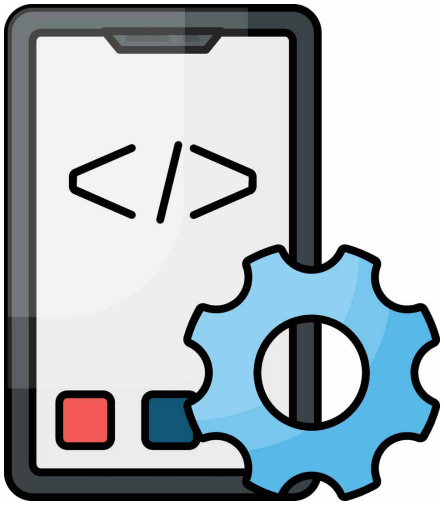
Programación con JavaScript II

Nuevos tipos y características

Introducción



La especificación ECMAScript es donde se precisan los comportamientos de JavaScript en un navegador. Una vez que se libera una nueva versión, los navegadores intentan cumplir con dicha especificación, sin embargo, no todos la consiguen. Por consiguiente, al momento de desarrollar cierta funcionalidad, esta no se ejecuta correctamente en uno o en otro navegador.

A lo largo de los años, el lenguaje JavaScript ha sufrido algunas modificaciones. No obstante, con la versión del año 2015 se marca un cambio dentro del mundo de JavaScript mediante variaciones que lo transformaron en un verdadero lenguaje moderno.

Por lo regular, cuando un desarrollador decide comenzar a programar en JavaScript, aprende sobre JavaScript 5, que es el que tiene más compatibilidad con la mayoría de los navegadores, pero no cuenta con las últimas novedades del lenguaje.

En este tema vas a aprender cuáles son algunas de las novedades del lenguaje y los nuevos tipos de datos que se han creado a partir de ECMAScript 2015 o ECMAScript 6.

Explicación

Nuevas características en JavaScript

Según MDN (2022), con las nuevas versiones de ECMAScript, han ido apareciendo nuevos tipos de datos, estructuras y colecciones que permiten que JavaScript sea más potente y versátil en su uso. Si bien es cierto que las que se mencionarán a continuación no son las únicas, sí son las que más se utilizan dentro de la industria del software.

- ✔ Symbols

Los símbolos se introdujeron en ES6 para que sirvieran como nombres de propiedades sin cadenas. Para comprender los símbolos, es necesario saber que el tipo de objeto fundamental de JavaScript es una colección desordenada de propiedades, donde cada propiedad tiene un nombre y un valor.

Los nombres de las propiedades suelen ser cadenas (hasta antes de ES6 eran exclusivamente de este tipo). Sin embargo, en ES6 y versiones posteriores, los símbolos también pueden servir para este propósito:

```
let strNombre = "string name"; // Una cadena para usar como nombre de propiedad
let symNombre = Symbol("propname"); // Un Símbolo para usar como nombre de propiedad
typeof strNombre // => "string": strNombre es una cadena
typeof symNombre // => "symbol": symNombre es un símbolo
let o = {}; // Crea un nuevo objeto
o[strNombre] = 1; // Define una propiedad con un strNombre cadena
o[symNombre] = 2; // Define una propiedad con symNombre símbolo
o[strNombre]; // => 1: accede a la propiedad strNombre
o[symNombre]; // => 2: accede a la propiedad símbolo
```

El tipo símbolo no tiene una sintaxis literal. Para obtener un valor de símbolo, llama a la función Symbol(). Esta función nunca devuelve el mismo valor dos veces, incluso cuando se llama con el mismo argumento.

```
let id=Symbol("id");
let id2=Symbol("id");

console.log(id === id2); // => false
```

Flanagan (2020) explica que, si se llama a Symbol() para obtener un valor de símbolo, se puede usar ese valor de forma segura como un nombre de propiedad para agregar una nueva propiedad a un objeto y no necesitas preocuparte de que puedas estar sobrescribiendo una propiedad existente con el mismo valor y nombre. De manera similar, si usan nombres de propiedad simbólicos y no comparte esos símbolos, puedes estar seguro de que otros módulos de código en el programa no sobrescribirán accidentalmente sus propiedades.

Explicación

En la práctica, los símbolos sirven como mecanismo de extensión del lenguaje. Cuando ES6 introdujo el bucle for/of y los objetos iterables, se necesitaba definir un método estándar que las clases pudieran implementar para volverse iterables, pero la estandarización de cualquier nombre de cadena en particular para este método de iteración habría roto el código existente, por lo que se utilizó un nombre simbólico en su lugar.

La función Symbol() toma un argumento de cadena opcional y devuelve un valor de símbolo único. Si se proporciona un argumento de cadena, esa cadena se incluirá en la salida del método toString() de Symbol. Ten en cuenta, sin embargo, que llamar a Symbol() dos veces con la misma cadena produce dos valores de Symbol completamente diferentes.

```
let s = Symbol("sym_x");
s.toString() // => "Symbol(sym_x)"
```

El único método interesante de instancias de Symbol es toString(), pero hay otras dos funciones relacionadas con los símbolos que debes conocer. Algunas veces, cuando se usan símbolos, se desea mantenerlos privados en su propio código para tener la garantía de que sus propiedades nunca entrarán en conflicto con las propiedades utilizadas por otro código. Sin embargo, en otras ocasiones, es posible que se desee definir un valor de símbolo y compartirlo ampliamente con otro código. Este sería el caso, por ejemplo, si se estuviera definiendo algún tipo de extensión en la que pudiera participar otro código.

Para atender este último caso de uso, JavaScript define un registro de símbolo global. La función Symbol.for() toma como argumento una cadena y devuelve un valor de símbolo que está asociado con la cadena que se pasó como argumento, en caso de que ningún símbolo esté asociado con esa cadena, se crea y se devuelve uno nuevo. Entonces, se puede resumir que la función Symbol.for() es completamente diferente a la función Symbol(), Symbol() nunca devuelve el mismo valor dos veces, mientras que Symbol.for() siempre devuelve el mismo valor cuando se llama con la misma cadena.

La cadena pasada a Symbol.for() aparece en la salida de toString() para el símbolo devuelto y también se puede recuperar llamando a Symbol.keyFor() ese mismo símbolo devuelto.

```
let s = Symbol.for("compartido");
let t = Symbol.for("compartido");
s === t // => true
s.toString() // => "Symbol(compartido)"
Symbol.keyFor(t) // => "compartido"
```

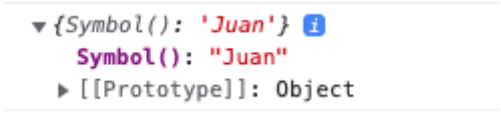
Explicación

Una buena regla de sintaxis es declararla como constante y con mayúsculas:

```
const NOMBRE=Symbol();
const persona={
[NOMBRE]:"Juan"
}

console.log(persona)
```

Observa en este ejemplo que, en la salida de este código, la consola informa que el objeto tiene una propiedad de tipo símbolo, pero no se presenta el nombre de la propiedad. Esto permitirá mantener ciertos valores como internos.



- Set

Set (conjunto) es una colección de valores, como lo es una matriz (arreglo). Sin embargo, a diferencia de las matrices, los conjuntos no están ordenados ni indexados. La principal característica de este tipo de estructuras es que no permiten duplicados: un valor es miembro de un conjunto o no es miembro, no es posible preguntar cuántas veces aparece un valor en un conjunto.

Se requiere del constructor Set() para crear un objeto Set:

```
let s = new Set(); // Un conjunto nuevo y vacío
let t = new Set([1, s]); // Un nuevo conjunto con dos miembros
```

El argumento del constructor Set() no necesita ser una matriz, se permite cualquier objeto iterable (incluidos otros objetos Set):

```
let t = new Set(s); // Un nuevo conjunto que copia los elementos de s.
let unico = new Set("Mississippi"); // 4 elementos: "M", "i","s" y "p"
```

Explicación

La propiedad de tamaño de Set es como la propiedad de longitud de un arreglo, indica cuántos valores contiene el conjunto:

```
unico.size // => 4
```

Kantor (s.f.) menciona que no es necesario inicializar los conjuntos al crearlos, puedes agregar y eliminar elementos en cualquier momento con add(), delete() y clear(). Recuerda que los conjuntos no pueden contener duplicados, por lo que agregar un valor a un conjunto cuando ya contiene ese valor no tiene efecto.

```
let s = new Set(); // Empezar vacío
s.size // => 0
s.add(1); // Agregar un número
s.size // => 1; ahora el conjunto tiene un miembro
s.add(1); // Agregar el mismo número de nuevo
s.size // => 1; el tamaño no cambia
s.add(true); // Agrega otro valor; tenga en cuenta que está bien mezclar tipos
s.size // => 2
s.add([1,2,3]); // Agregar un valor de matriz
console.log(s);
s.size // => 3; se agregó la matriz, no sus elementos
s.delete(1) // => verdadero: elemento 1 eliminado con éxito
s.size // => 2: el tamaño vuelve a ser 2
s.delete("test") // => falso: "test" no era miembro, la eliminación falló
s.delete(true) // => true: eliminación exitosa
s.delete([1,2,3]) // => falso: la matriz en el conjunto es diferente
s.size // => 1: todavía hay una matriz en el conjunto
s.clear(); // Eliminar todo del conjunto
s.size // => 0
```

- ✔ Maps

Un objeto Map representa un conjunto de valores conocidos como claves, donde cada clave tiene otro valor asociado (o “asignado a”). En cierto sentido, un mapa es como una matriz, pero en lugar de usar un conjunto de números enteros secuenciales como claves, los mapas te permiten usar valores arbitrarios como "índices". Al igual que las matrices, los mapas son rápidos: buscar el valor asociado con una clave será rápido (aunque no tan rápido como indexar una matriz), sin importar cuán grande sea el mapa.

Explicación

Crear un nuevo mapa con el constructor Map():

```
let m = new Map();  // Crea un nuevo mapa vacío
let n = new Map([ // Un nuevo mapa inicializado con claves de cadena asignadas a números
["one", 1],
["two", 2]
]);
```

El argumento opcional del constructor Map() debe ser un objeto iterable que produzca matrices de dos elementos [clave, valor]. En la práctica, esto significa que, si se desea inicializar un mapa cuando se crea, normalmente se escribirán las claves deseadas y los valores asociados como una matriz de matrices. Pero también se puede usar el constructor Map() para copiar otros mapas o para copiar los nombres y valores de propiedades de un objeto existente.

```
let copiar = new Map(n); // Un nuevo mapa con las mismas claves y valores como mapa n
let o = {x: 1, y: 2}; // Un objeto con dos propiedades
let p = new Map(Object.entries(o)); // Igual que new Map([["x",1], ["y", 2]])
```

Una vez creado un objeto Map, se puede consultar el valor asociado con una clave dada con get() y se puede agregar un nuevo par clave/valor con set(). Sin embargo, un mapa es un conjunto de claves, cada una de las cuales tiene un valor asociado. Esto no es lo mismo que un conjunto de pares clave/valor, si llama a set() con una clave que ya existe en el mapa, cambiará el valor asociado con esa clave, no agregará una nueva asignación de clave/valor. Además de get() y set(), la clase Map también define métodos que son como los métodos Set: se emplea has() para verificar si un mapa incluye la clave especificada; delete() para eliminar una clave (y su valor asociado) del mapa; clear() para eliminar todos los pares clave/valor del mapa; y se puede utilizar la propiedad size para averiguar cuántas claves contiene un mapa.

Explicación

Por lo tanto, si queremos llamar a las funciones, se deberá hacerlo desde callback:

```
let m = new Map(); // Empezar con un mapa vacío
m.size // => 0: los mapas vacíos no tienen claves
m.set("uno", 1); // Asignar la clave "uno" al valor 1
m.set("dos", 2); // Y la clave "dos" al valor 2.
m.size // => 2: el mapa ahora tiene dos claves
m.get("dos") // => 2: devuelve el valor asociado con la clave "dos"
m.get("tres") // => indefinido: esta clave no está en el conjunto
m.set("uno", true); // Cambiar el valor asociado con una clave existente
m.size // => 2: el tamaño no cambia
m.has("uno") // => verdadero: el mapa tiene una clave "uno"
m.has(true) // => false: el mapa no tiene clave true
m.delete("one") // => verdadero: la clave existía y la eliminación se realizó correctamente
m.size // => 1
m.delete("tres") // => falso: no se pudo eliminar una clave inexistente
m.clear(); // Elimina todas las claves y valores del mapa
```

- ✔ Iteradores y generadores

Los objetos iterables y sus iteradores asociados son una característica de ES6 que permite que las matrices, al igual que las cadenas y los objetos Set y Map, puedan ser recorridas. Esto significa que el contenido de estas estructuras de datos se puede iterar con el bucle for/of. Por ejemplo, si se desea iterar el siguiente objeto:

```
let rango = {
from: 1,
to: 5
};
```

Y se requiere que funcione de la siguiente manera: ***for (let num of range)*** y regrese ***num=1,2,3,4,5,*** es necesario agregarle un método llamado Symbol.iterator (símbolo especial utilizado para este tipo de casos). Por tanto, la implementación en código sería de la siguiente manera:

Explicación

```
// 1. Una llamada a for..of inicializa una llamada al método Symbol.iterator
rango[Symbol.iterator] = function() {
// ... devuelve el objeto iterador:
// 2. En adelante, for..of trabaja solo con el objeto iterador debajo, pidiéndole los siguientes valores
return {
current: this.from,
last: this.to,
// 3. next() es llamado en cada iteración por el bucle for..of
next() {
// 4. Debe devolver el valor como un objeto {done: Boolean, value : any}
if (this.current <= this.last) {
return { done: false, value: this.current++ };
} else {
return { done: true }; //Cuando regresa true, significa que el bucle ha finalizado
}
}
};
};
//para utilizarlo, como se tenía pensado
for (let num of rango) {
console.log(num); // 1, luego 2, 3, 4, 5
}
```

Un generador es un tipo de iterador definido con la nueva y potente sintaxis de ES6, el cual es particularmente útil cuando los valores a iterar no son los elementos de una estructura de datos, sino el resultado de un cálculo.

Para crear un generador, primero se debe definir una función de generador, que se conoce como función generadora. Esta es sintácticamente como una función JavaScript normal, pero se define con la palabra clave function* en lugar de function. (En teoría, esta no es una palabra clave nueva, solo un * después de la palabra clave y antes del nombre de la función).

Cuando se invoca una función generadora, en realidad no se ejecuta el cuerpo de la función, sino que devuelve un objeto generador, el cual es un iterador. Llamar a su método next() hace que el cuerpo de la función generadora se ejecute desde el principio (o cualquiera que sea su posición actual) hasta que llega a una declaración de rendimiento. Yield es nuevo en ES6 y es algo así como una declaración de devolución, el valor de esta instrucción yield se convierte en el valor devuelto por la llamada next() en el iterador. El siguiente ejemplo lo aclara:

Explicación

```
// Una función generadora que produce el conjunto de un dígito (base-10) primos.
function* primosUnDigito() { // Invocar esta función no ejecuta el código
yield 2; // pero solo devuelve un objeto generador. Vocación
yield 3; // se ejecuta el método next() de ese generador
yield 5; // el código hasta que una declaración de rendimiento proporcione
yield 7; // el valor devuelto por el método next().
}


// Cuando invocamos la función generador, obtenemos un generador
let primos = primosUnDigito();
// Un generador es un objeto iterador que itera los valores obtenidos
primos.next().value // => 2
primos.next().value // => 3
primos.next().value // => 5
primos.next().value// => 7
primos.next().done // => true
// Los generadores tienen un método Symbol.iterator para hacerlos iterable
primos[Symbol.iterator]() // => números primos
// Podemos usar generadores como otros tipos iterables
//[...primosUnDigito()] // => [2,3,5,7]
let suma = 0;
for(let primos of primosUnDigito()) suma += primos;
suma // => 17
```

En este ejemplo, se utilizó una declaración de function* para definir un generador, sin embargo, al igual que las funciones regulares, también puedes definir generadores en forma expresada. Una vez más, simplemente colocando un asterisco después de la palabra clave function:

```
const seq = function* (from, to) {
for (let i = from; i <= to; i++) yield i;
};
[...seq(3, 5)] // => [3, 4, 5]
```

Explicación

Los generadores son más interesantes si generan los valores que producen haciendo algún tipo de cálculo. Aquí, por ejemplo, hay una función generadora que produce números de Fibonacci:

```
function* fibonacciSequence() {
let x = 0, y = 1;
for (; ;) {
yield y;
[x, y] = [y, x + y];
}
}

}
```

Ten en cuenta que la función generadora de **fibonacciSequence**() aquí tiene un bucle infinito y produce valores para siempre sin regresar. Si este generador se usa con el operador spread, se repetirá hasta que se agote la memoria y el programa se cuelgue. Sin embargo, con cuidado, es posible usarlo en un bucle for/of:

```
// Regresa el n-avo número de Fibonacci
function fibonacci(n) {
for (let f of fibonacciSequence()) {
if (n-- <= 0) return f;
}
}
fibonacci(20) // => 10946
```

Explicación

Este tipo de generador infinito se vuelve más útil con un generador **take**() como este:

```
// Entrega los primeros n elementos del objeto iterable especificado
function* take(n, iterable) {
let it = iterable[Symbol.iterator](); // Obtener iterador para objeto iterable
while (n-- > 0) { // Bucle n veces:
let next = it.next(); // Obtener el siguiente elemento del iterador.
if (next.done) return; // Si no hay más valores, regresa temprano
else yield next.value; // de lo contrario, entrega el valor
}
}
// Una matriz de los primeros 5 números de Fibonacci
[...take(5, fibonacciSequence())] // => [1, 1, 2, 3, 5]
```

- ✔ Proxy

La clase Proxy, disponible en ES6 y posteriores, es la característica de metaprogramación más poderosa de JavaScript. Esta permite escribir código que altera el comportamiento fundamental de los objetos de JavaScript. Lo que hace la clase Proxy es que te permite implementar esas operaciones fundamentales tú mismo y crear objetos que se comporten de maneras que no son posibles para los objetos ordinarios.

Cuando creas un objeto proxy, especificas otros dos objetos: el objeto de destino y el objeto de controlador:

```
let proxy = new Proxy(target, handler);
```

El objeto proxy resultante no tiene estado ni comportamiento propio. Cada vez que realiza una operación en él (leer una propiedad, escribir una propiedad, definir una nueva propiedad, buscar el prototipo, invocarlo como una función), envía esas operaciones al objeto de los controladores o al objeto de destino.

Las operaciones admitidas por los objetos proxy son las mismas que las definidas por la API Reflect. Supón que **p** es un objeto proxy y escribe delete **p.x**. La función **Reflect.deleteProperty**() tiene el mismo comportamiento que el operador de eliminación. Cuando usa el operador de eliminación para eliminar una propiedad de un objeto proxy, busca un método **deleteProperty**() en el objeto de los controladores. Si tal método existe, lo invoca; si no existe, entonces el objeto proxy realiza la eliminación de la propiedad en el objeto de destino.

Explicación

Los *proxies* funcionan de esta manera para todas las operaciones fundamentales: si existe un método apropiado en el objeto manejador, invoca ese método para realizar la operación (los nombres y las firmas de los métodos son los mismos que los de las funciones Reflect); y si ese método no existe en el objeto handler, entonces el proxy realiza la operación fundamental en el objeto de destino. Esto significa que un proxy puede obtener su comportamiento del objeto buscado o del objeto handler. Si el objeto controlador está vacío, entonces el proxy es esencialmente un envoltorio transparente alrededor del objeto de destino.

```
let t = { x: 1, y: 2 };
let p = new Proxy(t, {});
p.x // => 1
delete p.y // => true: elimina la propiedad y del proxy
t.y // => indefinido: esto también lo elimina en el objetivo
p.z = 3; // Definiendo una nueva propiedad en el proxy
t.z // => 3: define la propiedad en el destino
```

Este tipo de proxy de envoltorio transparente es equivalente al objeto de destino subyacente, lo que significa que realmente no hay razón para usarlo en lugar del objeto envuelto. Sin embargo, los envoltorios transparentes pueden ser útiles cuando se crean como "proxies revocables". En lugar de crear un proxy con el constructor Proxy(), se puede usar la función de fábrica Proxy.revocable(). Esta función devuelve un objeto que incluye un objeto proxy y también una función revoke(). Una vez que llamas a la función revoke(), el proxy deja de funcionar inmediatamente.

La sintaxis es la siguiente:

```
let {proxy, revoke} = Proxy.revocable(target, handler);
```

Como lo indica Kantor (s.f.), la invocación regresa un objeto con el proxy y la función revoke para deshabilitarlo. Este sería un ejemplo de ello:

```
let objeto = {
datos: "datos valiosos"
};
let {proxy, revoke} = Proxy.revocable(objeto, {});
// pasamos el proxy en lugar del objeto...
alert(proxy.datos); // datos valiosos
// luego en nuestro código
revoke();
// el proxy no funciona más (revocado)
alert(proxy.datos); // Erro
```

Explicación

Propiedades dinámicas de los objetos

Flanagan (2020) indica que a veces se necesita crear un objeto con una propiedad específica, pero el nombre de esa propiedad no es una constante de tiempo de compilación que se pueda escribir literalmente en su código fuente. En su lugar, el nombre de la propiedad que necesita se almacena en una variable o es el valor de retorno de una función que invoca, no puede usar un objeto literal básico para este tipo de propiedad. Por el contrario, debe crear un objeto y después agregar las propiedades deseadas como un paso adicional.

```
const NOMBRE_PROPIEDAD = "p1";
function nombrePropiedadDinamica() { return "p" + 2; }
let o = {};
o[NOMBRE_PROPIEDAD] = 1;
o[nombrePropiedadDinamica()] = 2;
```

Es mucho más sencillo configurar un objeto como este con una característica de ES6 conocida como propiedades dinámicas, la cual permite tomar los corchetes del código anterior y moverlos directamente al objeto literal.

```
const NOMBRE_PROPIEDAD = "p1";
function nombrePropiedadDinamica() { return "p" + 2; };
let p = {
[NOMBRE_PROPIEDAD]: 1,
[nombrePropiedadDinamica()]: 2
};
p.p1 + p.p2 // => 3
```

Con esta nueva sintaxis, los corchetes delimitan una expresión JavaScript arbitraria. Esa expresión se evalúa y el valor resultante (convertido en una cadena si es necesario) se usa como el nombre de la propiedad.

Una situación en la que podrías querer usar propiedades calculadas es cuando tienes una biblioteca de código JavaScript que espera que le pasen objetos con un conjunto particular de propiedades y los nombres de esas propiedades se definen como constantes en esa biblioteca. Si estás escribiendo código para crear los objetos que se pasarán a esa biblioteca, podrías codificar los nombres de las propiedades, pero correrías el riesgo de errores si escribes el nombre de la propiedad de forma incorrecta en cualquier lugar. Además, se correría el riesgo de problemas de discrepancia de versiones si una nueva versión de la biblioteca cambia los nombres de propiedad requeridos. En su lugar, puedes encontrar que hace que tu código sea más robusto para usar la sintaxis de propiedad calculada con las constantes de nombre de propiedad definidas por la biblioteca

Cierre

En este tema aprendiste el uso de algunos de los tipos de datos añadidos recientemente al estándar ECMAScript y que son de los más utilizados por los desarrolladores, así como algunas de las estructuras y características de la metaprogramación en JavaScript más poderosas que existen.

Gracias a estas herramientas, ahora estás preparado para abordar proyectos más complejos y para poder realizar algunas actividades de manera más simple que como lo hacías con la programación básica en JavaScript.

Referencias bibliográficas

- Flanagan, D. (2020). *JavaScript: The Definitive Guide* (7a ed.). Estados Unidos: O'Reilly Media, Inc.
- Kantor, I. (s.f.). *El lenguaje JavaScript*. Recuperado de https://es.javascript.info/js
- MDN. (2022). *Tipos de datos y estructuras en JavaScript*. Recuperado de
- https://developer.mozilla.org/es/docs/Web/JavaScript/Data_structures#objetos

Para saber más

Lecturas

- Haverbeke, M. (2018). *La vida secreta de los objetos.* Recuperado de https://eloquentjs-es.thedojo.mx/06_object.html
- Lenguaje JS. (s.f.). *Tipos de datos.* Recuperado de https://lenguajejs.com/javascript/fundamentos/tipos-de-datos/.

Videos

- Aprogramar. (2021, 25 de julio). Aprogramar con JavaScript - 98 Computed properties en JS 2021 [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=CaO8Mp1mtzQ
- La Cocina del Código. (2019, 19 de septiembre). 2. TIPOS DE DATOS PRIMITIVOS en JAVASCRIPT | JS en ESPAÑOL [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=cC65D2q5f8I
- Midulive. (2022, 18 de marzo). Funciones Generadoras en JavaScript. ✅ Con este vídeo las vas a entender y dominar. [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=wl6X2XFnNtY

Checkpoints

Asegúrate de:

- Identificar los diferentes tipos de datos utilizados en JavaScript.
- Distinguir entre los tipos de datos primitivos y objetos.
- Comprender la diferencia entre ECMAScript y JavaScript.

Requerimientos técnicos

- Computadora con acceso a Internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

- Leer detenidamente y comprender el material explicado en el tema 3.
- Practicar todos los ejemplos que se describen en el tema 3.
- Revisar cada uno de los recursos adicionales que se proponen en el tema 3.