenas.

Las expresiones regulares concentran el formato: `\p{L}` con el flag `u` acepta nombres reales de cualquier idioma, y la del código postal comprueba a la vez la longitud y el rango de provincias. Lo que no es formato (la fecha que existe, la edad, el nombre dentro de la contraseña) se resuelve con código normal.

`Date` corrige en silencio las fechas imposibles; comprobar que la fecha no ha cambiado al crearla es la forma fiable de validarlas. Usar UTC y una fecha de referencia fija hace que el resultado no dependa de la zona horaria ni del día de ejecución.

10. Cargador de datos asíncrono con reintentos y dependencias

Muy difícil · JavaScript · 90 minutos · apuntesdam.com/ejercicios/javascript/cargador-asincrono-con-reintentos

Solución (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const LIMITE_MS = 100;
const INTENTOS = 3;
let peticiones = 0;

/** Petición simulada: tarda «ms» y falla mientras el intento sea menor o igual que «fallos». */
function peticion(recurso, intento) {
  peticiones++;
  return new Promise((resolve, reject) => {
    setTimeout(() => (intento <= recurso.fallos ? reject(new Error("error del servidor")) :
      resolve(recurso.nombre)), recurso.ms);
  });
}

/** La primera promesa que termine gana: la petición o el temporizador. */
function conTiempoLimite(promesa, ms) {
  const limite = new Promise(_, reject) => setTimeout(() => reject(new Error("tiempo agotado")), ms);
  return Promise.race([promesa, limite]);
}

async function conReintentos(recurso) {
  let motivo = "";
  for (let intento = 1; intento <= INTENTOS; intento++) {
    try {
      await conTiempoLimite(peticion(recurso, intento), LIMITE_MS);
      return { estado: "cargado", intentos: intento };
    } catch (error) {
      motivo = error.message;
    }
  }
  return { estado: "error", motivo };
}

const recursos = new Map();
lineas.forEach((linea, i) => {
  const texto = linea.trim();
  if (!texto) return;
  const m = texto.match(/^(\\S+)\\s+(\\d+)\\s+(\\d+)(?:\\s+(\\S+))?$$/);
  if (!m || recursos.has(m[1])) {
    console.log(`Aviso: línea ${i + 1} ignorada`);
    return;
  }
  recursos.set(m[1], { nombre: m[1], ms: Number(m[2]), fallos: Number(m[3]), depende: m[4] ?
    m[4].split(",") : [] });
});

// Cada recurso se carga una sola vez: se guarda su promesa y quien dependa de él la espera
const cargas = new Map();
function cargar(nombre) {
  if (!cargas.has(nombre)) {
    cargas.set(nombre, (async () => {
      const recurso = recursos.get(nombre);
      const deps = await Promise.all(recurso.depende.map((d) => (recursos.has(d) ? cargar(d) : { estado:
        "no existe" })));
      const i = deps.findIndex((r) => r.estado !== "cargado");
      if (i >= 0) {
        const razon = deps[i].estado === "no existe" ? "que no existe" : "que no se ha podido cargar";
        return { estado: "omitido", motivo: `depende de ${recurso.depende[i]}, ${razon}` };
      }
      return conReintentos(recurso);
    }));
  }
  return cargas.get(nombre);
}
```

```

async function main() {
  // Todas las cargas empiezan a la vez; Promise.all espera a que terminen todas
  const resultados = await Promise.all([...recursos.keys()].map(cargar));
  [...recursos.keys()].forEach((nombre, i) => {
    const r = resultados[i];
    if (r.estado === "cargado") console.log(`${nombre}: cargado (${r.intentos} ${r.intentos === 1 ?
      "intento" : "intentos"})`);
    else if (r.estado === "error") console.log(`${nombre}: error tras ${INTENTOS} intentos
      (${r.motivo})`);
    else console.log(`${nombre}: omitido (${r.motivo})`);
  });
  const cargados = resultados.filter((r) => r.estado === "cargado").length;
  console.log(`Cargados: ${cargados} de ${resultados.length} · Peticiones: ${peticiones}`);
}

main();

```

Cada pieza resuelve un problema distinto de la asincronía: `Promise.race` pone un límite de tiempo, el bucle con `await` reintentará, `Promise.all` espera a varias cosas a la vez y el `Map` de promesas evita trabajo repetido. Combinadas dan un cargador robusto con muy poco código.

Guardar la promesa (no el resultado) es la clave de las dependencias: quien llegue antes de que termine la carga espera a la misma promesa, y quien llegue después recibe el resultado al instante. Es la técnica que usan las cachés de peticiones de las librerías de datos.

Los errores asíncronos se capturan con `try/catch` alrededor del `await`, igual que los síncronos. Los recursos que no se pueden cargar no rompen nada: se convierten en un resultado con estado `error` u `omitido`, y el resto del panel sigue funcionando.

El informe no depende de los tiempos: los resultados se muestran en el orden de la entrada porque `Promise.all` conserva el orden. Es la diferencia entre un programa que funciona «casi siempre» y uno que funciona siempre.