Actividad 4: Fundamentos de JavaScript

Objetivos:

- Comprender las bases y la evolución de JavaScript
- Familiarizarse con las especificaciones y documentación de JavaScrip
- Mejorar la calidad del código mediante herramientas y convenciones
- Desarrollar habilidades para el manejo eficiente de datos y estructuras
- Aplicar lógica condicional y control de flujo
- Manipular cadenas, arreglos y objetos en JavaScript
- Dominar funciones y programación orientada a objetos

Esta actividad sintetiza las partes más básicas (arrays, objetos, cadenas y tipos de datos) de JavaScript para hacer un análisis más profundo de las partes más complejas, como funciones y closures. Además, aprenderás sobre los últimos cambios en JavaScript introducidos por la última especificación.También aprenderemos cómo JavaScript se ha convertido en un estándar en la toma de decisiones cuando se solicita un cambio para el lenguaje.

Adicionalmente, revisaremos algunas herramientas que nos ayudarán a escribir mejor JavaScript utilizando linters, herramientas de depuración y documentación adecuada para el código.

Esta es una actividad de tipo descriptiva.

JavaScript es un lenguaje poderoso

JavaScript es un lenguaje muy poderoso. Se utiliza en el frontend, backend, móvil, escritorio, IoT, y más. Es muy flexible y es muy fácil comenzar, pero también es muy difícil de dominar en profundidad.

Hay una cita muy famosa (https://www.crockford.com/javascript/javascript.html) de Douglas Crockford que dice:

JavaScript es el lenguaje de programación más incomprendido del mundo.

JavaScript es un lenguaje multi-paradigma, lo que significa que puedes usar diferentes estilos de programación, como programación orientada a objetos, programación funcional o programación declarativa. Esto es muy útil porque puedes usar el estilo de programación que mejor se adapte a tus necesidades. Es un lenguaje muy dinámico, lo que significa que puedes cambiar el comportamiento del lenguaje en tiempo de ejecución. Gracias a JavaScript, puedes aprender conceptos complejos, como **closures** y **prototipos,** y usarlos para crear aplicaciones muy poderosas y complejas. Pero también puedes usarlos para crear aplicaciones muy confusas y difíciles de mantener.

La versión – TC39

JavaScript se está volviendo viejo; fue creado en 1995 por Brendan Eich en Netscape Communications Corporation. Originalmente se llamaba Mocha, pero fue renombrado a LiveScript y finalmente a JavaScript.

La primera versión de JavaScript fue lanzada en 1996. Se llamaba ECMAScript 1 (ES1) y fue estandarizada por el European Computer Manufacturers Association (ECMA) en 1997.

La versión – ECMAScript

A lo largo de los años, se añadieron muchas nuevas características al lenguaje, como clases, módulos y funciones flecha. Las nuevas características se añadieron al lenguaje a través de un proceso de propuesta de ECMAScript (https://github.com/tc39/proposals) que son gestionadas directamente por el TC39 (https://tc39.es/process-document/), que se refiere a un comité de ECMA que es responsable de la evolución del lenguaje.

Desde 1997 hasta 2015, se añadieron nuevas características al lenguaje cada pocos años, pero en 2015, el TC39 decidió lanzar una nueva versión del lenguaje cada año, lo que significa que el lenguaje está evolucionando más rápido que nunca. Esto también nos ayuda con la adopción de las nuevas características porque no necesitamos esperar muchos años para usarlas en entornos de producción.

Actualmente, la última versión del lenguaje es ECMA-262 2023 (https://tc39.es/ecma262/), que fue lanzada en junio de 2023.

¿Qué se incluye en la próxima versión de JavaScript?

Para agregar nuevas características al lenguaje, el comité TC39 tiene un proceso que se divide en etapas. Cualquiera puede enviar una propuesta al comité TC39, pero no es una tarea fácil, porque la propuesta necesita ser aprobada por el comité antes de ser implementada.

Puedes encontrar todas las propuestas en el repositorio de GitHub del TC39 (https://github.com/tc39/proposals).

¿Qué no se incluye en la especificación de JavaScript?

La especificación de JavaScript es muy grande, pero no incluye muchas APIs que se utilizan comúnmente en aplicaciones JavaScript, como las APIs del navegador y las APIs de Node.js.

Si estás utilizando JavaScript en el navegador, puedes usar las APIs del navegador, como Document Object Model (DOM). Si estás utilizando JavaScript en Node.js, puedes usar las APIs de Node.js, como el sistema de archivos o HTTP.

Al final del día, JavaScript es solo un lenguaje de programación. Si estás acostumbrado a construir aplicaciones JavaScript en el navegador, es posible que estés familiarizado con muchas APIs que no están incluidas en la especificación de JavaScript y no están disponibles en Node.js. Por ejemplo, el objeto window (https://developer.mozilla.org/en-US/docs/Web/API/Window) está disponible en el navegador, pero no está disponible en Node.js.

Si bien ECMA-262 (https://262.ecma-international.org/14.0/) es una gran fuente de información, no es muy amigable para principiantes.

Importante: La fuente más completa de información es MDN Web Docs (https://developer.mozilla.org/en-US/docs/Web/JavaScript), que es una documentación impulsada por la comunidad. Es muy completa y se actualiza regularmente e incluso está traducida a otros idiomas.

Si estás familiarizado con el desarrollo frontend, es posible que hayas usado MDN Web Docs antes, porque es la principal fuente de información para las APIs del navegador, como el DOM (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction) y la Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API).

Si estás buscando documentación más concisa, puedes usar W3Schools (https://www.w3schools.com/js/default.asp), que es una gran fuente de información para principiantes, con muchos ejemplos.

Finalmente, si estás buscando una respuesta específica a una pregunta, puedes usar Stack Overflow (https://stackoverflow.com/questions/tagged/javascript), que es un sitio web de preguntas y respuestas impulsado por la comunidad.

Linting del código JavaScript

Linting es el proceso de ejecutar un programa que analizará tu código en busca de posibles errores. Es muy útil para detectar errores antes de ejecutar tu código, para que puedas corregirlos antes de que causen problemas.

JavaScript es un lenguaje muy flexible, lo que significa que es muy fácil cometer errores. A medida que te acostumbras más, cometerás menos errores, pero siempre es bueno tener un linter para ayudarte.

Usaremos ESLint (https://eslint.org/) para linting nuestro código, pero hay otras opciones disponibles, como JSLint (https://www.jslint.com/) y JSHint (https://jshint.com/).

Configurar un linter no es una tarea trivial, pero vale la pena el esfuerzo. Hay muchas reglas disponibles y no es fácil saber cuáles usar. Te recomiendo encarecidamente que uses las reglas estándar (https://standardjs.com/), que son una de las más populares y utilizadas por muchos proyectos de código abierto (incluyendo Node.js, Express y MongoDB) y empresas. Puedes encontrar todas las reglas disponibles en la página de JavaScript Standard Style (https://standardjs.com/rules.html).

Comentando el código JavaScript

Tienes múltiples opciones para incluir comentarios en tu código:

// Comentario de una línea


```
/*
Comentario
multilínea
*/
```

Si eres nuevo en JavaScript, te recomiendo que uses muchos comentarios para ayudarte a entender qué está sucediendo en tu código. A medida que te vuelvas más experimentado, necesitarás menos comentarios. Los comentarios también ayudan a otros desarrolladores a leer y entender tu código.

Usando JSDoc

Si necesitas orientación sobre cómo escribir buenos comentarios, puedes usar la sintaxis de JSDoc (https://jsdoc.app/). Otro beneficio adicional de usar JSDoc es que puedes usarlo para autogenerar documentación para tu código.

Esta es una solución bastante popular. Por ejemplo, Lodash utiliza este enfoque. Usa los siguientes enlaces para ver cómo se documenta el método _.chunk:

JSDoc en la práctica: https://github.com/lodash/lodash/blob/4.17.15/lodash.js#L6818

Documentación generada automáticamente por JSDocs: https://lodash.com/docs/4.17.15#chunk

Imprimiendo valores y depurando

El objeto **console** no es estándar; no es parte del lenguaje JavaScript, pero es proporcionado por el navegador y Node.js. Puedes usarlo para imprimir mensajes en la consola, lo cual es muy útil para fines de depuración y para los propósitos de este curso, para seguir los ejemplos. Es bastante común usarlo para imprimir el valor de una variable. Toma el siguiente ejemplo:

```
const name = "ZZZ";
console.log(name); // ZZZ
```

Sí, puedes usar console.log para imprimir múltiples valores al mismo tiempo, separados por comas, e incluso incluir información adicional para explicar lo que estás imprimiendo. No tienes que preocuparte por el tipo de la variable como en otros lenguajes; **console.log** lo hará por ti.

En algunos casos, necesitarás ayudar a console.log a imprimir el valor de una variable; por ejemplo, si deseas imprimir un objeto, a veces terminas obteniendo [object, object] o algo similar como mensaje de salida. En este caso, necesitarás usar **console.log(JSON.stringify(object))** para imprimir el objeto como una cadena:

```
const data = {
  nestedData: {
    moreNestedData: {
```


```
    value: 1
  }
 }
};
```

```
console.log(data); // [object, object]
console.log(JSON.stringify(data)); // {"nestedData":{"moreNestedData":{"value":1}}}
```

Con el tiempo, los motores de JavaScript mejoran la salida de la consola, por lo que este simple ejemplo podría imprimirse como se espera en tu navegador; pero ciertos objetos complejos aún pueden necesitar ser convertidos en cadena, por ejemplo, la respuesta de una solicitud HTTP larga.

El objeto **console** ofrece muchos métodos para imprimir la información en diferentes formatos, lo que mejorará mucho tu experiencia como desarrollador. La documentación está disponible para navegadores web (https://developer.mozilla.org/en-US/docs/Web/API/console) y para Node.js (https://nodejs.org/api/console.html).

Variables y constantes

Usamos variables para almacenar valores y usamos constantes para almacenar valores que no cambiarán. En JavaScript, podemos usar la palabra clave let para declarar una variable y la palabra clave const para declarar una constante. Antes de ES6, solo podíamos usar la palabra clave var para declarar variables, pero ya no se recomienda usarla.

Convenciones de nomenclatura

En JavaScript, es muy común usar camelCase para nombrar variables y constantes, pero también se admiten otras convenciones, como snake_case y PascalCase. También es posible comenzar variables con símbolos, pero no se recomienda.

Hay algunas limitaciones que debemos considerar al nombrar variables y constantes:

- Evita comenzar con un símbolo, como $resource
- No comiences con un número, como 1variable
- No uses espacios, como const my variable = 1
- No uses palabras reservadas, como const const = "constant"

let versus const

Usamos **let** para declarar variables y **const** para declarar constantes. La principal diferencia es que podemos reasignar un valor a una variable, pero no podemos reasignar un valor a una constante. Aquí hay un ejemplo de reasignar un valor a una variable:

```
let userName = "Kapu Mota";
console.log(userName); // Kapu Mota
userName = "Kapu Mota";
```


```
console.log(userName); // Kapu Mota
```

Como podemos ver aquí, no podemos reasignar un valor a una constante:

```
const userName = "Kapu Mota";

console.log(userName); // Kapu Mota

userName = "Ire"; // TypeError: Assignment to constant variable.
```

Es importante notar que podemos cambiar el valor de una constante si el valor es un objeto, pero no podemos reasignar un nuevo valor a la constante:

```
const user = {
  name: "Kapu Mota"
}

console.log(user.name); // Kapu Mota

user.name = "Kapu Kapu";

console.log(user.name); // Kapu Kapu

user = "Mr. Kapu"; // TypeError: Assignment to constant variable.
```

En JavaScript, hay otro mecanismo que debes entender. **Hoisting** es un comportamiento en JavaScript donde las declaraciones de variables y funciones se mueven a la parte superior de su alcance contenedor durante la fase de compilación. Esto se hace para optimizar el código, pero puede tener algunos efectos secundarios. Puedes encontrar una gran guía en https://www.freecodecamp.org/news/what-is-hoisting-in-javascript-3.

Entendiendo los tipos de datos

En JavaScript, existen varios tipos primitivos. Podemos agruparlos en dos grupos: antes de ES6 (undefined, object, boolean, number, string, y function) y después de ES6 (bigint y symbol). Para verificar el tipo de una variable, podemos usar el operador typeof.

undefined

No todos los lenguajes tienen un tipo **undefined,** pero JavaScript sí. Se usa para representar la ausencia de un valor. También se usa como valor predeterminado para variables no inicializadas.

object

El tipo object se usa para representar una colección de datos. Es un tipo muy genérico y se usa para representar muchas cosas diferentes, como arrays (listas), objetos (diccionarios), instancias de clases y null.

boolean

El tipo boolean se usa para representar un valor lógico. Puede ser true o false. Este tipo también puede generarse usando la función Boolean, ya que todo en JavaScript puede convertirse en un valor booleano.

number

El tipo number se usa para representar un valor numérico. Puede ser un número entero o un número de punto flotante. También se usa para representar valores numéricos especiales como Infinity, -Infinity y NaN (que significa Not a Number).

string

El tipo string se usa para representar una secuencia de caracteres. Puede crearse explícitamente usando comillas simples ('), comillas dobles ("), o acentos graves (') o implícitamente usando la función String o expresiones.

function

El tipo function se usa para representar una función. Las funciones en JavaScript son muy poderosas. Hay dos formas de crear una función, utilizando la palabra clave function o utilizando la sintaxis de funciones flecha.

bigint

bigint se introdujo en ES6 para trabajar con números grandes. number está limitado a valores entre -(2^53 – 1) y 2^53 – 1

symbol

El tipo symbol se usa para representar un identificador único. Es un nuevo tipo que se introdujo en ES6; realmente no necesitas estar familiarizado con él para seguir este curso.

Explorando números

JavaScript tiene un buen soporte para operaciones matemáticas y fechas, pero a veces puede ser más complicado y limitado que otros lenguajes de programación, por lo que muchos desarrolladores usan bibliotecas especializadas cuando la aplicación requiere matemáticas avanzadas. Por ejemplo, si necesitas trabajar con vectores, matrices o números complejos, debes usar una biblioteca como Math.js (https://mathjs.org/).

Aquí hay un ejemplo típico del problema de la precisión de punto flotante:

```
console.log(0.1 + 0.2); // 0.30000000000000004

console.log(0.1 + 0.2 === 0.3); // false
```

Como puedes ver, el resultado de 0.1 + 0.2 no es 0.3, sino 0.30000000000000004. Esto se debe a que JavaScript utiliza el estándar IEEE 754 (https://en.wikipedia.org/wiki/IEEE_754) para representar números, y no es posible representar todos los números decimales en binario. Este es un problema

común en muchos lenguajes de programación; no es un problema exclusivo de JavaScript. Pero puedes solucionarlo usando las funciones Number y toPrecision ya que implícitamente convertirás de número a cadena y luego de vuelta a número:

let impreciseOperation = 0.1 + 0.2;

Number(impreciseOperation.toPrecision(1)) === 0.3; // true

Como puedes ver, hay algunos casos extremos que no son fáciles de entender o resolver intuitivamente. La mayoría de las veces, no necesitarás preocuparte por esto, pero es importante saber que este problema existe y puedes usar bibliotecas si no tienes suficiente experiencia con números en JavaScript.

Operadores aritméticos

JavaScript tiene los operadores aritméticos esperados, +, -, *, /, %, y **, e indica prioridad con paréntesis como en cualquier lenguaje moderno.

Operadores de asignación

JavaScript tiene los operadores de asignación esperados, =, +=, -=, *=, /=, %=, y **=, como en otros lenguajes.

También puedes usar ++ y -- para incrementar y decrementar una variable. Este operador puede añadirse antes o después de la variable, y cambiará el valor de la variable antes o después de la operación:

let a = 5;

console.log(a++); // 5

console.log(a); // 6

console.log(++a); // 7

console.log(a); // 7

JavaScript también admite operaciones a nivel de bits, por lo que puedes trabajar con un conjunto de 32 bits (ceros y unos), en lugar de números decimales, hexadecimales u octales. Puedes consultar la documentación completa aquí: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_operators#bitwise_operators.

Métodos útiles

Hay métodos que son clave para realizar operaciones matemáticas o transformaciones en el trabajo diario:

Number.prototype.toFixed():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed

Number.prototype.toPrecision():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Number/toPrecision

Number.parseInt():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Number/parseInt

Number.parseFloat():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Number/parseFloat

El objeto Math

JavaScript tiene un objeto Math integrado que proporciona muchos métodos útiles para realizar operaciones matemáticas. Enumeraré algunos de ellos aquí, pero puedes encontrar la lista completa en la documentación de MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math).

Hay métodos que son clave para realizar operaciones matemáticas o transformaciones en el trabajo diario:

Math.random(): Devuelve un número pseudoaleatorio de punto flotante entre 0 (inclusive) y 1 (exclusive)
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Math.max(): Devuelve el valor numérico máximo entre los argumentos pasados
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math/max

Math.min(): Devuelve el valor numérico mínimo entre los argumentos pasados
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math/min

Math.floor(): Devuelve el número resultante de redondear un número hacia abajo al número entero más cercano que sea menor o igual al número dado
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math/floor

Otros números

En JavaScript, hay algunos valores especiales que son números, pero no son números reales. Estos valores son **NaN** e **Infinity**.

Not a Number (NaN)

NaN es un valor especial que representa Not a Number. Es el resultado de una operación matemática inválida o indefinida, por ejemplo, dividir 0 por 0, o multiplicar Infinity por 0. Puedes usar isNaN() para verificar si un valor es NaN (https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/isNaN).

Infinity

Infinity es un valor especial que representa infinito. Es el resultado de una operación matemática que excede el número más grande posible. Puedes usar isFinite() para verificar si un valor es finito (https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/isFinite).

Explorando el objeto Dates

Las fechas son un tema complejo para cualquier lenguaje de programación o sistema, ya que debes tener en cuenta muchas cosas, como las zonas horarias. Si necesitas trabajar intensivamente con fechas, considera usar una biblioteca como **Lunox** (https://github.com/moment/luxon/) o **date-fns** (https://date-fns.org/).

Para escenarios más simples, puedes usar el objeto Date integrado y la API Intl (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl) para formatear fechas.

La API ofrece varias formas de generar el objeto **Date** utilizando números, cadenas o varios argumentos. Además, tienes getters y setters para leer y modificar partes específicas, como el año o los milisegundos. También es posible realizar operaciones como comparar o agregar tiempo.

Durante muchos años, la única forma de formatear fechas en JavaScript fue utilizando el método **toLocaleString()**. Este método sigue siendo válido, pero tiene muchas limitaciones, especialmente cuando quieres comparar fechas de manera legible (por ejemplo, hace 3 días o hace 2 semanas).

En el pasado, necesitábamos usar bibliotecas externas para lograr esto, pero ahora podemos usar la API Intl (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl) para formatear fechas.

En el siguiente código, puedes ver cómo generar, manipular y formatear fechas:

```
const jsDateAnnouncement = new Date(818031600000);

const currentDate = new Date();

const diff = jsDateAnnouncement - currentDate;

const formatter = new Intl.RelativeTimeFormat('en', {
   numeric: 'auto'
});

const diffInDays = Math.round(diff / 86400000);

const diffInYears = Math.round(diffInDays / 365);

const diffInText = formatter.format(diffInDays, 'day');

console.log('JavaScript fue presentado al mundo hace ${formatter.format(diffInDays, 'day')}');

// JavaScript fue presentado al mundo hace 10,094 días

console.log('JavaScript fue presentado al mundo hace ${formatter.format(diffInYears, 'year')}');
```

// JavaScript fue presentado al mundo hace 28 años.

El resultado puede variar en tu máquina, ya que habrá pasado algún tiempo desde que escribí este código. Por lo tanto, ten en cuenta que la salida que observes puede diferir de la mía.

El TC39 está consolidando esta API, que incluye muchas características para formatear fechas, números, monedas y más. Te recomiendo que sigas el progreso de la propuesta y su implementación en los motores de JavaScript.

Sentencias condicionales

Existen muchas formas de escribir sentencias condicionales en JavaScript, pero las más comunes son if, switch y el operador ternario (?:).

Operadores de comparación matemática

Para operaciones matemáticas, tenemos los siguientes operadores: >, <, >=, y <=. Se utilizan para comparar dos valores y devolver un valor booleano. Su uso es el mismo que en la mayoría de los lenguajes de programación modernos.

Operadores de igualdad

Los operadores de igualdad se utilizan para comparar dos valores y devolver un valor booleano. Hay dos tipos de operadores de igualdad: estricta (=== y !==) y no estricta (== y !=).

El operador de igualdad estricta no se puede usar para comparar tipos no primitivos (como objeto, array y función) y ciertos valores como NaN, ya que siempre devolverá false:

```
console.log([1,2] === [1,2]) // false

console.log({ name: 'Jaaa' } === { name: 'Jaaa' }); // false

console.log(NaN === NaN); // false
```

No se recomienda usar operadores de igualdad no estricta, ya que pueden llevar a resultados inesperados, porque este operador no verifica el tipo de los valores:

```
console.log(1 == '1'); // true

console.log(1 != '1'); // false
```

Operadores lógicos

Es posible combinar múltiples condiciones utilizando operadores lógicos. Hay tres operadores lógicos, &&, ||, y !, y algunas variaciones de ellos, &&= y ||=, que se utilizan para reducir la cantidad de código para ciertas operaciones. No cubriremos todos en este curso.

Puedes combinar operadores para construir validaciones más complejas:

```
const num = 2

console.log((num == 2) && (3 >= 6)); // false
```

```
console.log((num > 3) || (17 <= 40)); // true
```

El operador NOT (!)

El operador NOT se utiliza para invertir el valor de un booleano. Devolverá true si el valor es false, y false si el valor es true:

```
console.log(!true); // false

console.log(!false); // true
```

Este ejemplo no es claro acerca de todas las posibilidades que se ofrecen, así que intentemos construir una analogía con una estructura más detallada, Boolean(value) === false. Básicamente, el operador ! convierte el valor en un booleano y luego lo compara con un valor false.

Igualdad en JavaScript

Debido a la naturaleza de JavaScript, es posible usar cualquier valor como condición. La condición se evaluará como un booleano, y si el valor es truthy, la condición será true. Si el valor es falsy, la condición será false. Esto puede ser un poco confuso, así que exploremos el método Boolean para entender cómo se transforman diferentes valores de datos:

```
// Los valores truthy:

console.log("Cadena:", Boolean("Kapu")  );

console.log("1235:", Boolean(1235));

console.log("-1235:", Boolean(-1235));

console.log("Objeto:", Boolean({text: "hola"}));

console.log("Array:", Boolean(["manzana", -1, false]));

console.log("Función:", Boolean(function(){}));

console.log("Función flecha:", Boolean(() => {}));

// Los valores falsy:

console.log("Cadena vacía:", Boolean(""));

console.log("0:", Boolean(0));

console.log("-0:", Boolean(-0));

console.log("null:", Boolean(null));

console.log("undefined:", Boolean(undefined));

console.log("NaN:", Boolean(NaN));
```

Podemos concluir fácilmente que los valores vacíos (como null, undefined, una cadena vacía o NaN) y 0 son falsy, y los valores con tipos de datos complejos (como objetos y funciones) o cadenas no vacías y números no nulos son truthy.

Esto es bastante conveniente cuando queremos verificar si un valor está vacío o no, como en el siguiente ejemplo:

```
function checkValue (value) {
  if(!value) {
    throw new Error ("¡El valor es inválido! Inténtalo de nuevo.")
  }
}
```

Esta transformación y comparación de Boolean pueden convertirse en una situación muy compleja si deseas comparar diferentes tipos de datos y valores, por ejemplo, Boolean([]) === Boolean({}). Puedes explorar este tema en detalle en la documentación de MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness). Pero en general, no necesitas ser un experto en esta área para seguir este curso.

Puedes obtener una mejor comprensión de este tema explorando la Tabla de Igualdad de JavaScript por Dorey en https://github.com/dorey/Javascript-Equality-Table/.

El operador de coalescencia nula (??)

El operador de coalescencia nula es un nuevo operador que se introdujo en ES2020. Se utiliza para verificar si un valor es null o undefined; si lo es, devolverá un valor predeterminado:

```
const name = null ?? "Ire Ire";
console.log(name); // Ire Ire
```

La sentencia if

La sentencia if es la forma más común de escribir una sentencia condicional. Ejecutará el código dentro del bloque si la condición es true. La sentencia else nos permite dar seguimiento cuando la condición no se cumple al ejecutar el código que está en la sentencia else. La sentencia else if es una variación de la sentencia if. Ejecutará el código dentro del bloque si la condición es true. Si la condición es false, ejecutará el código dentro del bloque else. Puedes agregar tantas sentencias else if como necesites:

```
const condition = true
const condition2 = true
if(condition) {
```


```
  console.log("La condición es verdadera")
} else if (condition2) {
  console.log("La condición2 es verdadera")
} else {
  console.log("La condición y la condición2 son falsas")
}
```

Puedes cambiar los valores en condition y condition2 para familiarizarte más con el comportamiento de las estructuras condicionales.

Uso de return

La sentencia return se utiliza ampliamente para evitar el uso de sentencias else y permite un código más limpio. Aquí hay un ejemplo:

```
const condition = true;
if(condition) {
  return console.log("La condición es verdadera");
}
console.log("La condición es falsa");
```

La sentencia switch

La sentencia switch es una buena opción cuando deseas comparar una variable con múltiples valores. Es útil cuando deseas asignar un valor a una variable según una condición.

La estructura switch está compuesta por la palabra clave switch, seguida de la variable que deseas comparar, y luego un bloque de sentencias case. Cada sentencia case está compuesta por la palabra clave case, seguida del valor que deseas comparar, "y luego un doble dos puntos (:). Después del doble dos puntos," puedes escribir el código que deseas ejecutar si la condición es true. La sentencia default es opcional y se ejecutará si ninguna de las sentencias case es true, como else cuando se utilizan sentencias if.

La sentencia **break** se utiliza para detener la ejecución de la sentencia switch. Si no agregas la sentencia break, el código continuará ejecutando la siguiente sentencia case. Aquí tenemos un ejemplo combinado:

```
const extension = ".md";
switch (extension) {
 case ".doc":
  console.log("Esta extensión .doc será obsoleta pronto")
```

```
 case ".pdf":
 case ".md":
 case ".svg":
   console.log("¡Felicidades! Puedes abrir este archivo");
   break;
 default:
   console.log('${extension} no es compatible');
}
```

Operador ternario

El operador ternario es una abreviatura para las sentencias if y else. Es una buena opción cuando deseas asignar un valor a una variable según una condición.

La estructura está compuesta por la condición, seguida de un signo de interrogación, ?, luego el valor que deseas asignar si la condición es true, seguido de un doble dos puntos (:), y luego el valor que deseas asignar si la condición es false: condition ? valueIfTrue : valueIfFalse.

Veamos un ejemplo con las sentencias if y else:

```
const isMember = true;
console.log('El pago es ${isMember ? "20.00€" : "50.00€"}');
// El pago es 20.00€
```

El operador ternario puede anidar múltiples operadores ternarios, pero no se recomienda porque puede ser extremadamente difícil de leer. Además, es posible usar el operador ternario para realizar múltiples operaciones, pero no se recomienda porque puede ser extremadamente difícil de leer, incluso si usas paréntesis.

Entendiendo los bucles

Hay muchas formas de crear bucles en JavaScript, pero los más comunes son las sentencias for y while y sus variaciones que son específicas para arrays y objetos. Además, las funciones en JavaScript se pueden utilizar para crear bucles cuando se utiliza la recursión. En esta parte, solo veremos las sentencias for, while y do...while.

while

La sentencia while crea un bucle que ejecuta un bloque de código mientras la condición sea true. La condición se evalúa antes de ejecutar el bloque de código:

```
let i = 1;
while (i <= 10) {
```

```
    console.log(i);
    i++;
};
```

do...while

La sentencia do...while crea un bucle que ejecuta un bloque de código al menos una vez, incluso si la condición no se cumple, y luego repite el bucle mientras la condición sea true. La condición se evalúa después de ejecutar el bloque de código:

```
let i = 0;
do {
    console.log('Valor de i: ${i}');
    i++;
} while (false);
// Valor de i: 0
```

for

La sentencia for crea un bucle que consta de tres expresiones opcionales, encerradas entre paréntesis y separadas por punto y coma, seguidas de una sentencia ejecutada en el bucle:

```
for (let i = 0; i < 10; i++) {
 console.log(i);
}
```

La primera expresión se ejecuta antes de que comience el bucle. Por lo general, se usa para inicializar la variable que actuará como contador.

La segunda expresión es la condición que se evalúa antes de ejecutar el bloque de código. Si la condición es true, se ejecuta el bloque de código. Si la condición es false, el bucle se detiene.

La tercera expresión se ejecuta después de que se ejecuta el bloque de código. Por lo general, se usa para incrementar o decrementar la variable contador.

Esta estructura es bastante flexible y algunos desarrolladores tienden a abusar de ella. Veamos un ejemplo con mala legibilidad:

```
for (let i = 0, x = 1, z = 2, limit = 10; i <= limit; x *= z, i++ ) {
    console.log('i: ${i}. x: ${x}. z: ${z}');
}
```


```
// i: 0. x: 1. z: 2

// ...

// i: 10. x: 1024. z: 2
```

Los problemas de legibilidad se deben a la gran cantidad de variables definidas y actualizadas en el bucle for. Es importante recordar que escribimos código que otros programadores pueden entender en el futuro. Veamos el mismo código con un enfoque más legible:

```
let x = 1;

const z = 2, limit = 10;

for (let i = 0; i <= limit; i++ ) {

   console.log('i: ${i}. x: ${x}. z: ${z}');

   x *= z

}
```

Ya puedes notar la diferencia; lleva menos tiempo y esfuerzo entenderlo.

Uso de cadenas en JavaScript

Las cadenas son valores primitivos. Son una secuencia de caracteres. Hay tres formas de crear cadenas en JavaScript: usando comillas simples, ', comillas dobles, " o acentos graves, '.

```
console.log('Hola Mundo');

console.log("Hola Mundo");

console.log('Hola Mundo');
```

Las cadenas son inmutables, lo que significa que una vez creadas, no se pueden modificar, pero puedes sobrescribir las variables o referencias dependiendo de la estructura de datos. Por lo tanto, todos los métodos que utilices para modificar una cadena devolverán una nueva cadena (o array):

Las cadenas de plantilla te permiten usar marcadores de posición, ${}, para insertar variables o expresiones dentro de una cadena. También se añade soporte para múltiples líneas:

```
const name = "Kapumota";

console.log('¡Hola ${name}!') //¡Hola Kapumota!
```

Métodos importantes

Hay muchas formas de realizar operaciones con cadenas, pero en esta parte, solo veremos los métodos más importantes que utilizarás en tu trabajo diario:

String.prototype.indexOf(): Encuentra el índice de la primera aparición de una subcadena especificada dentro de una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf

String.prototype.lastIndexOf(): Encuentra el índice de la última aparición de una subcadena especificada dentro de una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf

String.prototype.search(): Busca una subcadena especificada dentro de una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/search

String.prototype.includes(): Determina si una cadena contiene otra cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/includes

String.prototype.match(): Extrae coincidencias de un patrón de expresión regular de una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/match

String.prototype.matchAll(): Devuelve un iterador que produce todas las coincidencias de una expresión regular contra una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/matchAll

String.prototype.split(): Divide una cadena en un array de subcadenas según un separador especificado
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/split

String.prototype.slice(): Extrae una parte de una cadena y la devuelve como una nueva cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/slice

String.prototype.trim(): Elimina los caracteres de espacio en blanco de ambos extremos de una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/trim

String.prototype.replace(): Encuentra y reemplaza subcadenas dentro de una cadena
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/replace

Explorando arrays

Los arrays son valores no primitivos; son una colección de valores. Los valores pueden ser cualquier tipo de valor, incluidos otros arrays. Los arrays son mutables, lo que significa que puedes modificarlos y los cambios se reflejarán en el array original.

Los arrays están indexados en cero, lo que significa que el primer elemento está en el índice 0, el segundo elemento está en el índice 1, y así sucesivamente.

El método **Array.isArray()** determina si el valor pasado es un array:

```
const array = [1, 2, 3];

console.log(Array.isArray(array)); // true

const object = { name: "Kapumota" };
```

```
console.log(Array.isArray(object)); // false

console.log(typeof array); // object

console.log(typeof object); // object

console.log("¿son el objeto y el array del mismo tipo?", typeof(array) === typeof(object)); // true
```

Como los arrays son objetos, debes tener cuidado porque no se pueden comparar con el operador === o ==, porque comparará las referencias, no los valores:

```
const array1 = [1, 2, 3];

const array2 = [1, 2, 3];

console.log(array1 === array2); // false
```

Los arrays tienen una propiedad **length** que devuelve el número de elementos en el array y proporciona una forma fácil de iterar sobre el array.

Operaciones básicas

Creando un array

Hay muchas formas de crear un array en JavaScript. La más común es usar la notación literal de array, [], pero también puedes crear un array a partir de otros tipos de datos, como cuando divides una cadena, o usando el método **string.prototype.split()**. El siguiente es un ejemplo de creación de un array usando la notación literal de array:

```
const emptyArray = [];

const numbers = [1, 2, 3];

const strings = ["Hola", "Mundo"];

const mixed = [1, "Hola", true];
```

El método **Array.of()** crea una nueva instancia de array a partir de un número variable de argumentos, independientemente del número o tipo de argumentos:

```
const array = Array.of( 1, 2, 3 );
```

El método Array.from() crea una nueva instancia de array a partir de un objeto similar a un array o iterable:

```
console.log(Array.from('packt'));   // ['p', 'a', 'c', 'k', 't']
```

El operador de propagación, ..., se puede usar para crear un nuevo array a partir de un array existente o de una cadena:

```
console.log([...[1, 2, 3]]);    // [1, 2, 3]
```

```
console.log([...'pqrst']);      // ['p', 'q', 'r', 's', 't']
```

Además, puedes pasar una función de mapeo como segundo parámetro para que puedas realizar transformaciones cuando se crea el array:

```
console.log(Array.from([1, 2, 3], x => x + x)); // [2, 4, 6]
```

Accediendo a elementos

Puedes acceder a un elemento en un array usando el índice del elemento:

```
const fruits = ['banana', 'manzana', 'naranja'];

console.log(fruits[0]); // banana

console.log(fruits[1]); // manzana

console.log(fruits[2]); // naranja
```

Reemplazando elementos

Puedes reemplazar un elemento en un array usando el índice del elemento:

```
 const fruits = ['banana', 'manzana', 'naranja'];

fruits[0] = 'pera';

console.log(fruits); // ['pera', 'manzana', 'naranja']
```

Agregando elementos

Puedes agregar elementos a un array usando dos métodos principales:

Array.prototype.push():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/push

Array.prototype.unshift():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift

Siempre es preferible agregar nuevos elementos al final del array, porque agregar elementos al principio del array es una operación costosa. Esto se debe a que requiere reindexar todos los elementos en el array.

Eliminando elementos

Existen varios métodos que te permiten eliminar elementos del array:

Array.prototype.pop():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/pop

Array.prototype.shift():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/shift

Array.prototype.splice():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

Iterando sobre un array

Es posible iterar sobre un array utilizando un bucle for, pero también hay otras formas de iterar sobre un array.

JavaScript proporciona un gran soporte para la programación declarativa, lo que es especialmente útil cuando necesitas iterar sobre arrays. Así que resumamos las formas más comunes de iterar sobre un array.

La mayoría de estos métodos reciben una función como parámetro, y se ejecuta para cada elemento en el array. Dependiendo del método utilizado y los datos devueltos por la función se obtendrá un resultado u otro.

Otra cosa importante a recordar es que estos métodos se pueden encadenar, por lo que puedes usar un método tras otro y componer operaciones más complejas.

Iteración

Como los arrays pueden almacenar muchos elementos, es importante familiarizarse con los métodos que proporciona el array para iterar correctamente sobre ellos. Los más comunes son Array.prototype.map() y Array.prototype.forEach(). En ambos casos, iteraremos sobre el array, pero Array.prototype.map() devolverá directamente un nuevo array con la transformación aplicada. Veamos un ejemplo comparando ambos métodos:

```
const numbers = [1, 2, 3, 4, 5]

const mapTransformation = numbers.map(el => el * 10)

const forEachTransformation = []

numbers.forEach(el => {
  forEachTransformation.push(el * 10)
})

console.log(mapTransformation) // 10,20,30,40,50

console.log(forEachTransformation) // 10,20,30,40,50
```

Validación

Como los arrays pueden contener cualquier tipo de datos, es común necesitar validar si un array contiene un elemento específico o si todos los elementos en el array coinciden con una condición. Existen varios métodos, pero los más comunes son los siguientes:

Array.prototype.every():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/every

Array.prototype.some():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/some

Array.prototype.includes():
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/includes

Filtrado

Los arrays pueden almacenar mucha información, y es bastante común almacenar estructuras anidadas como grandes objetos. Hay muchas formas de hacer filtrado en JavaScript. La diferencia más importante entre ellas es cuál es tu salida esperada, ya que a veces estaremos interesados en un nuevo array con los valores filtrados, pero otras veces podríamos querer la posición (índice) de ciertos elementos en el array. El método más utilizado es Array.prototype.filter(), que se utiliza para generar un nuevo array con los elementos que cumplen con ciertos criterios. Veamos un ejemplo:

```
const numbers = [1, 2, 3, 4, 5]

const filteredNums = numbers.filter(el => el <= 3)

console.log(filteredNums) // [1, 2, 3]
```

Hay varios métodos que encontrarás relevantes en esta categoría:

Array.prototype.slice(): Devuelve una copia de una porción del array
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

Array.prototype.find(): Devuelve el valor del primer elemento en el array que satisface los criterios proporcionados
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/find

Array.prototype.findIndex(): Devuelve el índice del primer elemento en el array que satisface los criterios
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex

Utilidades

A veces necesitas aplanar un array de arrays. Puedes usar el método **array.flat()** para hacerlo:

```
const data = [1, [2, 3], [4, 5]];

const flatData = data.flat();

console.log(flatData); // [1, 2, 3, 4, 5]
```

Otro método común es el método **array.join()**, que se utiliza para unir todos los elementos en un array en una cadena:

```
const people = ['Mota', 'Chalo', 'Kapumota', 'Ire'];

console.log(people.join()); // Mota,Chalo,Kapumota,Ire

console.log(people.join(' + ')); // Mota + Chalo + Kapumota + Ire
```

Esto es muy útil cuando necesitas crear una cadena con una lista de elementos, por ejemplo, cuando necesitas crear una lista de elementos en HTML, XML, Markdown, y más:

```
const people = ['Mota', 'Chalo', 'Kapumota', 'Ire'];

const structuredPeople = people.map(person => '<li>${person}</li>\n');

console.log('
  <ul>
    ${structuredPeople.join('')}
  </ul>
')

// <ul>
//   <li>Mota</li>
//    ...
// </ul>
```

Mientras trabajas con datos, es muy común que necesitemos ordenar los elementos en el array. Esto se puede hacer con array.sort(). En general, es mejor si proporcionamos una función que especifique cómo ordenar correctamente los elementos solo para evitar resultados inesperados. Veamos un ejemplo:

```
const numbers = [7, 1,10, 3,15,20]

console.log(numbers.sort())

// [1, 10, 15, 20, 3, 7]

console.log(numbers.sort((a, b) => a - b))

// [1, 3, 7, 10, 15, 20]
```

Hay varios métodos que son bastante útiles y te encontrarás usando con mucha frecuencia:

Array.prototype.reverse(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse Array.prototype.concat(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat

Array.prototype.fill(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/fill

Array.prototype.reduce(): https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Desestructuración

ES6 introdujo una nueva sintaxis para desestructurar arrays y objetos. El lado izquierdo de una asignación es ahora un patrón para extraer valores de arrays y objetos. Este patrón se puede usar en declaraciones de variables, asignaciones, parámetros de funciones y valores de retorno de funciones. Además, puedes usar valores predeterminados (fail-soft) en caso de que el valor no esté presente en el array.

En el siguiente fragmento de código, podemos ver la forma clásica de hacer un fail-soft:

```
const list = [1, 2];

const a = list[0] || 0; // 1

const b = list[1] // 2

const c = list[2] || 4; // 4
```

El siguiente fragmento contiene el mismo código pero usando la desestructuración de ECMAScript 6:

```
const list = [1, 2];

const [ a = 0, b, c = 4 ] = list;
```

Como puedes ver, esta versión es más compacta. Actualmente, esta es la forma más popular de asignar valores predeterminados cuando es posible combinarla con la desestructuración.

Sets

ES6 introduce una nueva estructura de datos llamada Set. Un Set es una colección de valores, donde cada valor puede ocurrir solo una vez. Se puede usar para almacenar una colección de valores, pero no es un array, ya que no tiene índices. Es una solución bastante común para eliminar valores duplicados de un array, como podemos ver en el siguiente código:

```
let arr = [1,2,2,3,1,4,5,4,5]

let set = new Set(arr)

let uniques = Array.from(set)

console.log(uniques) // [1,2,3,4,5]
```

Puedes encontrar más información sobre métodos específicos de set en https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set.

Usando objetos en JavaScript

Los objetos son valores no primitivos; son una colección de propiedades. Una propiedad es un par clave-valor. La clave siempre es una cadena y el valor puede ser cualquier tipo de valor, incluidos otros objetos.

Operaciones básicas

Los objetos son la estructura más versátil en JavaScript. En esta parte, aprenderemos cómo crear objetos, cómo acceder y modificar sus propiedades y cómo iterar sobre las propiedades de un objeto.

Creando un objeto

Puedes crear un objeto utilizando la sintaxis literal de objetos, es decir, usando llaves:

```
const person = {}
```

También puedes crear un objeto y agregar directamente propiedades:

```
const person = {
   name: 'Chalo',
}
```

Puedes almacenar cualquier tipo de valor en un objeto, incluidos otros objetos o funciones (métodos):

```
const person = {
   name: 'Chalo',
   id: 1,
   favoriteColors: ['azul', 'verde'],
   address: {
      street: 'Main St',
      number: 1,
   },
   fullName: function() {
      return '${this.name} Doe'
   },
   sayHi: function() {
      console.log('¡Hola!')
   }
}
console.log(person.fullName()) // Kapu Kapu
person.sayHi() // ¡Hola!
```


```
console.log(person.address.street) // Main St

console.log(person.id) // 1

console.log(person.favoriteColors[0]) // azul
```

Creando y accediendo a propiedades

Puedes crear una nueva propiedad o sobrescribir las existentes en un objeto asignando un valor:

```
const person = {
   id: 12
}

person.name = 'Chalo'

console.log(person.name) // Chalo

person.id = 1

console.log(person.id) // 1
```

También puedes acceder a las propiedades de un objeto usando la notación de corchetes, lo que es útil cuando se utiliza acceso programático o cuando se utilizan claves con caracteres especiales o espacios en blanco:

```
const person = {
   id: 12
}

console.log(person['id']) // 12

const specialKey = 'nombre con espacios'

person[specialKey] = 'Chalo'

console.log(person[specialKey]) // Chalo
```

Eliminando propiedades

Puedes eliminar una propiedad de un objeto utilizando el operador **delete** o sobrescribirla con **undefined**:

```
const person = {
   id: 12,
   name: 'Chalo'
}

delete person.id
```

person.name = undefined

console.log(person.id) // undefined

console.log(person.name) // undefined

Iteración

Veamos cómo iterar sobre las propiedades de un objeto, y cómo obtener un array con las claves y valores de un objeto.

Este es nuestro objeto base:

```
const users = {
  admin: 'Chalo',
  moderator: 'Mota',
  user: 'Claudio',
}
```

Puedes iterar sobre las propiedades de un objeto usando el bucle **for...in**:

```
for (let role in users) {
  console.log('${users[role]} es el ${role}')}
// Chalo es la admin
// Mota es el moderator
// Claudio es el user
```

También puedes usar el método **Object.keys()** para obtener un array con las claves de un objeto, por lo que puedes usar métodos específicos de array para gestionar la iteración, como array.prototype.forEach():

```
const roles = Object.keys(users)
console.log(roles) // ['admin', 'moderator', 'user']
roles.forEach(role => {
  console.log(role) // admin
  console.log(users[role]) // Chalo
})
```

También puedes usar métodos adicionales que se han introducido recientemente en el lenguaje:

Object.values(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values

Object.entries(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries

Copia superficial (shallow) versus copia profunda (deep)

La forma en que JavaScript funciona significa que a veces no obtenemos la copia esperada de una variable. Veamos un ejemplo simple:

```
const name = "Chalo"

const number = 1

const array = [1, 2, 3]

const object = { id: 1, name: 'Chalo' }

// Copia

let nameCopy = name

let numberCopy = number

const arrayCopy = array

const objectCopy = object

// Modifica la copia

nameCopy = 'Mota'

numberCopy = 2

arrayCopy.push("elemento adicional")

objectCopy.name = 'Mota'

// Verifica el original

console.log(name) // Chalo

console.log(nameCopy) // Mota

console.log(number) // 1

console.log(numberCopy) // 2

console.log(array) // [1, 2, 3, "elemento adicional"]

console.log(arrayCopy) // [1, 2, 3, "elemento adicional"]

console.log(object) // { id: 1, name: 'Mota' }

console.log(objectCopy) // { id: 1, name: 'Mota' }
```

Este es un comportamiento bastante específico de JavaScript que frustra a muchos desarrolladores. ¿Cómo es posible que la variable original se modifique cuando modificamos la copia? La respuesta es que no estamos copiando la variable (copia profunda) en todos los escenarios; estamos copiando la referencia a la variable (copia superficial).

Solo los tipos primitivos (cadena, número, booleano, null, undefined y símbolo) se copian por valor; el resto se copian por referencia, por lo que en realidad obtienes una referencia a la variable original, como un atajo.

Esto te permite hacer algunas cosas interesantes, como crear referencias de atajo para objetos muy anidados:

```
const data = {item: {detail: { reference: {id: '123'} }}}

// hacer una referencia de atajo

const ref = data.item.detail.reference

ref.name = 'Chalo'

// verifica el original

console.log(data.item.detail.reference) // {id: '123', name: 'Chalo'}
```

Pero, como puedes ver, esto puede llevar a cambios en el objeto original. Esto puede ser un comportamiento inesperado si no tenemos claro cómo se copió la estructura original. Puede ser más difícil de detectar si estás utilizando estructuras anidadas.

Si deseas obtener una copia profunda de un objeto simple, puedes usar **Object.assign()** o el operador de propagación, ...:

```
const array = [1, 2, 3]

const object = { id: 1, name: 'Chalo' }

// Copia

const arrayCopy = [...array]

const objectCopy = Object.assign({}, object)

// Modifica la copia

arrayCopy.push("elemento adicional")

objectCopy.name = 'Mota'

// Verifica el original

console.log(array) // [1, 2, 3]

console.log(arrayCopy) // [1, 2, 3, "elemento adicional"]

console.log(object) // { id: 1, name: 'Chalo' }
```

```
console.log(objectCopy) // { id: 1, name: 'Mota' }
```

Pero los objetos anidados se copiarán por referencia, por lo que obtendrás el mismo comportamiento que antes:

```
const data = [{ 'a': 1 }, { 'b': 2 }];

const shallowCopy = [...data];

shallowCopy[0].a = 3;

console.log(data[0].a); // 3

console.log(shallowCopy[0].a); // 3
```

Una alternativa es utilizar una biblioteca especializada como **Lodash** (https://lodash.com/docs/4.17.15#cloneDeep) o transformarlo en JSON y digerir la estructura, pero esto tiene algunas limitaciones, como no poder copiar funciones o elementos que no están definidos en las especificaciones de JSON (https://datatracker.ietf.org/doc/html/rfc7159).

Mezclando objetos

Mezclar dos objetos se puede hacer con **Object.assign**, pero necesitas entender dos cosas:

El orden es importante, por lo que el primer elemento será sobrescrito por el siguiente elemento cuando compartan propiedades comunes.

Si los objetos son estructuras de datos complejas como objetos anidados o arrays, entonces el objeto final copiará las referencias (copia superficial)

Veamos un ejemplo:

```
const dst  = { quux: 0 }

const src1 = { foo: 1, bar: 2 }

const src2 = { foo: 3, baz: 4 }

Object.assign(dst, src1, src2)

console.log(dst) // {quux: 0, foo: 3, bar: 2, baz: 4}
```

Desestructuración

Desde ES6, JavaScript ha proporcionado asignación de desestructuración para objetos, lo cual es muy útil para extraer e incluir valores en objetos. Veamos un ejemplo con un objeto simple:

```
const name = "Chalo";

const age = 25;

const data = { item: "Lorem Ipsum", status: "OK" };
```

Si no usáramos la desestructuración, tendríamos que hacer algo como esto:

```
const user = {
  name: name,
  age: age,
  data: data,
};
const item = data.item;
const status = data.status;
```

Pero con la desestructuración, podemos hacerlo de una manera más concisa:

```
const user = { name, age, data };
const { item, status } = data;
```

Encadenamiento opcional (?.)

El operador de encadenamiento opcional es un nuevo operador introducido en ES2020. Te permite acceder a propiedades anidadas de un objeto sin preocuparte por si la propiedad existe o no. Antes del operador de encadenamiento opcional, tenías que verificar si la propiedad existía antes de acceder a ella. Esto era bastante tedioso para estructuras muy anidadas. Veamos un ejemplo práctico:

```
const user = {
  name: "Kapumota",
  address: {
    street: "Main Street",
  },
};
const otherUser = {
  name: "Chalo",
};
console.log(user.address?.street); // Main Street
console.log(otherUser.address?.street); // undefined
// sin encadenamiento opcional:
console.log(user.address.street); // Main Street
```

```
console.log(otherUser.address.street); // TypeError: Cannot read properties of undefined (reading 'street')
```

Explorando funciones

Las funciones son una de las estructuras más significativas en JavaScript. Existen ciertas características que las hacen diferentes de otros lenguajes de programación; por ejemplo pueden asignarse a una variable, pasarse como un argumento a otra función o devolverse de otra función.

Existen muchos conceptos avanzados relacionados con las funciones, pero en esta parte, solo veremos lo básico de las funciones en JavaScript. Comenzaremos con la declaración, ejecución y argumentos usando la palabra clave function. Luego, nos centraremos en las funciones flecha y closures.

Declaración

En esencia, una función es un bloque de código que se puede ejecutar cuando se llama. En JavaScript, podemos declarar una función utilizando la palabra clave **function.** La sintaxis es la siguiente:

```
function myFunction() {
  console.log("Este es el cuerpo de una función")
  // código a ejecutar
}
```

Ejecución

La función no se ejecuta cuando se declara; se ejecuta cuando se llama. Para llamar a una función, solo necesitamos escribir el nombre de la función seguido de paréntesis. Toma el siguiente ejemplo:

```
const myFunction = function() {
  console.log("Este es el cuerpo de una función")
  // código a ejecutar
}

myFunction() // Este es el cuerpo de una función
```

Funciones anónimas

Las funciones también se pueden declarar como una expresión de función. Esto se conoce como funciones anónimas. Un ejemplo simple es cuando pasamos la función como un argumento a otra función, como cuando usamos temporizadores: **setTimeout**, en este caso:

```
setTimeout(function() {
   console.log('1 segundo después')
```


```
}, 1000);
```

Valores de retorno

Una función puede devolver un valor utilizando la palabra clave return. Este valor se puede asignar a una variable o usar en otra función. Toma el siguiente ejemplo:

```
function isEven(number) {
  return number % 2 === 0
}
const result = isEven(2)
const otherResult = isEven(3)
console.log(result) // true
console.log(otherResult) // false
```

Argumentos

Las funciones pueden recibir argumentos; estos argumentos se pasan a la función cuando se llama. Toma el siguiente ejemplo:

```
function sayHi (name) {
  console.log('¡Hola ${name}!');
};
sayHi('Kapumota'); // ¡Hola Kapumota!
```

No necesitas especificar los argumentos; puedes usar el operador rest (...) para acceder a los argumentos. En este ejemplo, sumaremos todos los números pasados a la función:

```
function sum (...numbers) {
  console.log("Primer Número:", numbers[0])
  console.log("Último Número:", numbers[numbers.length - 1])
  let total = 0
  for (let number of numbers) {
    total += number
  }
  console.log("Total (SUMA):", total)
}
const result = sum(1, 2, 3, 4, 5)
```

// Primer Número: 1

// Último Número: 5

// Total (SUMA): 15

Funciones flecha

Una de las características más importantes introducidas en ES6 son las funciones flecha. Son una nueva sintaxis para escribir funciones en JavaScript, pero también introducen ciertos cambios que es importante tener en cuenta:

Las funciones flecha introducen una nueva sintaxis para escribir funciones

Las funciones flecha son siempre anónimas.

Sintaxis

Desde el principio de JavaScript, declaramos funciones usando la palabra clave function, como en el siguiente ejemplo:

```
const sampleFunction = function () { }

const sayHelloNow = function (name) {
  const now = new Date()
  console.log('¡Hola ${name}, en ${now}!')
}
```

La nueva sintaxis para escribir funciones flecha utiliza => y no utiliza la palabra clave function. El siguiente ejemplo es el mismo que el anterior, pero con la nueva sintaxis:

```
const sampleFunction = () => {}

const sayHelloNow = name => {
  const now = new Date()
  console.log('¡Hola ${name}, en ${now}!')
}
```

La nueva sintaxis tiene un retorno implícito, por lo que si deseas devolver un valor, puedes hacerlo sin usar la palabra clave return:

```
const alwaysTrue = () => true

const getData = (name, age) => ({ name: "Kapumota", age: 25 })
```

El ejemplo anterior se puede traducir a la sintaxis anterior de la siguiente manera:

```
const alwaysTrue = function () { return true }

const getData = function (name, age) {
  return { name: "Kapumota", age: 25 }
}
```

Las funciones flecha pueden recibir argumentos, pero si deseas recibir más de un argumento, debes usar paréntesis:

```
const sum = function (a, b) { return a + b }

// Traducción de función flecha

const sum = (a, b) => a + b
```

Cambios de comportamiento

Debido a que JavaScript tiene retrocompatibilidad con versiones anteriores, las funciones flecha introducen ciertos cambios en el comportamiento de las funciones. El más importante está relacionado con la palabra clave this.

Además, las funciones flecha no tienen una propiedad prototype, lo que significa que no se pueden usar como constructores o manejadores de métodos.

Importante

La gestión de this en JavaScript puede ser un poco confusa y es bastante avanzada para los objetivos de este curso. Si deseas aprender más al respecto, puedes leer la documentación de MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this.

Closures

Este es uno de los conceptos más populares e importantes en JavaScript, pero es un poco avanzado y no es fácil de entender.

Básicamente, un closure es una función devuelta por otra función. Aquí tenemos un ejemplo:

```
const outerFunction = function () {
  console.log("Esta es la función externa")
  const innerFunction = function () {
    console.log("Esta es la función interna")
  }
  return innerFunction
}
```

En este ejemplo, outerFunction devuelve innerFunction, por lo que podemos llamar a innerFunction después de llamar a outerFunction:

```
const innerFunction = outerFunction() // Esta es la función externa

innerFunction() // Esta es la función interna
```

Ahora, logremos el mismo resultado usando menos código al hacer ambas ejecuciones en la misma declaración:

```
// Ejecución en una sola línea

outerFunction()()
```

Pero, ¿cómo es esto útil?

Lo más importante de los closures es que pueden acceder e incluso modificar el alcance de la función principal (bloque de código y argumentos), incluso después de que la función principal haya devuelto. Veamos un ejemplo práctico:

```
const createCounter = (initialValue = 0) => {
  let counter = initialValue
  return (incrementalValue) => {
    counter += incrementalValue
    console.log(counter)
  }
}
```

En este ejemplo, agregamos los argumentos initialValue e incrementalValue a las funciones, y también definimos la variable counter para almacenar el valor actual del contador. En la práctica, podemos usar esta función para crear un contador que comience desde un valor específico, y luego podemos incrementarlo por un valor específico. No podemos acceder directamente a la variable counter porque solo vive en el ámbito dentro de la función y no fuera de ella, pero podemos usar el closure para acceder a ella e incluso manipular el valor:

```
const addToCounter = createCounter(10)

addToCounter(12) // 22

addToCounter(1)  // 23
```

En este ejemplo, vimos el uso básico de los closures, pero se pueden usar para muchas otras cosas. **Uno de los usos más comunes es crear abstracciones para gestionar servicios de terceros como bases de datos y APIs.**

Usaremos esta estructura cuando usemos MongoDB y Express.

Creando y gestionando clases

Las clases fueron introducidas en ES6. Son azúcar sintáctico sobre la herencia basada en prototipos. Históricamente, JavaScript no tenía clases formales como podemos esperar de los lenguajes típicos de Programación Orientada a Objetos (OOP).

En esta parte, aprenderemos cómo crear clases y cómo usarlas con ES6. Además, exploraremos cómo la herencia prototípica es una característica clave para mantener la retrocompatibilidad y extiende las funciones principales de JavaScript.

Creando una clase

Para crear una clase, necesitamos usar la palabra clave **class,** y luego podemos definir las propiedades predeterminadas de la clase utilizando el método constructor:

```
class Human{
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
const Chalo = new Human ("Chalo", 30);
console.log(Chalo.name); // Chalo
console.log(Chalo.age); // 30
```

En este ejemplo, creamos una clase llamada Human y luego creamos una instancia de la clase llamada Chalo. Podemos acceder a las propiedades de la clase usando la notación de punto.

Métodos de clase

Para definir un método en una clase, necesitamos usar una sintaxis similar a la que usamos para definir métodos en objetos:

```
class Human {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
```


```
    console.log('¡Hola, mi nombre es ${this.name}!');
  }
}
const Chalo = new Human ("Chalo", 30);
Chalo.sayHello(); // ¡Hola, mi nombre es Chalo!
```

En este ejemplo, definimos un método llamado sayHello en la clase Human, luego creamos una instancia de la clase, y llamamos al método.

Extendiendo clases

Podemos extender clases usando la palabra clave extends. Esto nos permitirá heredar las propiedades y métodos de la clase principal:

```
class Colleague extends Human {
  constructor(name, age, stack) {
    super(name, age);
    this.stack = stack;
    this.canCode = true;
  }
  code() {
    console.log('¡Puedo programar en ${this.stack}!');
  }
}
const Chalo = new Colleague ("Chalo", 30, ['JavaScript', 'React', 'MongoDB']);
console.log(Chalo.name); // Chalo
console.log(Chalo.canCode); // true
Chalo.sayHello(); // ¡Hola, mi nombre es Chalo!
Chalo.code(); // ¡Puedo programar en JavaScript, React y MongoDB!
```

En este ejemplo, creamos una clase llamada Colleague que extiende la clase Human, luego creamos una instancia de la clase, y llamamos a los métodos y propiedades heredados de ambas clases.

Métodos estáticos

Los métodos estáticos son métodos que se pueden llamar sin instanciar la clase. Se definen usando la palabra clave static:

```
class Car {
  constructor(brand) {
    this.brand = brand;
  }
  move() {
    console.log('¡El ${this.brand} se está moviendo!');
  }
  static speedLimits() {
    console.log("El límite de velocidad es de 120 km/h para autos nuevos");
  }
}
```

Ahora, podemos llamar al método speedLimits sin instanciar la clase:

```
Car.speedLimits(); // El límite de velocidad es de 120 km/h para autos nuevos
```

Getters y setters

Al igual que en otros lenguajes que admiten la programación orientada a objetos, puedes definir getters y setters usando las palabras clave get y set, respectivamente. Esto te permitirá acceder y modificar las propiedades de la instancia de una manera más clásica:

```
class Rectangle {
    constructor (width, height) {
        this._width  = width
        this._height = height
    }
    set width  (width)  { this._width = width            }
    get width  ()       { return this._width             }
    set height (height) { this._height = height          }
    get height ()       { return this._height            }
    get area   ()       { return this._width * this._height }
}
const shape = new Rectangle(5, 2)
```


```
console.log(shape.area) // 10

console.log(shape.height) // 2

console.log(shape.width) // 5

shape.height = 10

shape.width = 10

console.log(shape.area) // 100

console.log(shape.height) // 10

console.log(shape.width) // 10
```

Ejercicios

1. Escribe un script que imprima en la consola la versión de Node.js y una breve descripción de la versión actual de ECMAScript que está soportada por tu entorno.
2. Investiga los cambios introducidos en ECMAScript 2020 (ES11) y ECMAScript 2021 (ES12). Crea un proyecto que demuestre el uso de características como encadenamiento opcional, operador de coalescencia nula, BigInt, importación dinámica y campos privados en clases. Escribe un informe exhaustivo comparando el rendimiento y los casos de uso de estas características en aplicaciones del mundo real.
3. Escribe un script que declare una variable let y una constante const. Asigna valores a ambas y luego intenta reasignar un valor a la constante. Observa lo que sucede e imprime los resultados.
4. Configura un proyecto JavaScript con ESLint utilizando la guía de estilo de Airbnb. Crea varias violaciones intencionadas del estilo de código y luego resuélvelas utilizando las capacidades de corrección automática de ESLint. Escribe scripts para linting y formateo automático del código en cada commit usando hooks de Git.
5. Escribe una función simple sobre algún algoritmo que te resulte interesante. Comenta la función usando comentarios de una línea y múltiples líneas. Luego, usa JSDoc para documentar la función.
6. Crea un script que declare dos variables con números y aplique todos los operadores aritméticos (+, -, *, /, %, **). Guarda los resultados en nuevas variables y muestra los resultados en la consola
7. Crea una aplicación compleja que requiera gestión de estado (por ejemplo, una aplicación de tareas pendientes). Demuestra el uso correcto de let, const y el alcance de variables en diferentes funciones, closures y manejadores de eventos. Optimiza el uso de memoria asegurando que las variables estén correctamente delimitadas y se limpien cuando ya no sean necesarias.
8. Declara variables de diferentes tipos de datos (undefined, object, boolean, number, string, bigint, symbol). Usa typeof para imprimir el tipo de cada variable. Luego, manipula una cadena y un número utilizando métodos como toUpperCase(), toFixed(), etc.

9. Escribe un script que calcule la raíz cuadrada de un número, el valor máximo y mínimo de una lista de números, y un número aleatorio entre 1 y 100 usando el objeto Math.
10. Implementa una función que acepte varios tipos de datos y realice operaciones específicas para cada tipo (por ejemplo, operaciones aritméticas en números, manipulaciones de cadenas, transformaciones de arreglos). La función también debería manejar la coerción de tipos explícitamente y proporcionar advertencias cuando la conversión de tipos pueda llevar a resultados inesperados.
11. Crea un script que muestre la fecha y hora actual en diferentes formatos (fecha completa, solo la hora, etc.). Luego, suma 10 días a la fecha actual y muestra el resultado.
12. Escribe un script que compare dos números y use if, else if, else para determinar cuál es mayor. Usa operadores de comparación (>, <, >=, <=) y operadores de igualdad (===, !==).
13. Crea un mini-juego que use lógica condicional para determinar los resultados basados en las entradas del usuario. Incorpora varios operadores (==, ===, ??, !) y explora las implicaciones de la coerción de tipos y los valores nulos en la lógica del juego.
14. Crea un script que use un bucle for para imprimir los números del 1 al 10. Luego, crea un bucle while que haga lo mismo y otro que use do...while. Explica las diferencias entre los bucles.
15. Escribe un script que tome una cadena de texto y realice varias operaciones: dividirla en un array de palabras, ordenar el array alfabéticamente, y luego unir las palabras en una cadena nuevamente. Realiza operaciones similares en un array de números.
16. Construye una herramienta de procesamiento de texto que pueda realizar manipulaciones avanzadas de cadenas como encontrar palíndromos, contar la frecuencia de palabras y normalización de texto (por ejemplo, eliminar puntuación, convertir a minúsculas). Implementa estas características utilizando varios métodos de cadenas y expresiones regulares.
17. Implementa una pipeline de procesamiento de datos que tome un arreglo de objetos (por ejemplo, datos de usuarios) y realice operaciones como filtrado, mapeo, reducción y ordenamiento. Integra la desestructuración y métodos de arreglos para mejorar la eficiencia y legibilidad de la pipeline.
18. Crea un objeto que represente a un estudiante con propiedades como nombre, edad, cursos. Añade, elimina y modifica propiedades del objeto. Itera sobre las propiedades del objeto y muestra los valores en la consola.
19. Escribe una función que calcule el área de un rectángulo y otra que devuelva el nombre completo de una persona dada su primero y segundo nombre. Escribe ambas funciones como funciones tradicionales y luego conviértelas a funciones flecha.
20. Crea una clase Persona con propiedades nombre y edad, y un método que salude. Luego, crea una clase Estudiante que herede de Persona y añada la propiedad curso y un método para mostrar su curso.
21. Crea un módulo de JavaScript que manipule objetos anidados (por ejemplo, fusionar objetos anidados profundamente, crear copias profundas, acceso seguro utilizando encadenamiento opcional). Prueba el módulo con objetos complejos que contengan varios tipos de datos para garantizar su robustez.

22. Desarrolla una jerarquía de clases para una aplicación bancaria, incluyendo clases para Cuenta, CuentaAhorros, CuentaCorriente y Préstamo. Implementa métodos de clase, herencia y encapsulamiento. Añade métodos estáticos para gestionar tasas de interés y comisiones por transacciones. Prueba el comportamiento de los métodos getter y setter para validar la entrada del usuario.
23. Escribe un código que contenga varios errores comunes (variables no definidas, errores de sintaxis, etc.). Usa console.log para depurar y ESLint para encontrar y corregir los errores.

Ejercicios adicionales

Crea un servidor HTTP simple utilizando el módulo 'http' de Node.js. El servidor debe responder con un mensaje de "Hola, Mundo!" cuando un usuario visita la ruta raíz ('/'). Asegúrate de manejar correctamente las peticiones GET y POST.

Tareas adicionales:

- Modifica el servidor para que responda con un JSON que contenga un mensaje personalizado y la fecha/hora actual.
- Implementa el manejo de una ruta '/info' que devuelva información sobre el servidor (como el nombre del servidor, la versión de Node.js, etc.).

Crea un script en Node.js que lea un archivo de texto ('input.txt') y lo copie en otro archivo ('output.txt') utilizando el módulo 'fs'.

Tareas adicionales:

- Extiende el script para que transforme todo el texto a mayúsculas antes de guardarlo en 'output.txt'.
- Agrega funcionalidad para que el script cree un archivo nuevo si 'output.txt' ya existe, añadiendo un sufijo numérico (por ejemplo, 'output1.txt', 'output2.txt', etc.).

Divide tu código en varios módulos. Crea un módulo 'math.js' que exporte funciones básicas de matemáticas (suma, resta, multiplicación, división). Luego, utiliza este módulo en otro archivo para realizar cálculos en un array de números.

Tareas adicionales:

- Añade funciones para calcular el promedio y la mediana de un array de números.
- Implementa validaciones en las funciones para manejar errores como la división por cero.

Crea un archivo de configuración para tu aplicación en formato JSON ('config.json'). Este archivo debe incluir configuraciones como el puerto del servidor, nombre de la base de datos, y otras opciones que puedan cambiar según el entorno (desarrollo, producción).

Tareas adicionales:

- Implementa la lectura de variables de entorno utilizando 'process.env' para sobreescribir valores en el archivo de configuración.
- Crea diferentes archivos de configuración para distintos entornos (por ejemplo, 'config.development.json', 'config.production.json').

Desarrolla una herramienta de línea de comandos (CLI) en Node.js que acepte argumentos para realizar una operación matemática básica (suma, resta, multiplicación, división).

Tareas adicionales:

- Permite la entrada de múltiples números y realiza la operación seleccionada en todos ellos.
- Añade opciones para que el usuario pueda seleccionar la operación matemática a través de un flag (por ejemplo, '--operation=sum').

Utilizando el módulo 'net', crea un servidor TCP que acepte conexiones y permita a los clientes enviar mensajes. El servidor debe responder a cada mensaje recibido con un eco (enviar de vuelta lo que recibió).

Tareas adicionales:

- Modifica el servidor para manejar múltiples conexiones al mismo tiempo.
- Implementa un sistema de registro que guarde todos los mensajes recibidos en un archivo de log.

Crea un simulador de API en Node.js que lea datos desde un archivo JSON ('data.json') y los sirva a través de diferentes endpoints.

Tareas adicionales:

- Implementa un endpoint para agregar nuevos datos al archivo JSON y otro para eliminar datos.
- Añade validaciones para asegurarte de que no se agreguen datos duplicados y de que se envíe un mensaje de error si se intenta eliminar un dato que no existe.

Crea un script que intente leer un archivo que no existe. Maneja el error utilizando un bloque try...catch y muestra un mensaje de error adecuado.

Crea un objeto EventEmitter y registra dos oyentes para un evento llamado mensaje. Emite el evento mensaje y asegúrate de que ambos oyentes respondan.