EJERCICIO 1.

Dispones de un array que representa un inventario, donde el índice es la "posición física" en el almacén y el valor es el *id* del producto. El array puede ser disperso (con huecos) y contener distintos valores (números, `null`, `undefined`, etc.).

Para comenzar, crea este array:

```
1. const inventario = [];
2. inventario[0] = 101;
3. inventario[2] = undefined;
4. inventario[5] = 104;
5. delete inventario[5];
6. inventario[7] = null;
7. inventario.extra = "metadato";
```

Implementa una función `condensarInventario(arr)` que devuelva un array denso de longitud `arr.length`, cumpliendo:

- Si el índice **no existe** en `arr` (hueco), en la salida pon `{ id: null, estado: 'hueco' }`.
- Si el índice **existe** y el valor es:
  - número ⇒ `{ id: valor, estado: 'ok' }`
  - `undefined` ⇒ `{ id: undefined, estado: 'indefinido' }`
  - `null` ⇒ `{ id: null, estado: 'nulo' }`
  - cualquier otro ⇒ `{ id: valor, estado: 'otro' }`

Para implementar esta función, no utilices `delete` (para "crear" huecos). Ten en cuenta que `forEach` ignoran huecos. También puedes usar el operador `in` para distinguir hueco de `undefined`.

Finalmente, comprueba cómo funciona cada bucle de la siguiente forma: sobre el inventario original y sobre el inventario denso de `condensarInventario(arr)`:

- forEach → muestra por consola cada par `(índice, valor)` visitado.
- for…of → muestra por consola los valores que produce (fíjate en los huecos).
- for…in → muestra por consola las claves que visita (fíjate si aparece extra).

EJERCICIO 2.

Implementa `compactarHuecos(arr)`, que elimine los huecos de un array preservando el orden de los elementos definidos. Además, debe preservar el valor `undefined` si es un valor explícito almacenado en una posición existente (no un hueco).

Implementa dos variantes de esta función:

1. *Destructiva*: compactación del mismo array original usando dos variables como punteros o iteradores. Si necesitas eliminar elementos al final del array, utiliza `length` pero evita `delete` para generar los huecos (no se recomienda su uso).
2. *No destructiva*: devuelve un nuevo array comprimido (el original no se modifica).

Puedes utilizar el siguiente array de entrada:

```
1. const a = [];
2. a[0] = 10;
3. a[2] = undefined;
4. a[5] = 30;
5. delete a[5];
6. a[7] = 40;
7. a.extra = "meta";
```

Pistas:

- Recuerda que `for...in` recorre claves presentes (y puede incluir propiedades que no sean índice), mientras que un bucle `for` por índices recorre `0..length-1`. Para este ejercicio, se recomienda que utilices índices numéricos.
- `delete` deja huecos, pero evita su uso. `splice` reindexa y ajusta `length`.
- `i in a` dice si el índice existe

Para probar tu código debes realizar, al menos, la siguiente prueba:

- Imprime `length` antes y después.
- Imprime los índices presentes con `for (let i=0; i<a.length; i++) console.log(i in a, a[i])`.
- Verifica que `a.extra` sigue existiendo como propiedad, pero no como elemento del array comprimido.

EJERCICIO 3.

Implementa las siguientes funciones:

1. `toJSONDenso(arr)` ⇒ devuelve una cadena JSON que representa fielmente un array posiblemente disperso, sin perder información de los huecos. Cuando haya un hueco en el array, deberás serializarlo como el objeto marcador `{"__gap__": true}`
2. `fromJSONDenso(json)` ⇒ hace el parseo inverso y reconstruye un array con los mismos huecos en los mismos índices.

Para la implementación de las funciones debes considerar:

- No utilices librerías externas. Puedes utilizar las funciones `JSON.stringify(…)` y `JSON.parse(…)`
- Debes preservar el orden y la propiedad `length`.
- Los valores definidos (números, null, undefined, strings, objetos simples) se serializan con JSON normal.
- Los huecos debes serializarlos en JSON con el marcador `{"__gap__": true}`
- Puedes ignorar cualquier propiedad no índice añadida al array.
- Usa un bucle por índices (`for`) y el operador `in` para inspeccionar huecos correctamente.

- JSON no tiene undefined explícito, por lo que puedes utilizar una clave-valor de la forma `{ "__undefined__": true }`

Como datos de entrada puedes utilizar:

```
1. const inv = [];
2. inv[0] = 100;
3. inv[2] = undefined;
4. inv[5] = 300;
5. delete inv[5];
6. inv[7] = null;
7. inv.extra = "metadato";
```

Recuerda que la función fromJSONDenso(json) debe realizar el proceso inverso hasta generar el array original:

- Debe restaurar los huecos cuando vea `{"__gap__": true}`
- Debe escribir `undefined` cuando vea `{"__undefined__": true}`
- Debe respetar `length` (incluyendo huecos al final si los hubiera).

EJERCICIO 4

Dispones de un array que contiene números, valores no numéricos, huecos, y `undefined` explícitos. El ejercicio consiste en calcular tres medidas estadísticas: media, mediana y moda. Para ello, ignora los huecos, pero cuenta los `undefined` explícitos como si fueran un valor más.

Utiliza los siguientes datos de entrada:

```
1. const datos = [];
2. datos[0] = 10;
3. datos[1] = undefined;
4. datos[3] = 20;
5. delete datos[3];
6. datos[5] = 30;
7. datos[6] = 10;
8. datos[9] = undefined;
9. datos.extra = "metadato que no debe contarse";
```

Implementa la función `calcularEstadísticas(arr)` que devuelva un objeto de la siguiente forma:

```
1. {
2.   media: <number>,
3.   mediana: <number>,
4.   moda: <valor o array de valores>,
5.   n_elementos: <cuántos valores se han contado>
6. }
```

EJERCICIO 5

Codifica un "generador de handlers" (*callbacks*) dentro de un `for` que demuestre el clásico *bug de cierre* con `var` y su corrección con `let`. Después, recapacita sobre lo que ocurre internamente.

Crea un array que simule tres "botones" de un documento cualquiera (mediante su *id*) y una función que "asocie" *handlers*:

```
1. const botones = [{id:'A'}, {id:'B'}, {id:'C'}];
2.
3. function registrar(elt, handler) {
4.   elt._onClick = handler; // Equivaldría a: .addEventListener('click', handler)
5. }
```

Caso 1. Escribe una función `generarHandlers_var()` que recorra los botones con un bucle for clásico, cuyos índices se inicialicen con `var`. Dentro del bucle, registra el *handler* que, al ejecutarse, muestre el índice `i` y el `id` del botón. Finalmente, ejecuta manualmente los tres *handlers* (`_onClick()`).

Para probar la función, podrías ejecutar el siguiente código:

```
1. generarHandlers_var(botones);
2. botones.forEach(b => b._onClick());
```

Deberías obtener una salida similar a:

```
1. var -> i: 3, id: undefined
2. var -> i: 3, id: undefined
3. var -> i: 3, id: undefined
```

Caso 2.

Repite el caso anterior, ahora como `generarHandlers_let()` usando `let i` en el `for`. ¿Cuál es la salida?

Reflexiona sobre lo que está ocurriendo en cada uno de los casos.

Caso 3.

Otra alternativa para evitar el error del bucle habitual en el uso de `var`, en caso de que no se pueda utilizar `let`, sería crear una factoría que capture el valor de `i` como parámetro, creando un nuevo entorno léxico por iteración. Es importante observar aquí cómo el valor de `i` se envuelve en forma de argumento de la función para "copiar" su valor (evitamos los efectos de `var`). Cada handler cierra sobre el parámetro local `idx`.

```
1. function makeHandler(idx, id) {
2.   return function handler() {
3.     console.log('factory(var) -> i:', idx, 'id:', id);
4.   };
5. }
6.
7. function generarHandlers_conFactoria(botones) {
8.   for (var i = 0; i < botones.length; i++) {
9.     registrar(botones[i], makeHandler(i, botones[i].id));
10.  }
11. }
```

Prueba este código y analiza cómo se ha resuelto el problema aun cuando se está utilizando `var`.

NOTA: En código moderno se prefiere el uso de `let`/`const`, dejando el uso de `var` a situaciones legadas (código ya existente).

EJERCICIO 6

El operador *rest* (...) puede utilizarse para **desestructuración** de objetos, valores por defecto, restos y omisiones. En este ejercicio, dispones de un objeto anidado que representa el resultado de una API de cursos.

Para realizarlo, te recomiendo la lectura de la siguiente página:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring

Debes extraer información con desestructuración de objetos y arrays usando:

- *Valores por defecto* cuando una propiedad/fila no exista.
- *Omisión de elementos* al desestructurar arrays.
- El *operador rest* (...) para "el resto" de elementos/propiedades.

Recuerda que los parámetros *rest* trabajan con arrays/objetos, mientras que el valor por defecto se aplica solo cuando el valor es `undefined`.

Partamos de este código:

```
const payload = {
  meta: { version: 3, locale: 'es-ES' },
  course: {
    id: 42,
    title: 'JS Básico',
    teachers: [{ id: 1, name: 'Ada' }, { id: 2, name: 'Alan' }, { id: 3, name: 'Grace' }],
    schedule: ['lun 10:00', 'mié 12:00', 'vie 09:00'],
    tags: ['js', 'web', 'intro'],
    // course.level podría venir undefined
  },
  // payload.audit no siempre viene
};
```

Consideremos la siguiente desestructuración básica con renombrado y valores por defecto:

1. Objeto *payload*
   - Extrae de *payload.course*:
     - *CourseId* desde *id*,
     - *name* desde *title*,
     - *level* con valor por defecto *'beginner'* si falta o es `undefined`.
2. Arrays: omitir y tomar el "resto"
   - De *schedule*, toma solo el primer horario como *primerSlot* y el resto en *otrosSlots*.
   - De *teachers*, extrae la primera persona como *lead* y omite la segunda posición usando una coma "vacía" para que la tercera vaya a *backup*. Comprueba qué pasa si el array tuviera menos elementos.
3. Objetos: resto de propiedades
   De *payload.meta*, toma *locale* y captura el resto en *metaRest*. Observa qué contiene *metaRest*.
4. Anidada + por defecto

Extrae tags como *primerTag* (el primero) y *otrosTags* (resto). Si tags no existiera (o fuese `undefined`), usa como valor por defecto el array ['general']. Ojo: el por defecto solo se dispara ante `undefined`.

¿Cómo lo hacemos? Aquí tienes algunas guías para completar el ejercicio.

- Para arrays: [head, , third] omite el segundo; [first, ...tail] captura el resto.
- Para objetos: { a, ...rest } recoge el resto de las propiedades propias en rest.
- El por defecto se evalúa si el valor extraído es `undefined` (no con `null`).

Finalmente, con este ejercicio (algo más complejo que los anteriores seguramente) habrás aprendido cómo desestructurar objetos, algo que resulta imprescindible para el tratamiento correcto de APIs más complejas y para lo que JS ofrece gran potencia.

Observa para el ejercicio:

- Valores por defecto. Se disparan solo cuando el valor desestructurado es `undefined` (no con `null`). Ej.: `let [a = 1] = [] ⇒ a=1`
- Omisión en arrays: comas vacías "saltan" posiciones: `[first, , third]`
- Rest de arrays: `[head, ...tail]` recoge lo restante en un nuevo array; útil para conservar orden.
- Rest de objetos: `{ a, ...rest }` agrupa el resto de las propiedades propias no extraídas.
- Recuerda que las copias son superficiales; referencias internas se comparten.

EJERCICIO 7

Implementa la función `sumaRobusta(lista)` que sume solo los elementos convertibles a número finito, ignorando el resto. Escribe una tabla de conversión que registre cómo decidiste convertir (o no) cada valor.

Puedes partir de la siguiente entrada:

```
const entrada = [1, "2", true, null, undefined, NaN, "tres"];
```

Sigue las siguientes reglas:

1. Solo se suman valores que, tras una conversión explícita, resulten en número finito (no `NaN`, no `Infinity`). Usa `Number(...)` y comprueba con `Number.isNaN` / `isFinite`
2. Registra un log de decisiones por cada elemento de la lista que sea de la forma:

```
{ original: <valor>, convertido: <número|null>, razon: "<por qué incluyes/excluyes>" }
```

3. Evita que el operador + haga concatenación de cadenas accidentalmente.
4. Considera estas conversiones conocidas del lenguaje (útiles para justificar):
    - Number("2") → 2 (OK), Number("tres") → NaN (excluir).
    - Number(true) → 1, Number(false) → 0 (OK).
    - Number(null) → 0 (OK), Number(undefined) → NaN (excluir).
    - NaN nunca es igual a sí mismo y debes detectarlo con Number.isNaN

La función debe devolver un objeto de la siguiente forma:

```
{
  suma: <number>,
  incluidos: <número de valores sumados>,
  excluidos: <número de valores descartados>,
  decisiones: <array de logs>
}
```

Para probar la función, puedes utilizar la ofrecida arriba, así como alguna más compleja:

```
[ "10", "  3.5 ", "-2e3", "", "42px", NaN, Infinity, false ]
```

EJERCICIO 8

Finalmente, vamos a probar cómo funciona la carga de scripts. Crea una página HTML con 4 `<script>` externos (dos con `async`, dos con `defer`) que registren en consola el orden real en que se ejecutan. Debes explicar por qué se observa ese orden y qué bloquea el render.

Paso 1. Desarrolla la estructura base en HTML.

Crea *index.html* con un contenido suficiente para ver el parseo (por ejemplo, varios párrafos y algunas imágnes). Añade cuatro scripts externos:

```
<!-- Dos async (A1, A2) -->
<script src="A1.async.js" async></script>
<script src="A2.async.js" async></script>

<!-- Dos defer (D1, D2) -->
<script src="D1.defer.js" defer></script>
<script src="D2.defer.js" defer></script>
```

Añade también *listeners* a `DOMContentLoaded` y `load` en el propio HTML (en el HEAD) para registrar el momento en el que ocurre.

```
<script>
  document.addEventListener('DOMContentLoaded', () =>
    console.log('[PAGE] DOMContentLoaded'));
  window.addEventListener('load', () =>
    console.log('[PAGE] load'));
</script>
```

Paso 2. Implementa los cuatro ficheros JS.

Cada fichero JS debe hacer lo siguiente:

- Registrar e, inicio y fin de la ejecución con un `console.log` identificativo (A1/A2/D1/D2).
- Simular una pequeña latencia con `setTimeout` o un bucle ligero para observar intercalados. https://developer.mozilla.org/es/docs/Web/API/Window/setTimeout
- Registrar el tiempo relativo (`performance.now()`) y el estado del DOM (`document.readyState`) para su análisis. https://developer.mozilla.org/en-US/docs/Web/API/Performance https://developer.mozilla.org/en-US/docs/Web/API/Document/readyState

Un ejemplo de lo anterior sería:

```
console.log('A1 start', performance.now().toFixed(1), document.readyState);
setTimeout(() => {
  console.log('A1 end', performance.now().toFixed(1), document.readyState);
}, 0);
```

`document.readyState` suele ser "*loading*" durante el parseo, "*interactive*" al completar el DOM (antes de `load`) y "*complete*" tras `load`. Podrás observar como los `async` pueden interrumpir el parseo al momento de ejecutarse; los `defer` se ejecutan después del parseo y antes de `DOMContentLoaded`.

Puedes jugar con este ejercicio para, por ejemplo, añadir un script clásico en el HEAD o al final del BODY. También puedes medir tiempos con y sin imágenes (o con carga *lazy* de imágenes) para ver el impacto.