## FUNCIONES

Hay varias formas de crear funciones en Javascript: por declaración (la más usada por principiantes), por expresión (la más habitual en programadores con experiencia) o mediante constructor de objeto (no recomendada):

***function nombre(p1, p2...) { }***función mediante declaración.

***var nombre = function(p1, p2...) { }***función expresión.

***new Function(p1, p2..., code);*** función constructor objeto.

### *Funciones por declaración*

Probablemente, la forma más popular de estas tres, y a la que estaremos acostumbrados si venimos de otros lenguajes de programación, es la primera, a la creación de funciones por declaración. Esta forma permite declarar una función que existirá a lo largo de todo el código:

```
function saludar() {
  return "Hola";
}
saludar(); // 'Hola'
typeof saludar; // 'function'
```

### *Funciones por expresión*

Sin embargo, en Javascript es muy habitual encontrarse códigos donde los programadores «guardan funciones» dentro de variables, para posteriormente «ejecutar dichas variables». Se trata de un enfoque diferente, creación de funciones por expresión, que fundamentalmente, hacen lo mismo con algunas diferencias:

```
// El segundo "saludar" (nombre de la función) se suele omitir: es redundante
const saludo = function saludar() {
  return "Hola";
};
saludo(); // 'Hola'
```

### *Funciones anónimas*

Las funciones anónimas o funciones lambda son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

```
// Función anónima "saludo"
const saludo = function () {
  return "Hola";
};
saludo; // ƒ () { return 'Hola'; }
saludo(); // 'Hola'
```

### *Callbacks*

Ahora que conocemos las funciones anónimas, podremos comprender más fácilmente como utilizar callbacks (también llamadas funciones callback o retrollamadas). A grandes rasgos, un callback (llamada hacia atrás) es pasar una función B por parámetro a una función A, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función:

```
// fB = Función B
const fB = function () {
  console.log("Función B ejecutada.");
};

// fA = Función A
const fA = function (callback) {
  callback();
};

fA(fB);
```

### *Funciones autoejecutables*

Pueden existir casos en los que necesites crear una función y ejecutarla sobre la marcha. En Javascript es muy sencillo crear funciones autoejecutables. Básicamente, sólo tenemos que envolver entre paréntesis la función anónima en cuestión (no necesitamos que tenga nombre, puesto que no la vamos a guardar) y luego, ejecutarla:

```
// Función autoejecutable
(function () {
  console.log("Hola!!");
})();

// Función autoejecutable con parámetros
(function (name) {
  console.log(`¡Hola, ${name}!`);
})("Manz");
```

### *Clausuras*

Las clausuras o cierres, es un concepto relacionado con las funciones y los ámbitos que suele costar comprender cuando se empieza en Javascript. Es importante tener las bases de funciones claras hasta este punto, lo que permitirá entender las bases de una clausura.

A grandes rasgos, en Javascript, una clausura o cierre se define como una función que «encierra» variables en su propio ámbito (y que continúan existiendo aún habiendo terminado la función). Por ejemplo, veamos el siguiente ejemplo:

```
// Clausura: Función incr()
const incr = (function () {
  let num = 0;
  return function () {
    num++;
    return num;
  };
})();

typeof incr; // 'function'
incr(); // 1
incr(); // 2
incr(); // 3
```

**Ámbito léxico de this**

Aunque aún no la hemos utilizado, una de las principales diferencias de las funciones flecha respecto a las funciones tradicionales, es el valor de la palabra clave this, que no siempre es la misma.

```
// Si son funciones globales
const a = function () {
  console.log(this);
};
const b = () => {
  console.log(this);
};
a(); // Window
b(); // Window
```

### *Arrow functions*

Las Arrow functions, funciones flecha o «fat arrow» son una forma corta de escribir funciones que aparece en Javascript a partir de ECMAScript 6. Básicamente, se trata de reemplazar eliminar la palabra function y añadir => antes de abrir las llaves:

```
const func = function () {
  return "Función tradicional.";
};
const func = () => {
  return "Función flecha.";
};
const func = () => "Funcion flecha en linea";
```

## OBJETOS

En Javascript, existe un tipo de dato llamado objeto . Una primera forma de verlo, es como una variable especial que puede contener más variables en su interior. De esta forma, tenemos la posibilidad de organizar múltiples variables de la misma temática en el interior de un objeto.

*const objeto = new Object(); // Evitar esta sintaxis en Javascript (no se suele usar)*

***Declaración de un objeto***

```
const player = {
  name: "Manz",
  life: 99,
  power: 10,
};
```

***Propiedades de un objeto***

```
// Notación con puntos (preferida)
console.log(player.name);     // Muestra "Manz"
console.log(player.life);     // Muestra 99
// Notación con corchetes
console.log(player["name"]);  // Muestra "Manz"
console.log(player["life"]);  // Muestra 99
```

***Añadir propiedades***

```
// FORMA 1: A través de notación con puntos
const player = {};
player.name = "Manz";
player.life = 99;
player.power = 10;

// FORMA 2: A través de notación con corchetes
const player = {};
player["name"] = "Manz";
player["life"] = 99;
player["power"] = 10;
```

***Métodos de un objeto***

Si dentro de una variable del objeto metemos una función (o una variable que contiene una función), tendríamos lo que se denomina un método de un objeto:

```
const user = {
  name: "Manz",
  talk: function() { return "Hola"; }
};
user.name;       // Es una variable (propiedad), devuelve "Manz"
user.talk();     // Es una función (método), se ejecuta y devuelve "Hola"
```

***El método toString***

```
const number = 42.5;
number.toString();          // Devuelve "42.5" (Método de variables de tipo Object)
number.toLocaleString();    // Devuelve "42,5" (Método de variables de tipo Object)
number.toFixed(3);          // Devuelve "42.500" (Método de variables de tipo Number)
```

***¿Qué es JSON?***

JSON son las siglas de JavaScript Object Notation, y no es más que un formato ligero de datos, con una estructura (notación) específica, que es totalmente compatible de forma nativa con Javascript. Como su propio nombre indica, JSON se basa en la sintaxis que tiene Javascript para crear objetos.

```
{
  "name": "Manz",
  "life": 3,
  "totalLife": 6
  "power": 10,
  "dead": false,
  "props": ["invisibility", "coding", "happymood"],
  "senses": {
    "vision": 50,
    "audition": 75,
    "taste": 40,
    "touch": 80
  }
}
```

***Convertir JSON a objeto***

```
const json = `{
  "name": "Manz",
  "life": 99
}`;

const user = JSON.parse(json);
user.name;  // "Manz"
user.life;  // 99
```

***Convertir objeto a JSON***

```
const user = {
  name: "Manz",
  life: 99,
  talk: function () {
    return "Hola!";
  },
};

JSON.stringify(user);       // '{"name":"Manz","life":99}'
```

las funciones no están soportadas por JSON, por lo que si intentamos convertir un objeto que contiene métodos o funciones, JSON.stringify() no fallará, pero simplemente devolverá un string omitiendo las propiedades que contengan funciones. (fetch() lee un JSON externo)

***Desestructuración de objetos***

Utilizando la desestructuración de objetos podemos separar en variables las propiedades que teníamos en el objeto:

```
const user = {
  name: "Manz",
  role: "streamer",
  life: 99
}

const { name, role, life } = user;

console.log(name);
console.log(role, life);
```

***Reestructuración de objetos***

sta característica de desestructuración podemos aprovecharla a nuestro favor, para reutilizar objetos y recrear nuevos objetos a partir de otros

```
const user = {
  name: "Manz",
  role: "streamer",
  life: 99
}

const fullUser = {
  ...user,
  power: 25,
  life: 50
}
```

## ¿Qué es el DOM?

Las siglas DOM significan Document Object Model, o lo que es lo mismo, la estructura del documento HTML. Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol DOM (o simplemente DOM).

***getElementById()***

El primer método, .getElementById(id) busca un elemento HTML con el id especificado en

```
const page = document.getElementById("page");
// <div id="page"></div>
```

***getElementsByClassName()***

Por otro lado, el método .getElementsByClassName(class) permite buscar los elementos con la clase especificada

```
const items = document.getElementsByClassName("item");
// [div, div, div]
console.log(items[0]);
// Primer item encontrado: <div class="item"></div>
console.log(items.length);  // 3
```

Exactamente igual funcionan los métodos ***getElementsByName(name),getElementsByTagName(tag)*** salvo que se encargan de buscar elementos HTML por su atributo name o por su propia etiqueta de elemento HTML, respectivamente:

```
// Obtiene todos los elementos con atributo
name="nickname"
const nicknames =
document.getElementsByName("nickname");

// Obtiene todos los elementos <div> de la página
const divs = document.getElementsByTagName("div");
```

***querySelector()***

Devuelve el primer elemento que encuentra que encaja con el selector CSS suministrado. Al igual que su «equivalente» *.getElementById()*, devuelve un solo elemento y en caso de no coincidir con ninguno, devuelve null

```
const page = document.querySelector("#page");
// <div id="page"></div>
const info = document.querySelector(".main .info");
// <div class="info"></div>
```

***querySelectorAll()***

Por otro lado, el método .querySelectorAll() realiza una búsqueda de elementos como lo hace el anterior, sólo que como podremos intuir por ese All(), devuelve un array con todos los elementos que coinciden con el selector CSS:

```
// Obtiene todos los elementos con clase "info"
const infos = document.querySelectorAll(".info");
// Obtiene todos los elementos con atributo
name="nickname"
const nicknames =
document.querySelectorAll('[name="nickname"]');
// Obtiene todos los elementos <div> de la página HTML
const divs = document.querySelectorAll("div");
```

***El método createElement()***

Mediante el método .createElement() podemos crear un elemento HTML en memoria  Con dicho elemento almacenado en una variable, podremos modificar sus características o contenido, para posteriormente insertarlo en una posición  del DOM o documento HTML.

```
// Creamos un <div></div>
const div = document.createElement("div");
```

***Reemplazar contenido***

La propiedad ***.textContent*** nos devuelve el contenido de texto de un elemento HTML. Es útil para obtener (o modificar) sólo el texto dentro de un elemento, obviando el etiquetado HTML

```
const div = document.querySelector("div");  // <div></div>
div.textContent = "Hola a todos";
// <div>Hola a todos</div>
div.textContent;  // "Hola a todos"
```

Por otro lado, la propiedad ***.innerHTML*** nos permite hacer lo mismo, pero interpretando el código HTML indicado y renderizando sus elementos:

```
const div = document.querySelector(".info");
// <div class="info"></div>
div.innerHTML = "<strong>Importante</strong>";
// Interpreta el HTML
div.innerHTML;    // "<strong>Importante</strong>"
div.textContent;  // "Importante"
div.textContent = "<strong>Importante</strong>";  // No
interpreta el HTML
```

La diferencia principal entre .innerHTML y .textContent es que el primero renderiza e interpreta el marcado HTML, mientras que el segundo lo inserta como contenido de texto literalmente.

***Insertar elementos***

***appendChild()*** inserta el elemento como un hijo al final de todos los elementos hijos que existan.

```
const img = document.createElement("img");
img.src = "https://lenguajejs.com/assets/logo.svg";
img.alt = "Logo Javascript";
document.body.appendChild(img);
```

Los métodos de la familia **insertAdjacent s**on bastante más versátiles que .appendChild(), ya que permiten muchas más posibilidades. Tenemos tres versiones diferentes:

***.insertAdjacentElement()*** donde insertamos un objeto

***.insertAdjacentHTML()*** donde insertamos código HTML directamente (similar a innerHTML)

***.insertAdjacentText()*** donde no insertamos elementos HTML, sino un  con texto

***Eliminar elementos***

```
const div =
document.querySelector(
".deleteme");


div.isConnected;    // true
div.remove();
div.isConnected;    //
false
```

```
const div =
document.querySelector(
".item:nth-child(2)");   //
<div
class="item">2</div>

document.body.removeC
hild(div); // Desconecta
el segundo .item
```

```
const div =
document.querySelector(
".item:nth-child(2)");
const newnode =
document.createElement(
"div");
newnode.textContent =
"DOS";
document.body.replaceC
hild(newnode, div);
```

***La propiedad className***

La propiedad .className viene a ser la modalidad directa y rápida de utilizar el getter .getAttribute("class") y el setter .setAttribute("class", v).

```
const div = document.querySelector(".element");
// Obtener clases CSS
div.className;              // "element shine dark-theme"
div.getAttribute("class");  // "element shine dark-theme"
// Modificar clases CSS
div.className = "elemento brillo tema-oscuro";
div.setAttribute("class", "elemento brillo tema-oscuro");
```

***El objeto classList***

Para trabajar más cómodamente, existe un sistema muy interesante para trabajar con clases: el objeto ***classList***. Se trata de un objeto especial que contiene una serie de ayudantes que permiten trabajar con las clases de forma más intuitiva y lógica.

El siguiente elemento HTML en nuestro documento. Vamos a acceder a el y a utilizar el objeto .classList con dicho elemento:

```
<div id="page" class="info data dark" data-number="5"></div>
```

***Acceder a clases CSS***

Al margen de acceder a la lista de clases mediante .classList y al número de clases del elemento con .classList.length, es posible acceder a la propiedad .classList.values para obtener un  como lo haría ***.className***:

```
const div = document.querySelector("#page");
div.classList;              // ["info", "data", "dark"] (DOMTokenList)
div.classList.length;       // 3
Array.from(div.classList)   // ["info", "data", "dark"] (Array)
[...div.classList];         // ["info", "data", "dark"] (Array)
div.classList.values;       // "info data dark" (String)
div.classList.item(0);      // "info"
div.classList.item(1);      // "data"
div.classList.item(3);      // null
```

***Añadir y eliminar clases CSS***

Los métodos ***.classList.add()*** y ***.classList.remove()*** permiten indicar una o múltiples clases CSS a añadir o eliminar.

```
const div = document.querySelector("#page");
div.classList.add("uno", "dos");    // No devuelve nada.
div.classList;  // ["info", "data", "dark", "uno", "dos"]
div.classList.remove("uno", "dos"); // No devuelve nada.
div.classList;  // ["info", "data", "dark"]
```

***Comprobar si existen clases CSS***

Con el método .classList.contains() podemos comprobar si existe una clase en un elemento HTML

```
const div = document.querySelector("#page");
div.classList;                      // ["info", "data", "dark"]
div.classList.contains("info");     // Devuelve `true` (existe esa clase)
div.classList.contains("warning");  // Devuelve `false` (no existe esa clase)
```

***Conmutar o alternar clases CSS***

Otro ayudante muy interesante es el del método ***.classList.toggle()***, que lo que hace es añadir o eliminar la clase CSS dependiendo de si ya existía previamente. Es decir, añade la clase si no existía previamente o elimina la clase si existía previamente:

```
const div = document.querySelector("#page");
div.classList; // ["info", "data", "dark"]
div.classList.toggle("info"); // Como "info" existe, lo elimina. Devuelve "false"
div.classList; // ["data", "dark"]
div.classList.toggle("info"); // Como "info" no existe, lo añade. Devuelve "true"
div.classList; // ["info", "data", "dark"]
```

***Reemplazar una clase CSS***

Por último, tenemos un método ***.classList.replace()*** que nos permite reemplazar la primera clase indicada por parámetro, por la segunda. Veamos este método en acción:

```
const div = document.querySelector("#page");
div.classList; // ["info", "data", "dark"]
div.classList.replace("dark", "light");       // Devuelve `true` (se hizo el cambio)
div.classList.replace("warning", "error");    // Devuelve `false` (no existe la clase warning)
```

***Navegar a través de elementos***

Las propiedades que veremos a continuación devuelven  información de otros elementos relacionados con el  elemento en cuestión.

| Propiedades de elementos HTML | Descripción |
|---|---|
| ARRAY children | Devuelve una lista de elementos HTML hijos. |
| ELEMENT parentElement | Devuelve el padre del elemento o NULL si no tiene. |
| ELEMENT firstElementChild | Devuelve el primer elemento hijo. |
| ELEMENT lastElementChild | Devuelve el último elemento hijo. |
| ELEMENT previousElementSibling | Devuelve el elemento hermano anterior o NULL si no tiene. |
| ELEMENT nextElementSibling | Devuelve el elemento hermano siguiente o NULL si no tiene. |

***Navegar a través de nodos***

La primera tabla que hemos visto nos muestra una serie de propiedades cuando trabajamos con . Sin embargo, si queremos hilar más fino y trabajar a nivel de , podemos utilizar las siguientes propiedades, que son equivalentes a las anteriores:

| Propiedades de nodos HTML | Descripción |
|---|---|
| ARRAY childNodes | Devuelve una lista de nodos hijos. Incluye nodos de texto y comentarios. |
| NODE parentNode | Devuelve el nodo padre del nodo o NULL si no tiene. |
| NODE firstChild | Devuelve el primer nodo hijo. |
| NODE lastChild | Devuelve el último nodo hijo. |
| NODE previousSibling | Devuelve el nodo hermano anterior o NULL si no tiene. |
| NODE nextSibling | Devuelve el nodo hermano siguiente o NULL si no tiene. |

## ¿Qué es un evento?

En Javascript existe un concepto llamado evento, que no es más que una notificación de que alguna característica interesante acaba de ocurrir, generalmente relacionada con el usuario que navega por la página.
Dichas características pueden ser muy variadas:

- Click del usuario sobre un elemento de la página
- Pulsación de una tecla específica del teclado
- Reproducción de un archivo de audio/video
- Scroll de ratón sobre un elemento de la página
- El usuario ha activado la opción Imprimir

***Eventos en HTML***

```
<button onClick="alert('Hello!')">Saludar</button>
```

***Eventos en JS***

```
<button>Saludar</button>

<script>
const button = document.querySelector("button");
button.onclick = function() {
  alert("Hello!");
}
</script>
```

***Utilizando setAttribute()***

```
<button>Saludar</button>

<script>
const button = document.querySelector("button");
const doTask = () => alert("Hello!");
button.setAttribute("onclick", "doTask()");
</script>
```

***Método .addEventListener()***

Con el método .addEventListener() permite añadir una escucha del evento indicado

```
const button = document.querySelector("button");
button.addEventListener("click", function() {
  alert("Hello!");
});
```

***Múltiples listeners***

Dicho método .addEventListener() permite asociar múltiples funciones a un mismo evento, algo que, aunque no es imposible, es menos sencillo e intuitivo en las modalidades de gestionar eventos que vimos anteriormente:

```
<button>Saludar</button>
<style>
  .red { background: red }
</style>
<script>
const button = document.querySelector("button");
const action = () => alert("Hello!");
const toggle = () => button.classList.toggle("red");

button.addEventListener("click", action);
// Hello message
button.addEventListener("click", toggle);
 // Add/remove red CSS
</script>
```

***Método .removeEventListener()***

El ejemplo anterior, se puede completar haciendo uso del método .removeEventListener(), que sirve como su propio nombre indica para eliminar un listener que se ha añadido

```
<button>Saludar</button>
<style>
  .red { background: red }
</style>
<script>
const button = document.querySelector("button");
const action = () => alert("Hello!");
const toggle = () => button.classList.toggle("red");
button.addEventListener("click", action);
// Add listener
button.addEventListener("click", toggle);
// Toggle red CSS
button.removeEventListener("click", action);
// Delete listener
</script>
```

***El objeto event***

```
const button = document.querySelector("button");
button.addEventListener("click", (event) =>
console.log(event));
```

***Ejemplo (con PointerEvent)***

Vamos a realizar un ejemplo con el evento click. Nuestro código nos permitirá hacer click con el ratón en cualquier parte de la pantalla. Nos mostrará los siguientes datos:

```
<span></span>

<style>
  body {
    margin: 0;
    width: 100vw;
    height: 100vh;
    background: lightgrey;
    user-select: none;
    font-size: 2rem;
  }
</style>

<script>
const span = document.body.querySelector("span");
const action = (event) => {
  const { x, y, detail } = event;
  span.textContent = `Has hecho ${detail} CLICK en las
coordenadas (${x}x${y})`;
};
document.body.addEventListener("click", action);
</script>
```

## ¿Qué es un array?

Un array es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. En algunas ocasiones también se les suelen llamar arreglos o vectores. En Javascript, se pueden definir de varias formas:

| Constructor | Descripción |
|---|---|
| ARRAY `new Array(` NUMBER `size)` | Crea un array vacío de tamaño **size**. Sus valores no están definidos, pero son UNDEFINED. |
| ARRAY `new Array(e1, e2...)` | Crea un array con los elementos indicados. |
| ARRAY `[e1, e2...]` | Simplemente, los elementos dentro de corchetes: **[]**. **Notación preferida**. |

```
// Forma tradicional (no se suele usar en Javascript)
const letters = new Array("a", "b", "c");
// Array con 3 elementos
const letters = new Array(3); // Array vacío de tamaño 3

// Mediante literales (notación preferida)
const letters = ["a", "b", "c"];  // Array con 3 elementos
const letters = [];               // Array vacío (0 elementos)
const letters = ["a", 5, true];   // Array mixto
```

***Acceso a elementos del array***

Al igual que los , saber el número elementos que tiene un array es muy sencillo. Sólo hay que acceder a la propiedad .length, que nos devolverá el número de elementos existentes en un array:

| Forma | Descripción |
|---|---|
| NUMBER `.length` | Propiedad que devuelve el número de elementos del array. |
| OBJECT `[pos]` | Operador que devuelve (o modifica) el elemento número **pos** del array. |
| OBJECT `.at(pos)` ES2022 | Método que devuelve el elemento en la posición **pos**. Números negativos en orden inverso. |

***Acceder***

```
const letters = ["a", "b", "c"];
letters.length;   // 3
letters[0];       // 'a'
letters[2];       // 'c'
letters[5];       // undefined
```

***Modificar***

```
const letters =  ["a", "b", "c"];
letters[1] = "Z";       // Devuelve "Z" y modifica letters a ["a", "Z", "c"]
letters[3] = "D";       // Devuelve "D" y modifica letters a ["a", "Z", "c", "D"]
letters[5] = "A";       // Devuelve "A" y modifica letters a ["a", "Z", "c", "D", undefined, "A"]
```

***Añadir o eliminar elementos***

Existen varias formas de añadir elementos a un array ya existente

| Método | Descripción |
|---|---|
| NUMBER `.push(e1, e2, e3...)` | Añade uno o varios elementos al final del array. Devuelve el tamaño del array. |
| OBJECT `.pop()` | Elimina el último elemento del array. Devuelve dicho elemento. |
| NUMBER `.unshift(e1, e2, e3...)` | Añade uno o varios elementos al inicio del array. Devuelve el tamaño del array. |
| OBJECT `.shift()` | Elimina el primer elemento del array. Devuelve dicho elemento. |

En los arrays, Javascript proporciona métodos tanto para insertar o eliminar elementos por el final del array, como por el principio:

Los métodos **.push**() y **.pop**() insertan al final del array.

Los métodos **.unshift**() y **.shift**() insertan al inicio del array.

```
const elements = ["a", "b", "c"]; // Array inicial

elements.push("d");
// Devuelve 4.   Ahora elements = ['a', 'b', 'c', 'd']
elements.pop();
// Devuelve 'd'. Ahora elements = ['a', 'b', 'c']

elements.unshift("Z");
// Devuelve 4.   Ahora elements = ['Z', 'a', 'b', 'c']
elements.shift();
// Devuelve 'Z'. Ahora elements = ['a', 'b', 'c']
```

***Concatenar arrays***

```
const elements = [1, 2, 3];
elements.push(4, 5, 6);
 // Devuelve 6. Ahora elements = [1, 2, 3, 4, 5, 6]
elements.push([7, 8, 9]);
// Devuelve 7. Ahora elements = [1, 2, 3, 4, 5, 6, [7, 8, 9]]
```

***Usando concat***

```
const firstPart = [1, 2, 3];
const secondPart = [4, 5, 6];
firstPart.concat(firstPart);       // Devuelve [1, 2, 3, 1, 2, 3]
firstPart.concat(secondPart);    // Devuelve [1, 2, 3, 4, 5, 6]
// Se pueden pasar elementos sueltos
firstPart.concat(4, 5, 6);         // Devuelve [1, 2, 3, 4, 5, 6]
// Se pueden concatenar múltiples arrays e incluso mezclarlos con elementos sueltos
firstPart.concat(firstPart, secondPart, 7);
// Devuelve [1, 2, 3, 1, 2, 3, 4, 5, 6, 7]
```

***Separar y unir strings***

```
const letters = ["a", "b", "c"];
// Une elementos del array por el separador indicado
letters.join("->");       // Devuelve 'a->b->c'
letters.join(".");        // Devuelve 'a.b.c'
// Separa elementos del string por el separador indicado
"a.b.c".split(".");       // Devuelve ['a', 'b', 'c']
"5-4-3-2-1".split("-");   // Devuelve ['5', '4', '3', '2', '1']
```

Ten en cuenta que, como se puede ver en los ejemplos, **.join**() siempre devolverá los elementos como string, mientras que **.split**() devolverá un array.

Observa un caso especial, en el que pasamos un cadena de texto string vacía al **.split**():

```
"Hola a todos".split("");   // ['H', 'o', 'l', 'a', ' ', 'a', ' ', 't', 'o', 'd', 'o', 's']
```

En este caso, le hemos pedido dividir el string sin indicar ningún separador, por lo que Javascript toma la unidad mínima como separador: nos devuelve un array con cada carácter del string original. Ten en cuenta que los espacios en blanco también cuentan como carácter.

***Buscar elementos en un array***

***¿El elemento existe? (includes)***

El método .**includes**() comprueba si el elemento indicado está incluido en el array. Es posible indicar un segundo parámetro donde indicaremos la posición number from desde donde empezamos a buscar:

```
const numbers = [5, 10, 15, 20, 25];
numbers.includes(3);    // false
numbers.includes(15);   // true
numbers.includes(15, 1);  // true
numbers.includes(15, 4);  // false
```

***Buscar la posición (indexOf)***

El método .**indexOf**() (el índice de..., la posición de...) hace algo similar a .includes(), pero en lugar de devolver un boolean, devuelve la posición del elemento buscado. De no encontrarlo, devuelve un valor negativo -1.

```
const numbers = [5, 10, 15, 20, 25, 20, 15, 10, 5];
numbers.indexOf(5);   // 0
numbers.indexOf(15);  // 2
numbers.indexOf(25);  // 4
numbers.indexOf(99);  // -1
numbers.indexOf(15, 1); // 2
numbers.indexOf(15, 4); // 6
numbers.indexOf(15, 7); // -1
```

***Buscar al final (lastIndexOf)***

Por su parte, el método .**lastIndexOf**() hace exactamente lo mismo que .**indexOf**(), con la única diferencia que empezará a buscar desde el final, en lugar desde el principio del array.

```
const numbers = [5, 10, 15, 20, 25, 20, 15, 10, 5];
numbers.lastIndexOf(5);   // 8
numbers.lastIndexOf(15);  // 6
numbers.lastIndexOf(25);  // 4
numbers.lastIndexOf(99);  // -1
numbers.lastIndexOf(15, 8); // 6
numbers.lastIndexOf(15, 5); // 2
numbers.lastIndexOf(15, 1); // -1
```

***Buscar un elemento en un array***

Ahora que conocemos los métodos anteriores, podemos aprovecharlos para buscar un elemento en un array. Cuando se trata de estructuras muy simples, no tiene sentido buscar el mismo elemento que ya conocemos y tenemos previamente, pero en estructuras más complejas si puede ser interesante.

Observa el siguiente ejemplo donde partimos de una estructura array de object, que tienen el nombre y la edad de una persona. Vamos a implementar una función **findElement**() para buscar la primera persona con una cierta edad:

```
const names = [
  { name: "María", age: 20 },
  { name: "Bernardo", age: 28 },
  { name: "Pancracio", age: 22 },
  { name: "Andrea", age: 19 },
  { name: "Sara", age: 29 },
  { name: "Jorge", age: 32 },
  { name: "Yurena", age: 38 },
  { name: "Ayoze", age: 18 }
];
// Busca el primer elemento con la edad indicada, sino devuelve -1
const findElement = (array, searchedAge) => {
  for (let i = 0; i < array.length; i++) {
    const element = array[i];
    if (element.age === searchedAge) {
      return element;
    }
  }
  return -1;
}
findElement(names, 19);     // { name: "Andrea", age: 19 }
findElement(names, 32);     // { name: "Jorge", age: 32 }
findElement(names, 33);     // -1
```

***Modificar o crear sub arrays***

***Crear fragmento (slice)***

El método .**slice**() devuelve los elementos del array desde la posición start hasta la posición end, permitiendo crear un nuevo array más pequeño con ese grupo de elementos. Recuerda que las posiciones empiezan a contar desde 0. En el caso de que no se proporcione el parámetro end, se devuelven todos los elementos desde la posición start hasta el final del array.

```
  const letters = ["a", "b", "c", "d", "e"];
  letters.slice(3);        // ["d", "e"]
  letters.slice(0, 2);     // ["a", "b"]
  letters.slice(4, 5);     // ["e"]
  letters.slice(2, 5);     // ["c", "d", "e"]
  letters.slice(-2);       // ["d", "e"]
```

**Alterar fragmento (splice)**

Por otro lado, el método .**splice**() realiza algo parecido a .**slice**() pero con una gran diferencia: modifica el array original. Además, en el método .**splice**() el segundo parámetro size no es la posición final del subarray (como en el caso anterior), sino el tamaño del array final, es decir, el número de elementos que se van a obtener desde la posición start.

```
  const letters = ["a", "b", "c", "d", "e"];
  letters.splice(0, 2);      // Devuelve ["a", "b"]
  letters                    // ["c", "d", "e"]
  const letters = ["a", "b", "c", "d", "e"];
  letters.splice(2, 1);      // Devuelve ["c"]
  letters                    // ["a", "b", "d", "e"]
```

***Diferencias entre slice y splice***

Veamos un ejemplo ilustrativo para entender bien la diferencia entre ambos:

```
const letters = ["a", "b", "c", "d", "e"];
// .slice() no modifica el array
letters.slice(2, 4);          // Devuelve ['c', 'd']. El array no se modifica.
// .splice() si modifica el array
letters.splice(2, 2);         // Devuelve ['c', 'd']. Ahora array = ['a', 'b', 'e']
letters.splice(1, 0, "z", "x");  // Devuelve []. Ahora array = ['a', 'z', 'x', 'b', 'e']
```

***Ordenar un array***

***El método reverse***

En primer lugar, el método **reverse**() reordena los elementos del array en orden inverso, es decir, si tenemos [5, 4, 3] lo modifica de modo que obtenemos [3, 4, 5]. Veamos su funcionamiento con el ejemplo anterior, y una serie de matices que conviene aclarar:

```
const elements = ["A", "B", "C", "D", "E", "F"];
const reversedElements = elements.reverse();
reversedElements          // ["F", "E", "D", "C", "B", "A"]
elements                  // ["F", "E", "D", "C", "B", "A"]
reversedElements === elements    // true
```

Si queremos crear un nuevo array independiente del original, tendríamos que hacer lo siguiente:

```
const elements = ["A", "B", "C", "D", "E", "F"];
const reversedElements =
structuredClone(elements).reverse();
reversedElements          // ["F", "E", "D", "C", "B", "A"]
elements                  // ['A', 'B', 'C', 'D', 'E', 'F']
reversedElements === elements    // false
```

**El método sort**

Cualquier estructura de datos array en Javascript tiene el método **sort**(). Este método realiza una ordenación de los elementos del array, con la peculiaridad que siempre realizará una ordenación alfabética:

```
const names = ["Alberto", "Zoe", "Ana", "Mauricio",
"Bernardo"];
const sortedNames = names.sort();
sortedNames   // ["Alberto", "Ana", "Bernardo",
"Mauricio", "Zoe"]
names   // ["Alberto", "Ana", "Bernardo", "Mauricio",
"Zoe"]
sortedNames === names     // true
```

***¿Qué son las Array functions?***

Básicamente, son métodos que tiene cualquier variable que sea de tipo array, y que permite realizar una operación con todos los elementos de dicho array (o parte de ellos) para conseguir un objetivo concreto, dependiendo del método. En general, a dichos métodos se les pasa por parámetro una función callback y unos parámetros opcionales.

***Bucle***

***Por cada uno (forEach)***

Como se puede ver, el método forEach() no devuelve nada y espera que se le pase por parámetro una function que se ejecutará por cada elemento del array. Esa función, puede ser pasada en cualquiera de los formatos que hemos visto: como función tradicional o como función flecha:

```
const letters = ["a", "b", "c", "d"];
// Con funciones por expresión
const f = function () {
  console.log("Un elemento.");
};
letters.forEach(f);
// Con funciones anónimas
letters.forEach(function () {
  console.log("Un elemento.");
});
// Con funciones flecha
letters.forEach(() => console.log("Un elemento."));
```

Sin embargo, este ejemplo no tiene demasiada utilidad. A la callback se le pueden pasar varios parámetros opcionales:
Si se le pasa un primer parámetro, este será el elemento del array.
Si se le pasa un segundo parámetro, este será la posición en el array.
Si se le pasa un tercer parámetro, este será el array en cuestión.

```
const letters = ["a", "b", "c", "d"];
letters.forEach((element) => console.log(element));
// Devuelve 'a' / 'b' / 'c' / 'd'
letters.forEach((element, index) => console.log(element,
index));  // Devuelve 'a' 0 / 'b' 1 / 'c' 2 / 'd' 3
letters.forEach((element, index, array) =>
console.log(array[0])); // Devuelve 'a' / 'a' / 'a' / 'a'
```

***Comprobaciones***

***Todos (every)***

El método **every**() permite comprobar si todos y cada uno de los elementos de un array cumplen la condición que se especifique en la funcion callback:

```
const letters = ["a", "b", "c", "d"];
letters.every((letter) => letter.length === 1); // true
```

Si expandimos el ejemplo anterior a un código más detallado, tendríamos el siguiente ejemplo equivalente, que quizás sea más comprensible para entenderlo:

```
const letters = ["a", "b", "c", "d"];
// Esta función se ejecuta por cada elemento del array
const condition = function (letter) {
  // Si el tamaño del elemento (string) es igual a 1
  if (letter.length == 1) {
    return true;
  }
  else {
    return false;
  }
};
// Si todos los elementos devuelven true, devuelve true
letters.every(condition);   // true
```

***Al menos uno (some)***

De la misma forma que el método anterior sirve para comprobar si todos los elementos del array cumplen una determinada condición, con some() podemos comprobar si al menos uno de los elementos del array, cumplen dicha condición definida por el callback.

```
const letters = ["a", "bb", "c", "d"];
letters.some((element) => element.length == 2);   // true
```

***Transformadores y filtros***

***Transformar (map)***

El método **map**() es un método muy potente y útil para trabajar con arrays, puesto que su objetivo es devolver un nuevo array donde cada uno de sus elementos será lo que devuelva la función callback por cada uno de los elementos del array original:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio",
"Heriberto"];
const nameSizes = names.map((name) => name.length);
nameSizes; // Devuelve [3, 5, 5, 9, 9]
```

***Filtrar (filter)***
El método **filter**() nos permite filtrar los elementos de un array y devolver un nuevo array con sólo los elementos que queramos. Para ello, utilizaremos la función callback para establecer una condición que devuelve true sólo en los elementos que nos interesen:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
const filteredNames = names.filter((name) => name.startsWith("P"));
filteredNames; // Devuelve ['Pablo', 'Pedro', 'Pancracio']
```

***Búsquedas***
***Buscar (find / findIndex)***
Dentro de las Array functions, existen dos métodos interesantes: **find**() y **findIndex**() . Ambos se utilizan para buscar elementos de un array mediante una condición, la diferencia es que el primero devuelve el elemento mientras que el segundo devuelve su posición en el array original. Veamos como funcionan:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
names.find((name) => name.length == 5);       // 'Pablo'
names.findIndex((name) => name.length == 5);  // 1
```

***findLast / findLastIndex***
De la misma forma, tenemos **findLastIndex**() y **findLast**(), que son las funciones equivalentes a **findIndex**() y **find**(), pero buscando elementos desde derecha a izquierda, en lugar de izquierda a derecha:

```
const names = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
names.findLast((name) => name.length == 5);  // 'Pedro'
names.findLastIndex((name) => name.length == 5);  // 2
```

***Acumuladores***
***Acumulador (reduce)***
Por último, nos encontramos con una pareja de métodos denominados **reduce**() y **reduceRight**(). Ambos métodos se encargan de recorrer todos los elementos del array, e ir acumulando sus valores (o alguna operación diferente) y sumarlo todo, para devolver su resultado final.
En este par de métodos, encontraremos una primera diferencia en su función callback, puesto que en lugar de tener los clásicos parámetros opcionales (element, index, array) que hemos utilizado hasta ahora, tiene (first, second, iteration, array), que funciona de forma muy similar, pero adaptado a este tipo de acumuladores.
En la primera iteración, first contiene el valor del primer elemento del array y second del segundo. En siguientes iteraciones, first es el acumulador que contiene lo que devolvió el callback en la iteración anterior, mientras que second es el siguiente elemento del array, y así sucesivamente. Veamos un ejemplo para entenderlo:

```
const numbers = [95, 5, 25, 10, 25];
numbers.reduce((first, second) => {
  console.log(`F=${first} S=${second}`);
  return first + second;
});
```

***Hacia izquierda (reduceRight)***
Gracias a esto, podemos utilizar el método reduce() como acumulador de elementos de izquierda a derecha y reduceRight() como acumulador de elementos de derecha a izquierda. Veamos un ejemplo de cada uno, realizando una resta en lugar de una suma:

```
const numbers = [95, 5, 25, 10, 25];
numbers.reduce((first, second) => first - second);      // 95 - 5 - 25 - 10 - 25. Devuelve 30
numbers.reduceRight((first, second) => first - second); // 25 - 10 - 25 - 5 - 95. Devuelve -110
```

***Desestructuración de arrays***

```
const elements = [5, 2];
const [first, last] = elements;    // first = 5, last = 2
const elements = [5, 4, 3, 2];
const [first, second] = elements;  // first = 5, second = 4, rest = discard
const elements = [5, 4, 3, 2];
const [first, , third] = elements; // first = 5, third = 3, rest = discard
const elements = [4];
const [first, second] = elements;  // first = 4, second = undefined
```

***Spread (Expandir)***

```
const debug = (param) => console.log(...param);
const array = [1, 2, 3, 4, 5];
debug(array);  // 1 2 3 4 5
```

***Rest (Agrupar)***
Sin embargo, veamos en el siguiente ejemplo, como colocamos los **...** en los parámetros de la función. Luego, al llamar a la función debug() le pasamos los 5 datos individuales:

```
const debug = (...param) => console.log(param);
debug(1, 2, 3, 4, 5);   // [1, 2, 3, 4, 5]
```

***Reestructuración de arrays***

Aprovechando estas características que hemos visto de desestructuración, también podríamos aprovecharlas para reestructurar un array y recrear arrays. Veámoslo con un ejemplo.

Tenemos un array de 2 elementos [3, 4] y queremos aprovecharlo para crear un nuevo array del 1 al 5. Vamos a hacer uso de la desestructuración para reaprovecharlo:

```
const pair = [3, 4];

// Usando desestructuración + spread
const complete = [1, 2, ...pair, 5];  // [1, 2, 3, 4, 5]

// Sin usar desestructuración
const complete = [1, 2, pair, 5];     // [1, 2, [3, 4], 5]
```

***¿Qué son los módulos (ESM)?***
A partir de ECMAScript se introduce una característica nativa denominada Módulos ES (ESM), que permite la importación y exportación de fragmentos de datos entre diferentes ficheros Javascript, eliminando las desventajas que teníamos hasta ahora y permitiendo trabajar de forma más flexible en nuestro código Javascript.

***Exportación de módulos***
Por defecto, un fichero Javascript no tiene módulo de exportación si no se usa un export al menos una vez en su código. Si se usa al menos un export, entonces tendrá un objeto llamado módulo de exportación, donde puede tener uno o múltiples datos. Existen varias formas de exportar datos mediante la palabra clave de Javascript export:

***Declaración y exportación***
Existen varias formas de exportar elementos. La más habitual, quizás, es la de simplemente añadir la palabra clave export a la izquierda de la declaración del elemento Javascript que deseamos exportar, ya sea una variable, una constante, una función, una clase u otro objeto más complejo:

```
export let number = 42;
export const hello = () => "Hello!";
export class CodeBlock { };
```

***Exportación post-declaración***
Si vienes del mundo de NodeJS, es muy probable que te resulte más intuitivo exportar módulos al final del fichero, ya que es como se ha hecho siempre en Node con los module.exports. Esta forma tiene como ventaja que es mucho más fácil localizar la información que ha sido exportada, ya que siempre estará al final del fichero.
Así pues, primero declarariamos la información y posteriormente, al final del fichero, exportamos lo que queramos:

```
let number = 42;
const hello = () => "Hello!";
const goodbye = () => "¡Adiós!";
class CodeBlock { };
export { number };
export { hello, goodbye as bye };
export { hello as greet };
```

***Exportación externa***
Esta modalidad es menos frecuente, pero puede ser interesante en algunas ocasiones. Se trata de añadir a nuestro módulo de exportación del fichero actual, todos los elementos exportados en el fichero math.js:

```
// CASO 1: Exporta todo lo exportado en el fichero math.js (abs, min, max, random)
export * from "./math.js";
// CASO 2: Exporta sólo abs, min y max del fichero math.js
export { abs, min, max } from "./math.js";
// CASO 3: Exporta todo lo exportado en el fichero math.js en un objeto con nombre
export const number = 42;
export * as math from "./math.js";
```

***Exportacion por defecto***

```
export const number = 42;    // Declaración y exportación
export default "Manz";       // Exportación por defecto
```

***Importación de módulos***
En Javascript, podemos utilizar import para hacer la operación inversa a export. Si habíamos mencionado que con export ponemos datos o elementos de un fichero .js a disposición de otros, con import podemos cargarlos y utilizarlos en el código de nuestro fichero actual.

**Importación con nombre**

```
import { nombre } from "./file.js";
import { number, element } from "./file.js";
import { brand as brandName } from "./file.js";
```

**Importación por defecto**

```
import nombre from "./math.js";
```

**Importación masiva**

```
import * as module from "./file.js";
```

**Importación de código**

```
import "./math.js";
```

**Importaciones remotas**

```
import { ceil } from "https://unpkg.com/lodash-es@4.17.21/lodash.js";
```

***Import estático vs dinámico***
En dicho capítulo, tratamos la palabra clave import que es lo que se conoce como un import estático, una forma de importar módulos de ficheros externos. Se colocan en la parte superior del fichero Javascript y son algo similar a lo siguiente:

```
import { name } from "./module.js";     // Static import
```

Sin embargo, existe otro tipo de importación en Javascript, popularmente conocida como import dinámico (dynamic import), que tiene el siguiente aspecto, ligeramente diferente al anterior:

```
import("./module.js")     // Dynamic import
  .then(data => { /* ... */ });
```

En este segundo caso, el import() tiene unos paréntesis que lo diferencian del anterior, y por la existencia del .then() sabemos que nos devuelve una promesa, por lo que se trata de código asíncrono.

***Diferencias***
Los import estáticos son muy útiles, pero tienen algunas desventajas si se presentan ciertos casos específicos. Los más frecuentes suelen ser los siguientes:

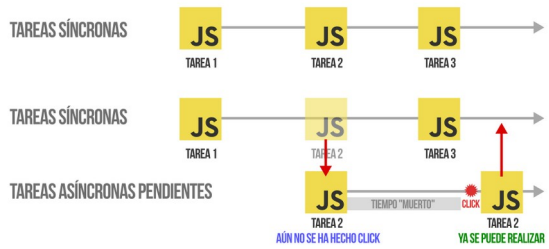
- Queremos importar un módulo si se cumple una determinada condición
- Queremos importar un módulo interpolando variables o constantes
- Queremos importar un módulo dentro de un ámbito específico
- Queremos importar un módulo desde un script normal (sin type="module")
- Queremos importar un fichero javascript (sin módulo) y ejecutarlo bajo demanda

## *¿Qué es la asincronía?*

La asincronía es uno de los conceptos principales que rige el mundo de Javascript. Cuando comenzamos a programar, normalmente realizamos tareas de forma síncrona, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que el orden o flujo del programa es sencillo y fácil de observar en el código:

```
primera_funcion();    // Tarea 1: Se ejecuta primero
segunda_funcion();    // Tarea 2: Se ejecuta cuando termina primera_funcion()
tercera_funcion();    // Tarea 3: Se ejecuta cuando termina segunda_funcion()
```

Sin embargo, en el mundo de la programación, tarde o temprano necesitaremos realizar operaciones asíncronas, especialmente en ciertos lenguajes como Javascript, donde tenemos que realizar tareas que tienen que esperar a que ocurra un determinado suceso que no depende de nosotros, y reaccionar realizando otra tarea sólo cuando dicho suceso ocurra.




## *¿Qué es la asincronía?*

Pero esto no es todo. Ten en cuenta que pueden existir múltiples tareas asíncronas, dichas tareas puede que terminen realizandose correctamente (o puede que no) y ciertas tareas pueden depender de otras, por lo que deben respetar un cierto orden. Además, es muy habitual que no sepamos previamente cuanto tiempo va a tardar en terminar una tarea, por lo que necesitamos un mecanismo para controlar todos estos factores: las promesas, las cuales veremos algunos capítulos más adelante.

Ejemplos de tareas asíncronas

En Javascript no todas las tareas son asíncronas, pero hay ciertas tareas que si lo son, y probablemente se entiendan mejor con ejemplos reales:

Un **fetch**() a una URL para obtener un archivo .json.
Un new Audio() de una URL con un .mp3 al que se hace .play() para reproducirlo.
Una tarea programada con setTimeout() que se ejecutará en el futuro.
Una comunicación desde Javascript a la API del sintetizador de voz del navegador.
Una comunicación desde Javascript a la API de un sensor del smartphone.

## *¿Qué es un callback?*

Como hemos dicho, las funciones callback no son más que un tipo de funciones que se pasan por parámetro a otras funciones. Además, los parámetros de dichas funciones toman un valor especial en el contexto del interior de la función.

Pero veamos un ejemplo. Imaginemos el siguiente bucle tradicional para recorrer un :

```
const list = ["A", "B", "C"];
for (let i = 0; i < list.length; i++) {
  console.log("i=", i, " list=", list[i]);
}
```

podemos hacer este mismo bucle utilizando el método forEach() del array al cuál le pasamos una función callback:

```
list.forEach(function(e,i) {
  console.log("i=", i, "list=", e);
});
```

Esto se puede reescribir como:

```
["A", "B", "C"].forEach((e,i) => console.log("i=", i, "list=", e));
```

Lo importante de este ejemplo es que se vea que la función callback que le hemos pasando a forEach() se va a ejecutar por cada uno de los elementos del array, y en cada iteración de dicha función callback, los parámetros e, i van a tener un valor especial:

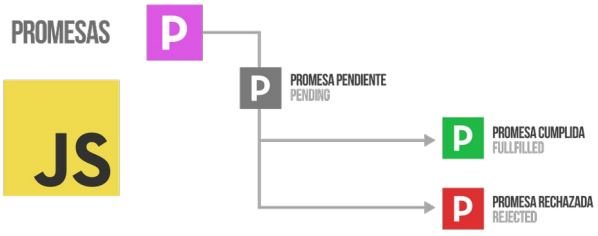
e es el elemento del array
i es el índice (posición) del array

## *Callbacks en Javascript*

```
setTimeout(function() {
  console.log("He ejecutado la función");
}, 2000);
//arrow function
setTimeout(() => console.log("He ejecutado la función"), 2000);
//diferencias
setTimeout(() => console.log("Código asíncrono."), 2000);
console.log("Código síncrono.");
```

## *¿Qué es una promesa?*

Como su propio nombre indica, una promesa es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:




## *Promesas en Javascript*

Las promesas en Javascript se representan a través de un objeto, y cada promesa estará en un estado concreto: pendiente, aceptada o rechazada.

## *Consumir una promesa*

La forma general de consumir una promesa es utilizando el .then() con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla:

```
fetch("/robots.txt").then(function(response) {
  /* Código a realizar cuando se cumpla la promesa */
});
```

Recuerda que podemos hacer uso del método .**catch**() para actuar cuando se rechaza una promesa:

```
fetch("/robots.txt")
  .then(function(response) {
    /* Código a realizar cuando se cumpla la promesa */
  })
  .catch(function(error) {
    /* Código a realizar cuando se rechaza la promesa */
  });
```

Además, se pueden encadenar varios .**then**() si se siguen generando promesas y se devuelven con un return:

```
fetch("/robots.txt")
  .then(response => {
    return response.text(); // Devuelve una promesa
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => { /* Código a realizar cuando se rechaza la promesa */ });
```

De hecho, usando arrow functions se puede mejorar aún más la legibilidad de este código, recordando que cuando sólo tenemos una sentencia en el cuerpo de la arrow function hay un return implícito:

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => console.log(data))
  .finally(() => console.log("Terminado."))
  .catch(error => console.error(data));
```

Observese además que hemos añadido el método .**finally**() para añadir una función callback que se ejecutará tanto si la promesa se cumple o se rechaza, lo que nos ahorrará tener que repetir la función en el .**then**() como en el .**catch**().

***Asincronía con promesas***

Vamos a implementar el ejercicio base que hemos comentado en el primer capítulo de este tema utilizando promesas. Observa que lo primero que haremos es crear un nuevo objeto promesa que «envuelve» toda la función de la tarea doTask().

Al new Promise() se le pasa por parámetro una función con dos callbacks, el primero resolve el que utilizaremos cuando se cumpla la promesa, y el segundo reject cuando se rechace:

```
/* Implementación con promesas */
const doTask = (iterations) => new Promise((resolve, reject) => {
  const numbers = [];
  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      reject({
        error: true,
        message: "Se ha sacado un 6"
      });
    }
  }
  resolve({
    error: false,
    value: numbers
  });
});
```

Como ves, se trata de una implementación muy similar a los callbacks que vimos en el apartado anterior, pero utilizan una  nativa para poder luego consumirla cómodamente:

```
doTask(10)
  .then(result => console.log("Tiradas correctas: ", result.value))
  .catch(err => console.error("Ha ocurrido algo: ", err.message));
```

***API de las promesas***

El objeto Promise de Javascript tiene varios métodos estáticos que podemos utilizar en nuestro código. Todos devuelven una promesa y son los que veremos en la siguiente tabla:

***Promise.all()***

El método Promises.all() funciona como un «todo o nada»: devuelve una promesa que se cumple cuando todas las promesas del  se cumplen. Si alguna de ellas se rechaza, Promise.all() también lo hace.

***Promise.allSettled()***

El método Promises.allSettled() funciona como un «todas procesadas»: devuelve una promesa que se cumple cuando todas las promesas del  se hayan procesado, independientemente de que se hayan cumplido o rechazado.

***Promise.any()***

El método Promise.any() funciona como «la primera que se cumpla»: Devuelve una promesa con el valor de la primera promesa individual del  que se cumpla. Si todas las promesas se rechazan, entonces devuelve una promesa rechazada.

***Promise.race()***

El método Promise.race() funciona como una «la primera que se procese»: la primera promesa del  que sea procesada, independientemente de que se haya cumplido o rechazado, determinará la devolución de la promesa del Promise.race(). Si se cumple, devuelve una promesa cumplida, en caso negativo, devuelve una rechazada.

## *Async/Await*

### *La palabra clave async*

En primer lugar, tenemos la palabra clave **async**. Esta palabra clave se colocará previamente a function, para definirla así como una función asíncrona, el resto de la función no cambia:

```
async function funcion_asincrona() {
  return 42;
}
```

En el caso de que utilicemos **arrow function**, se definiría como vemos a continuación, colocando el async justo antes de los parámetros de la **arrow function**:

```
const funcion_asincrona = async () => 42;
```

Al ejecutar la función veremos que ya nos devuelve una que ha sido cumplida, con el valor devuelto en la función (en este caso, 42). De hecho, podríamos utilizar un .then() para manejar la promesa:

```
funcion_asincrona().then(value => {
  console.log("El resultado devuelto es: ", value);
});
```

### *La palabra clave await*

Cualquier función definida con async, o lo que es lo mismo, cualquier puede utilizarse junto a la palabra clave await para manejarla. Lo que hace await es esperar a que se resuelva la promesa, mientras permite continuar ejecutando otras tareas que puedan realizarse:

```
const funcion_asincrona = async () => 42;
const value = funcion_asincrona();
// Promise { <fulfilled>: 42 }
const asyncValue = await funcion_asincrona();  // 42
```

Observa que en el caso de value, que se ejecuta sin await, lo que obtenemos es el valor devuelto por la función, pero «envuelto» en una promesa que deberá utilizarse con .then() para manejarse. Sin embargo, en asyncValue estamos obteniendo un tipo de dato , guardando el valor directamente ya procesado, ya que await espera a que se resuelva la promesa de forma asíncrona y guarda el valor.

### *Asincronía con async/await*

Volvamos al ejemplo que hemos visto en los anteriores capítulos. Recordemos que la función doTask() realiza 10 lanzamientos de un dado y nos devuelve los resultados obtenidos o detiene la tarea si se obtiene un 6.

La implementación de la función sufre algunos cambios, simplificándose considerablemente. En primer lugar, añadimos la palabra clave async antes de los parámetros de la arrow function. En segundo lugar, desaparece cualquier mención a promesas, se devuelven directamente los objetos, ya que al ser una función async se devolverá todo envuelto en una promesa:

```
const doTask = async (iterations) => {
  const numbers = [];
  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      return {
        error: true,
        message: "Se ha sacado un 6"
      };
    }
  }
  return {
    error: false,
    value: numbers
  };
}
```

Pero donde se introducen cambios considerables es a la hora de consumir las promesas con async/await. No tendríamos que utilizar .then(), sino que podemos simplemente utilizar await para esperar la resolución de la promesa, obteniendo el valor directamente:

```
const resultado = await doTask(10);
 // Devuelve un objeto, no una promesa
```

**Top-level await**

En principio, el comportamiento de await solo permite que se utilice en el interior de funciones declaradas como async. Por lo que, si el ejemplo anterior lo ejecutamos en una consola de Javascript, funcionará correctamente (estamos escribiendo comandos de forma asíncrona), pero si lo escribimos en un fichero, probablemente nos aparecerá el siguiente error:

*Uncaught SyntaxError: await is only valid in async function*

Esto ocurre porque, como bien dice el mensaje de error, estamos ejecutando await en el contexto global de la aplicación, y debemos ejecutarlo en un contexto de función asíncrona. Para corregirlo, podemos añadir un <button> en el HTML y modificar la linea anterior del await:

```
document.querySelector("button").addEventListener("cli
ck", async () => {
  const resultado = await doTask(10);
  console.log(resultado);
});
```