TALLER DE JAVASCRIPT

APRENDIZ: JUAN DAVID FORONDA JARAMILLO

INSTRUCTOR: JOHN BECERRA VANEGAS

PROGRAMA: DESARROLLO DE SOFTWARE

FICHA:2696521

FECHA: 23/05/2024

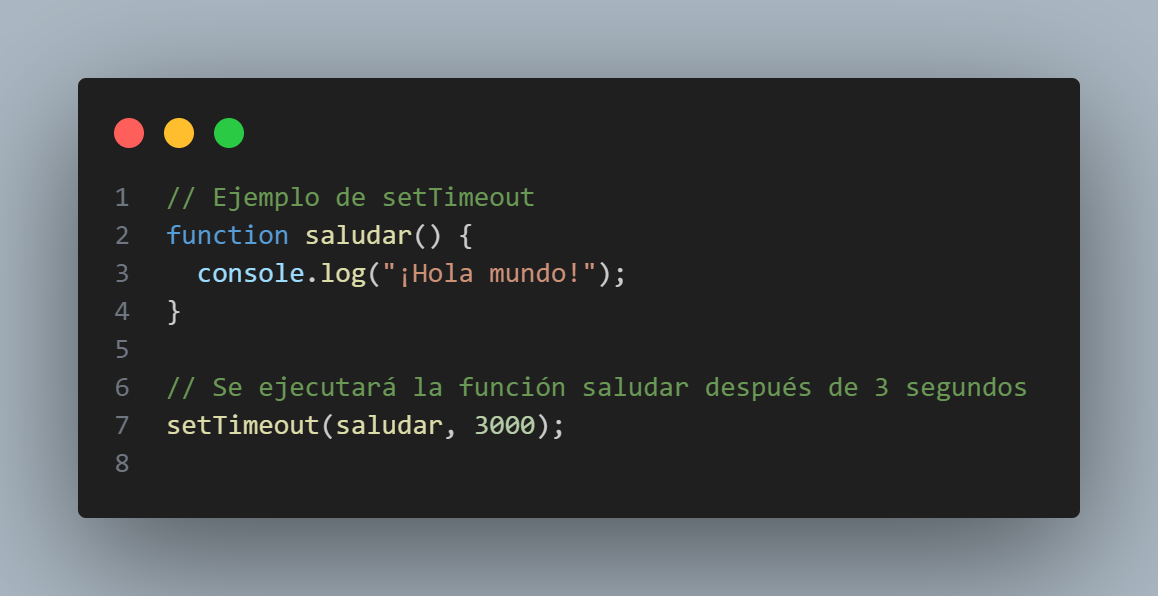
SENA CIMI – GIRON

**PROGRAMACION ASINCRONA:**

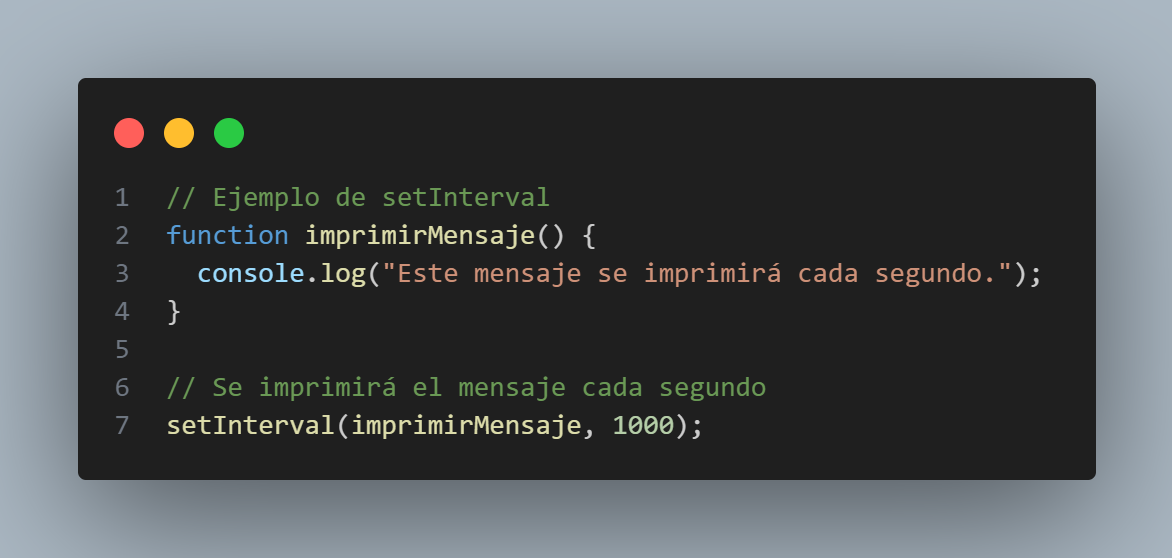
**Que son los temporizadores setimeout y setinterval (de un ejemplo)**

Los temporizadores setTimeout y setInterval son funciones en JavaScript que se utilizan para ejecutar código de forma asíncrona después de cierto período de tiempo. Aquí tienes una breve descripción de cada uno junto con un ejemplo de cómo se usan:

setTimeout: Esta función ejecuta una función específica una vez después de que haya transcurrido un cierto tiempo.



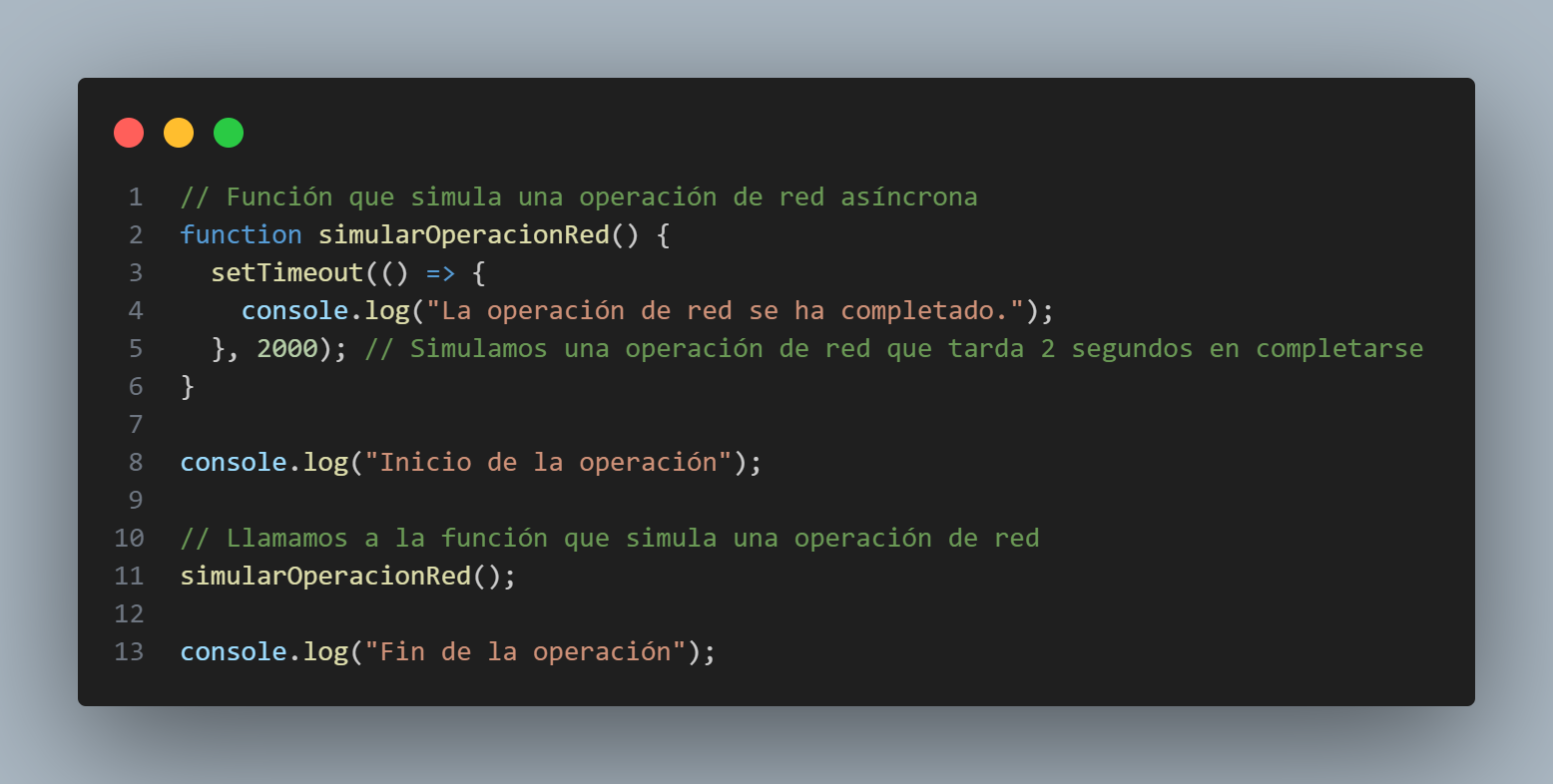
setInterval: Esta función ejecuta una función específica repetidamente cada cierto intervalo de tiempo.



**Que es la asincronía y el evento loop (de un ejemplo)**

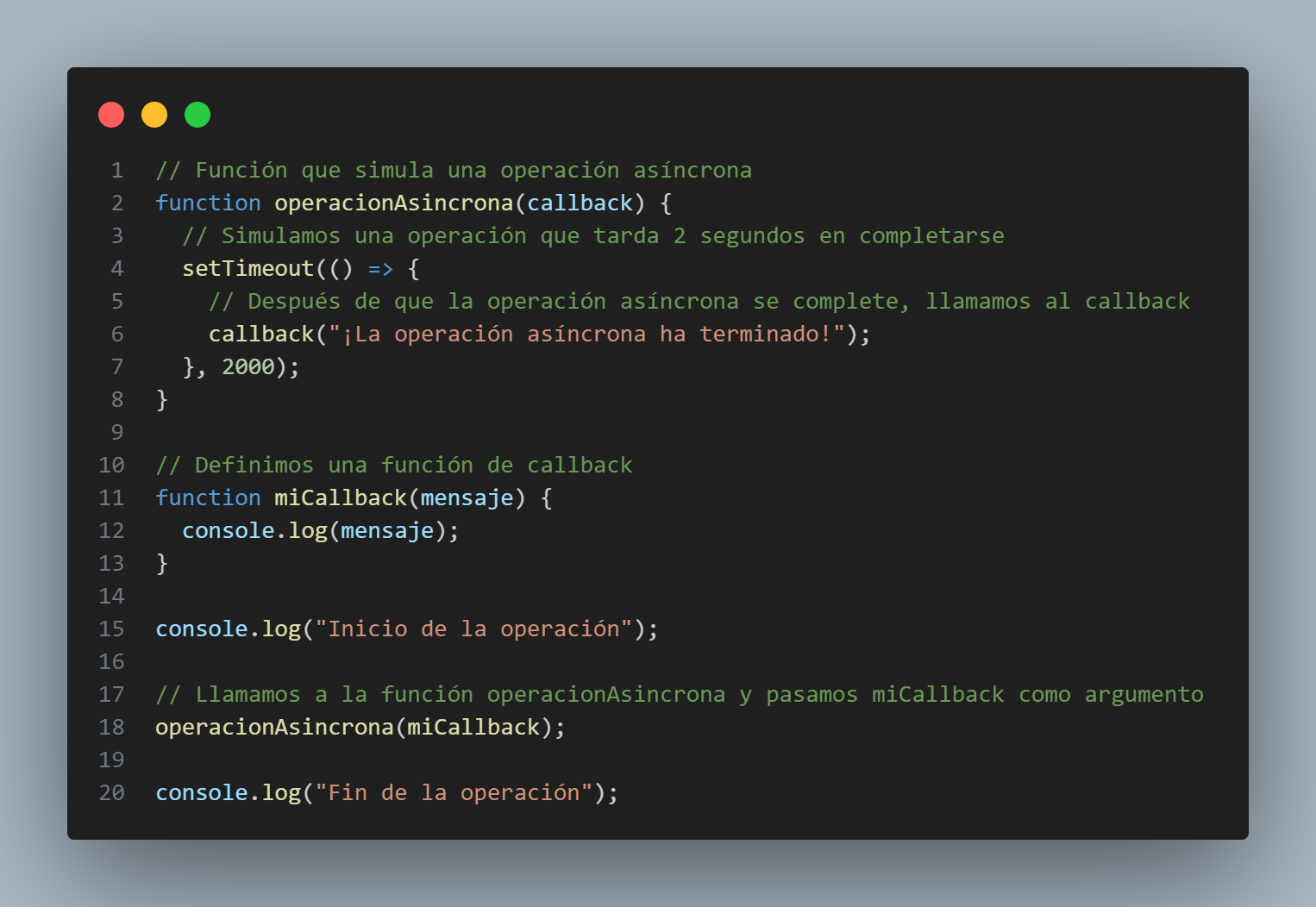
La asincronía en programación se refiere a la capacidad de ejecutar múltiples tareas simultáneamente sin esperar a que una tarea se complete antes de comenzar otra. Esto es crucial en entornos como JavaScript, donde las operaciones pueden ser bloqueantes (como las solicitudes de red o la lectura de archivos), y bloquear la ejecución del código puede llevar a una experiencia de usuario deficiente.

El Event Loop es un concepto fundamental en JavaScript que maneja la asincronía. Es un bucle que espera eventos y los maneja de manera asíncrona. En términos simples, permite que JavaScript espere y responda a eventos como clics de mouse, solicitudes de red y temporizadores sin detener la ejecución de otros scripts. Esto es lo que hace que JavaScript sea un lenguaje de programación no bloqueante.



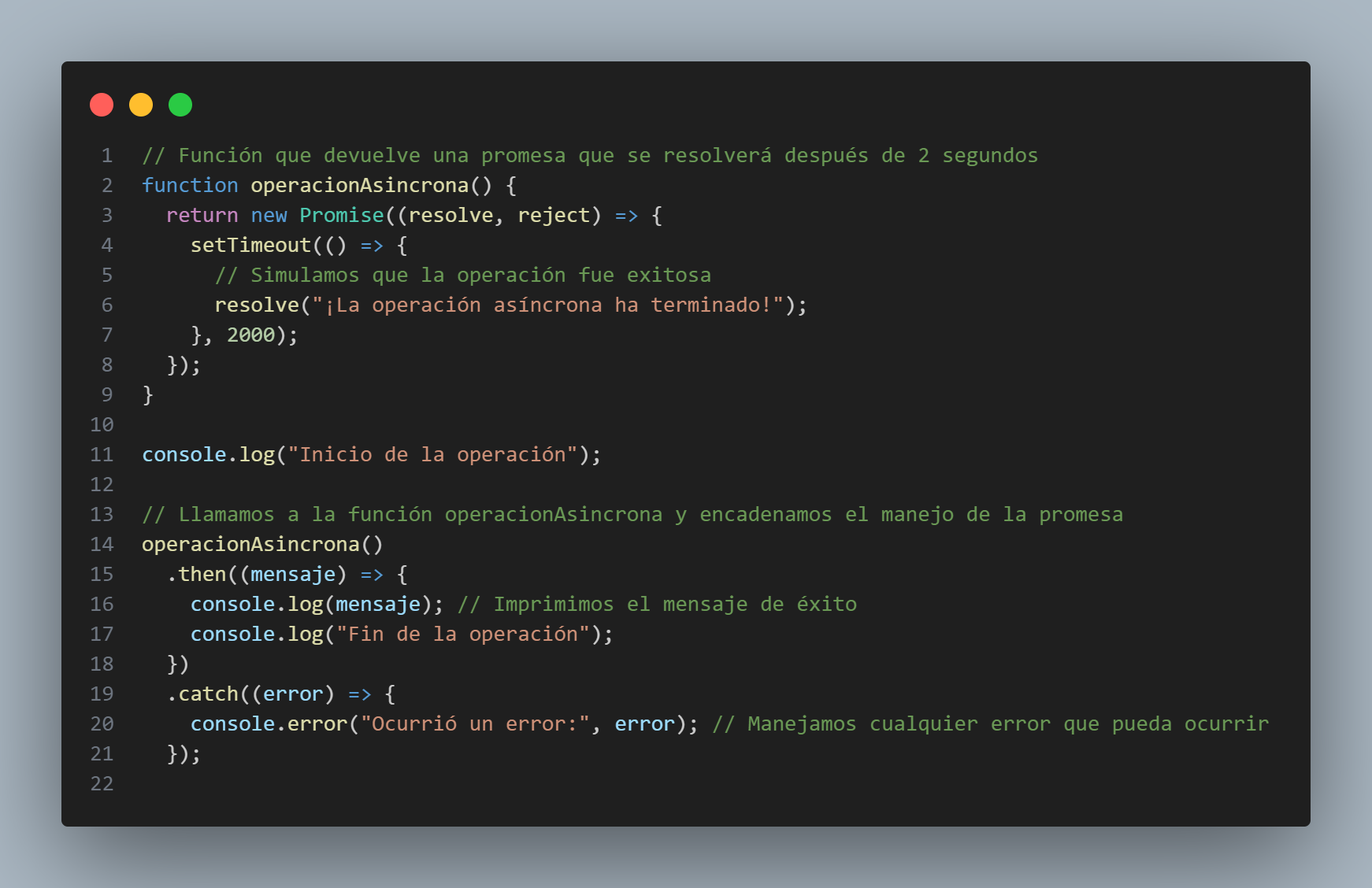
**o Que son los callbacks (de un ejemplo)**

Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se haya completado una operación asíncrona o algún otro tipo de evento. Son una forma común de manejar la asincronía en JavaScript, permitiendo que cierto código se ejecute solo después de que se haya completado una operación particular.



**o Que son las promesas (de un ejemplo)**

Las promesas son un mecanismo en JavaScript que nos permite manejar operaciones asíncronas de una manera más fácil y legible, evitando el exceso de anidamiento de callbacks (lo que comúnmente se conoce como "callback hell"). Una promesa representa un valor que puede estar disponible ahora, en el futuro o nunca.



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

* La promesa se cumple (promesa resuelta)
* La promesa no se cumple (promesa rechazada)
* La promesa se queda en un estado incierto indefinidamente (promesa pendiente)

Con estas sencillas bases, podemos entender el funcionamiento de una promesa en Javascript. Antes de empezar, también debemos tener claro que existen dos partes importantes de las promesas: como consumirlas (utilizar promesas) y como crearlas (preparar una función para que use promesas y se puedan consumir).

**o Que son las funciones asíncronas (async / await) (de un ejemplo)**

Tenemos dos partes cuando usamos async/await en nuestro código.

En primer lugar, tenemos la palabra clave async , que se pone delante de una declaración de función para convertirla en una función async.

Una función asíncrona es una función que sabe que es posible que se use la palabra clave await dentro de ella para invocar código asíncrono.

La palabra clave async se añade a las funciones para que devuelvan una promesa en lugar de un valor directamente.



**Cómo una función Async devuelve una promesa**

Este es uno de los rasgos de las funciones asíncronas — se garantiza que sus valores de retorno se convertirán en promesas. Para manejar los datos devueltos por una función async podemos usar una palabra clave then para obtener los datos.

****

**Cómo usar Async/Await con manejo de errores**

****

El try-catch anterior solamente manejará errores al obtener los datos, como una sintaxis incorrecta, nombres de dominio incorrectos, errores de red, etc.

Cuando quieras manejar un mensaje de error del código de respuesta de la API, puedes usar res.ok ( res es la variable en la que se almacena la respuesta). Te dará un Boolean con el valor verdadero si el código de respuesta está entre 200 y 209.

**Nuevos tipos y características de JavaScript:**

**Que son los Symbols (de un ejemplo de cada uno):**

**Tipo Symbol**

Según la especificación, solo dos de los tipos primitivos pueden servir como clave de propiedad de objetos:

string, o

symbol.

Si se usa otro tipo, como un número, este se autoconvertirá a string. Así, obj[1] es lo mismo que obj["1"], y obj[true] es lo mismo que obj["true"].

Hasta ahora solo estuvimos usando strings.

Ahora exploremos symbols y ver lo que pueden hacer por nosotros.

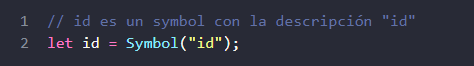
**Symbols:**

El valor de “Symbol” representa un identificador único.

Un valor de este tipo puede ser creado usando **Symbol():**

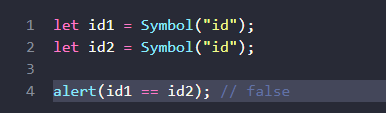


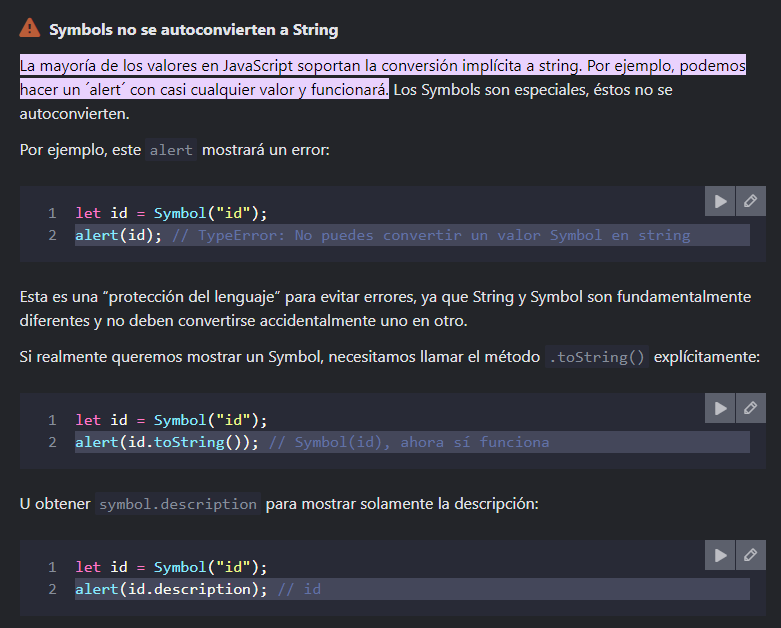
**Al crearlo, podemos agregarle una descripción (también llamada symbol name), que será útil en la depuración de código:**



Se garantiza que los símbolos son únicos. Aunque declaremos varios Symbols con la misma descripción, éstos tendrán valores distintos. La descripción es solamente una etiqueta que no afecta nada más.

Por ejemplo, aquí hay dos Symbols con la misma descripción… pero no son iguales:



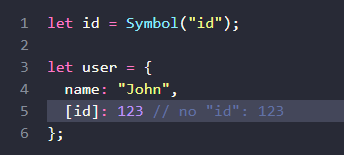


**CLASES DE SYMBOLS**

**Symbols en objetos literales**

Si queremos usar un Symbol en un objeto literal, debemos usar corchetes.

**Como se muestra a continuación:**



**Se hace así porque necesitamos que el valor de la variable id sea la clave, no el string “id”.**

**Symbols Globales**

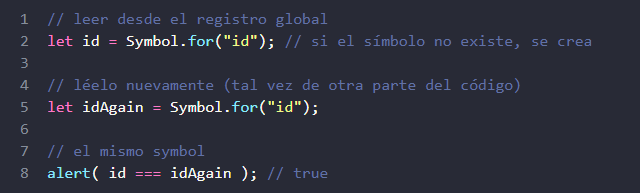
Como hemos visto, normalmente todos los Symbols son diferentes aunque tengan el mismo nombre. Pero algunas veces necesitamos que symbols con el mismo nombre sean la misma entidad.

Para lograr esto, existe un global symbol registry. Ahí podemos crear symbols y accesarlos después, lo cual nos garantiza que cada vez que se acceda a la clave con el mismo nombre, esta te devuelva exactamente el mismo symbol.

Para crear u accesar a un symbol en el registro global, usa Symbol.for(key).

Esta llamada revisa el registro global, y si existe un symbol descrito como key, lo retornará; de lo contrario creará un nuevo symbol Symbol(key) y lo almacenará en el registro con el key dado.

**Por ejemplo:**

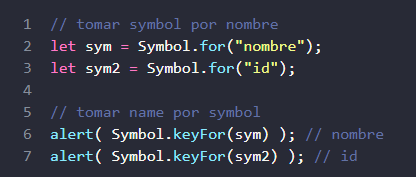


**Los Symbols dentro de este registro son llamados global symbols y están disponibles y al alcance de todo el código en la aplicación.**

**Symbol.keyFor**

Hemos visto que para los global symbols, Symbol.for(key) devuelve un symbol por su nombre. Para hacer lo opuesto, – devolver el nombre de un global symbol – podemos usar: Symbol.keyFor(sym).

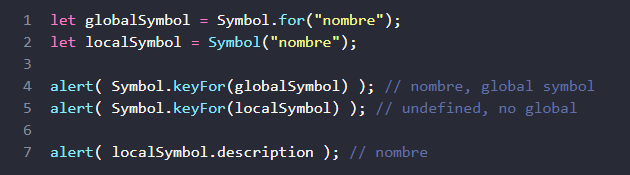
**Por ejemplo:**



El Symbol.keyFor utiliza internamente el registro “global symbol registry” para buscar la clave del symbol, por lo tanto, no funciona para los symbol que no están dentro del registro. Si el symbol no es global, no será capaz de encontrarlo y por lo tanto devolverá undefined.

Dicho esto, todo symbol tiene la propiedad description.

**Por ejemplo:**



**System symbols**

Existen varios symbols del sistema que JavaScript utiliza internamente, y que podemos usar para ajustar varios aspectos de nuestros objetos.

**Se encuentran listados en Well-known symbols :**

* Symbol.hasInstance
* Symbol.isConcatSpreadable
* Symbol.iterator
* Symbol.toPrimitive

**Por ejemplo, Symbol.toPrimitive nos permite describir el objeto para su conversión primitiva. Más adelante veremos su uso.**

**Resumen**

**Symbol es un tipo de dato primitivo para identificadores únicos.**

Symbols son creados al llamar Symbol() con una descripción opcional.

Symbols son siempre valores distintos aunque tengan el mismo nombre. Si queremos que symbols con el mismo nombre tengan el mismo valor, entonces debemos guardarlos en el registro global: Symbol.for(key) retornará un symbol (en caso de no existir, lo creará) con el key como su nombre. Todas las llamadas de Symbol.for con ese nombre retornarán siempre el mismo symbol.

**Symbols se utilizan principalmente en dos casos:**

**Propiedades de objeto “Ocultas”**

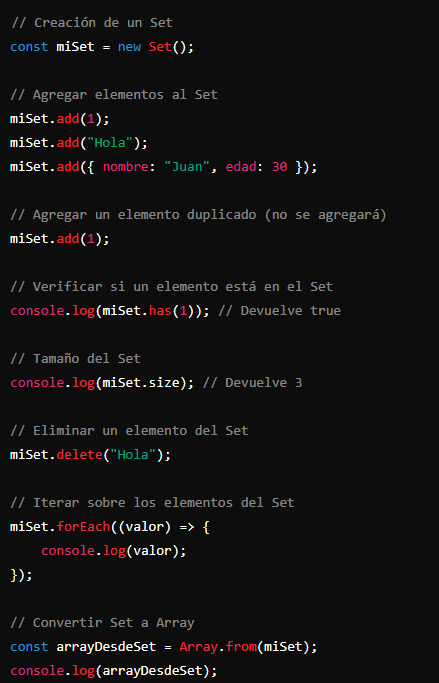
Si queremos agregar una propiedad a un objeto que “pertenece” a otro script u otra librería, podemos crear un symbol y usarlo como clave. Una clave symbol no aparecerá en los ciclos for..in, por lo que no podrá ser procesada accidentalmente junto con las demás propiedades. Tampoco puede ser accesada directamente, porque un script ajeno no tiene nuestro symbol. Por lo tanto la propiedad estará protegida contra uso y escritura accidentales.

Podemos “ocultar” ciertos valores dentro de un objeto que solo estarán disponibles dentro de ese script usando las claves de symbol.

Existen diversos symbols del sistema que utiliza Javascript, a los cuales podemos accesar por medio de Symbol.\*. Podemos usarlos para alterar algunos comportamientos. Por ejemplo, más adelante en el tutorial, usaremos Symbol.iterator para iterables, Symbol.toPrimitive para configurar object-to-primitive conversion.

**o Que son los sets (de un ejemplo):**

Los Sets son una estructura de datos en JavaScript que permite almacenar valores únicos de cualquier tipo, ya sean primitivos o referencias a objetos. Los valores en un Set están siempre en orden de inserción, y no se permiten duplicados.



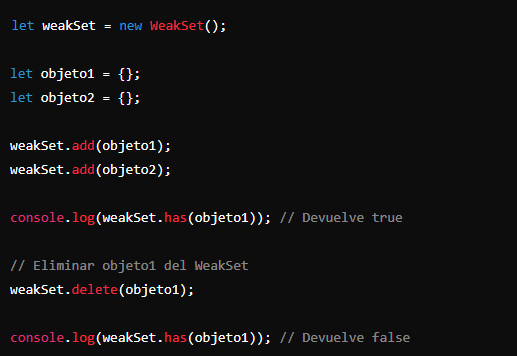
También podemos eliminar elementos de un Set con el método delete(), e iterar sobre los elementos utilizando el método forEach(). Finalmente, mostramos cómo convertir un Set en un Array utilizando Array.from(). Los Sets son útiles cuando necesitamos almacenar colecciones de valores únicos sin preocuparnos por el orden.

**o Que son los WeakSets y WeakMaps (de un ejemplo de cada uno)**

Los WeakSets y WeakMaps son estructuras de datos en JavaScript que son similares a los Sets y Maps respectivamente, pero con algunas diferencias clave en su comportamiento en cuanto a la gestión de la memoria.

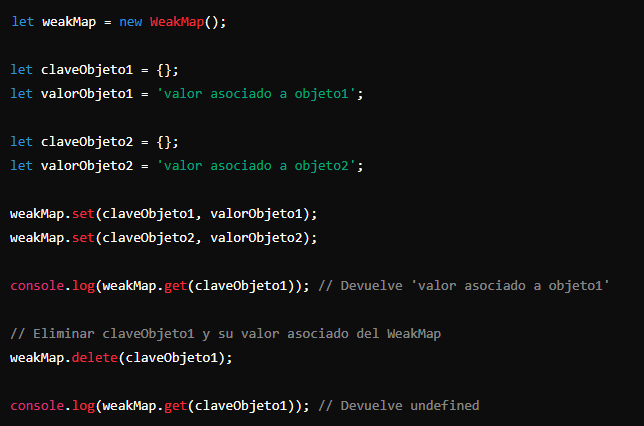
**WeakSet:**

Un WeakSet es una colección de objetos débilmente mantenidos. Esto significa que si no hay otras referencias al objeto almacenado en el WeakSet, el recolector de basura puede eliminar el objeto de la memoria. Los WeakSets solo pueden contener objetos, no aceptan valores primitivos.



**WeakMap:**

Un WeakMap es una colección de pares clave/valor en la que las claves son débilmente mantenidas. Al igual que los WeakSets, si no hay otras referencias a la clave, esta puede ser eliminada por el recolector de basura. Al igual que con los Mapas, las claves pueden ser de cualquier tipo, pero a diferencia de los Mapas, las claves en WeakMaps son débilmente mantenidas.

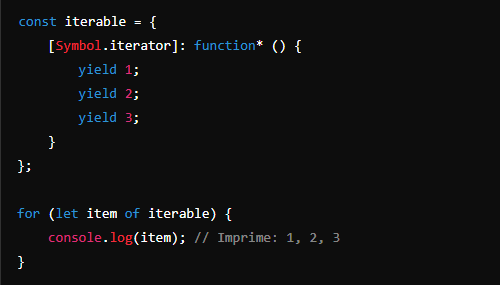


**o Que son los iterables y los iteradores (de un ejemplo)**

Los iterables y los iteradores son conceptos fundamentales en JavaScript que nos permiten recorrer secuencias de datos, como arreglos o cadenas, de una manera genérica y uniforme.

**Iterables:**

Un iterable es cualquier objeto que implementa el método Symbol.iterator, que devuelve un objeto iterador. Los iterables pueden ser recorridos mediante un bucle for...of o mediante la función Array.from().



**Iteradores:**

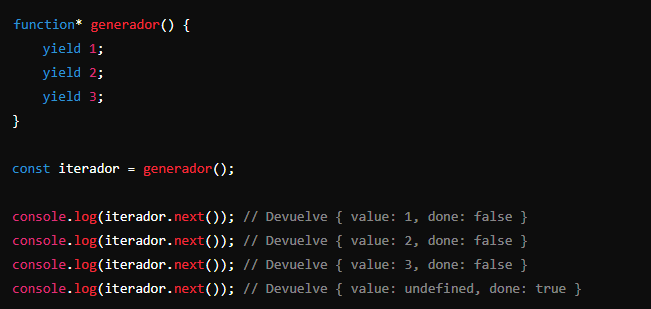
Un iterador es un objeto que proporciona un método next(), que devuelve un objeto con dos propiedades: value (el siguiente valor en la secuencia) y done (un booleano que indica si la iteración ha terminado).



**o Que son los generadores (de un ejemplo de cada uno)**

Los generadores son funciones especiales en JavaScript que permiten pausar y continuar la ejecución de una función de manera controlada. Se definen utilizando la palabra clave function\* y contienen uno o más yield, que es una expresión que devuelve un valor y pausa la ejecución del generador.

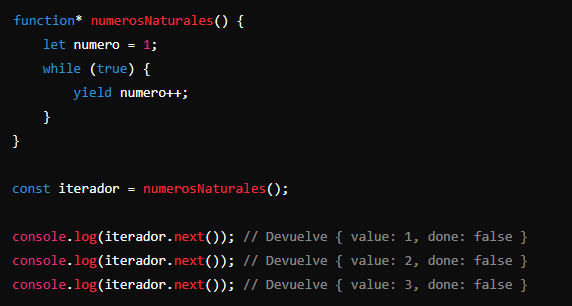
**EJEMPLO BASICO:**



En este ejemplo, generador es una función generadora que produce los valores 1, 2 y 3 cuando se llama a next() en el iterador. Después de la tercera llamada a next(), el generador termina y devuelve { value: undefined, done: true }.

**Existen tres tipos de generadores en JavaScript:**

1. Generadores básicos: Como el ejemplo anterior, donde se utilizan las palabras clave function\* y yield para definir un generador.
2. Generadores infinitos: Son generadores que pueden producir un número infinito de valores. Un ejemplo común es el generador que produce números naturales infinitos.



**o Que son los proxies (de un ejemplo)**

Un Proxy es un objeto que envuelve a otro objeto e intercepta operaciones básicas sobre él, como acceder, asignar y eliminar propiedades. El proxy es un aspecto crucial de JavaScript que permite a los desarrolladores escribir código más versátil y robusto.

**Sintaxis y propiedades de Proxy**

El Proxy de JavaScript es una capacidad que permite la creación de objetos capaces de modificar y personalizar las operaciones básicas realizadas sobre otros objetos.

**Crear un objeto proxy**

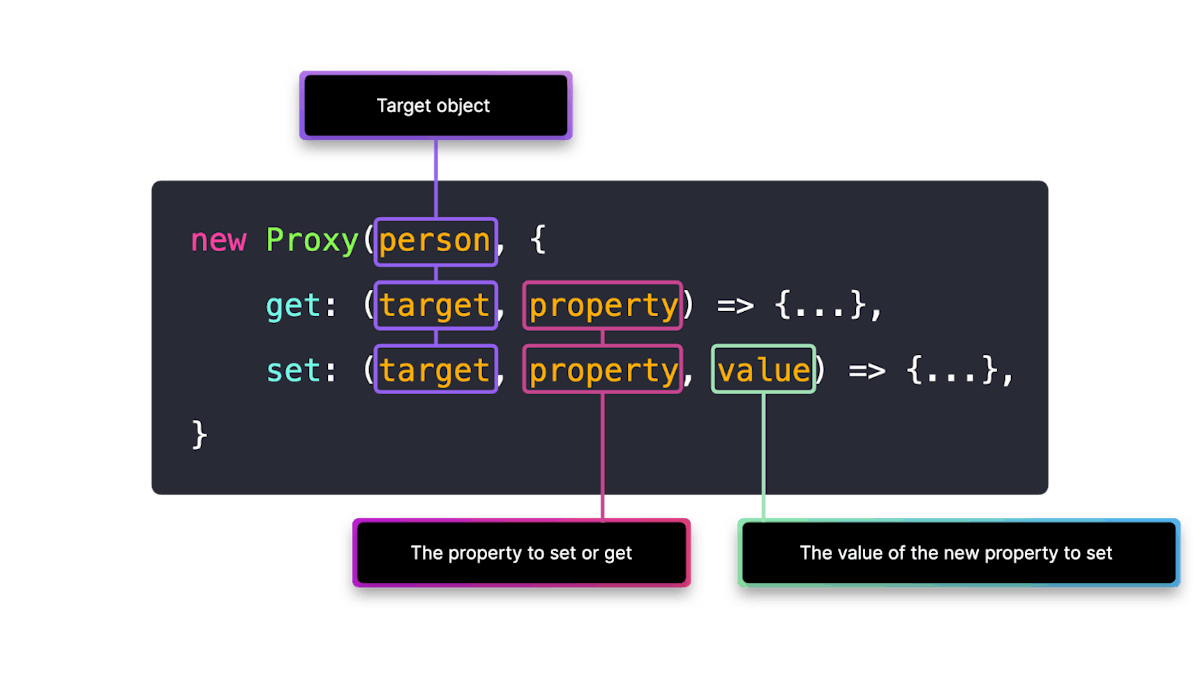
Para establecer un objeto Proxy, se necesitan dos componentes: un objeto de destino y un objeto controlador. El objeto objetivo es aquel en el que se van a interceptar las operaciones, mientras que el objeto controlador es responsable de mantener las trampas o métodos utilizados para capturar estas operaciones.

Aquí hay un ejemplo que demuestra cómo crear un objeto Proxy básico:



En este ejemplo, generamos un objeto de destino que tiene dos características: nombre y edad. También generamos un objeto controlador que tiene una trampa de obtención para capturar cualquier esfuerzo por leer una propiedad en el objeto de destino. Después de eso, producimos un objeto Proxy proporcionando los objetos de destino y controlador al constructor de Proxy. Por último, recuperamos la propiedad de nombre del objeto Proxy, que invoca la trampa get y envía un mensaje a la consola.

**Trampas y su comportamiento.**



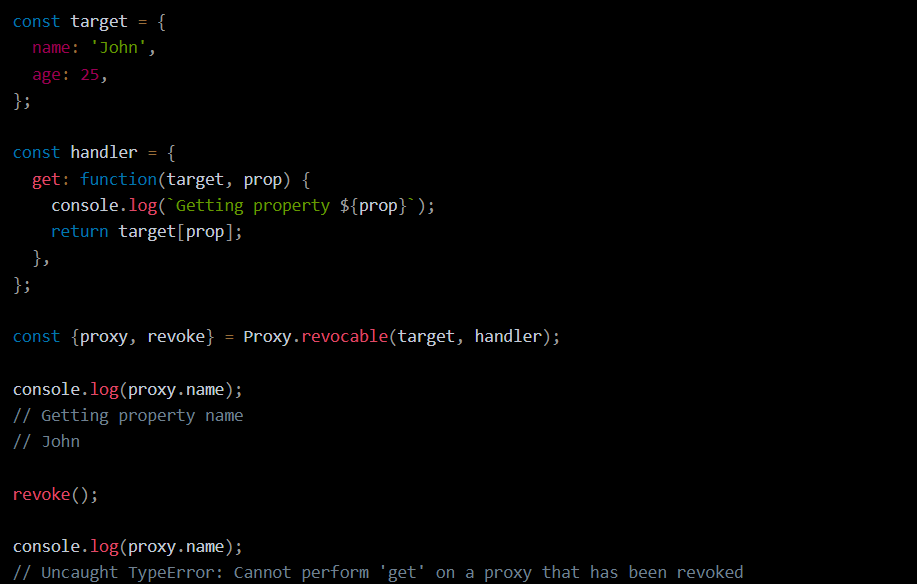
Las trampas son métodos que interceptan operaciones en el objeto de destino. Hay varias trampas que puede usar con un objeto Proxy, incluidas get, set, has, deleteProperty y más.

A continuación se ofrece una breve descripción de algunas de las trampas más utilizadas:

1. ***get*** : esta trampa intercepta los intentos de leer una propiedad en el objeto de destino. Se necesitan dos argumentos: el objeto de destino y la propiedad a la que se accede. La trampa devuelve el valor de la propiedad.
2. ***set*** : esta trampa captura cualquier esfuerzo para establecer una propiedad en el objeto de destino. Requiere tres parámetros: el objeto de destino en sí, la propiedad que se está estableciendo y el valor actualizado de esa propiedad. El mecanismo tiene la capacidad de alterar el valor que se está estableciendo, o puede generar un error para prohibir que se establezca el valor.
3. ***tiene*** : esta trampa intercepta los intentos de verificar si existe una propiedad en el objeto de destino. Se necesitan dos argumentos: el objeto de destino y la propiedad que se está verificando. La trampa devuelve un valor booleano que indica si la propiedad existe o no.
4. ***deleteProperty*** : esta trampa intercepta los intentos de eliminar una propiedad del objeto de destino. Se necesitan dos argumentos: el objeto de destino y la propiedad que se elimina. La captura puede eliminar la propiedad o generar un error para evitar que se elimine la propiedad.

**Poderes revocables**

Los objetos proxy poseen una característica fascinante que les permite invalidarse, lo que hace que sus trampas ya no intercepten operaciones en el objeto objetivo. Para construir un objeto Proxy que pueda invalidarse, utilice la Proxy.revocable()función.



En este ejemplo, creamos un objeto Proxy revocable usando el Proxy.revocable()método. Luego accedemos a la propiedad de nombre del objeto Proxy, que activa la trampa de obtención y registra un mensaje en la consola. Luego revocamos el objeto Proxy usando el revoke()método, lo que significa que cualquier intento adicional de acceder a las propiedades del objeto Proxy fallará.

**Herencia con proxy**

Otra característica interesante de los objetos Proxy es que pueden usarse para implementar patrones de herencia en JavaScript. Al utilizar un objeto Proxy como prototipo de otro objeto, puede interceptar búsquedas de propiedades y personalizar el comportamiento de la cadena de prototipos.

He aquí un ejemplo:



En este ejemplo, se define un objeto principal y tiene un atributo de nombre. Luego, creamos un objeto controlador con una trampa de obtención que evita cualquier solicitud de lectura de las propiedades del objeto secundario. La trampa utiliza el método Reflect.get() para recurrir al objeto principal si la propiedad está ausente en el objeto secundario.

Luego, utilizando un objeto Proxy como prototipo y el objeto controlador como controlador, construimos un objeto hijo. Finalmente, obtenemos acceso y modificamos la propiedad de nombre del objeto secundario, lo que activa get y set traps y registra mensajes en la consola.

**Ejemplos de la vida real usando Proxy**

**Almacenamiento en caché**

Un caso de uso de Proxy es almacenar en caché costosas llamadas a funciones. En este ejemplo, creamos un objeto Proxy que almacena en caché el resultado de una llamada a una función en función de sus argumentos.



**Ventajas y limitaciones del proxy**

**Ventajas del proxy**

1. **Comportamiento personalizable** : con objetos proxy, puede interceptar y personalizar operaciones básicas en otros objetos, lo que le permite crear funciones avanzadas como control de acceso, almacenamiento en caché y registro.
2. **Herencia** : los objetos proxy ofrecen la capacidad de implementar patrones de herencia en JavaScript, lo que puede generar un código más versátil y escalable.
3. **Revocable** : los objetos proxy se pueden deshabilitar o revocar después de su creación, lo que los hace útiles para limitar el alcance del objeto proxy o por razones de seguridad.

**Limitaciones del proxy**

1. A pesar de que el Proxy existe desde hace mucho tiempo, no todas las versiones de navegadores admiten esta funcionalidad.
2. Además, el uso de Proxies puede afectar negativamente el rendimiento de su aplicación, especialmente si los usa con demasiada frecuencia.
3. Es importante comprender el significado de utilizar un proxy. No se debe confiar en él para momentos críticos de la aplicación, como la validación importante de la entrada del usuario.

**Mejores prácticas para usar proxy**

**Tenga en cuenta las restricciones** : antes de implementar un proxy en su código, tenga en cuenta las restricciones que impone y cómo pueden afectar la velocidad y seguridad de su aplicación.

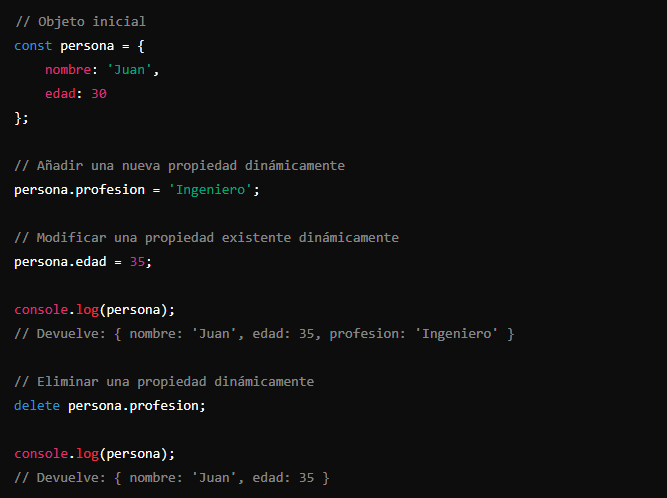
Los objetos proxy sólo deben usarse cuando sean absolutamente esenciales porque pueden afectar el rendimiento de su código.

**Pruebe cuidadosamente** : cuando utilice objetos Proxy, asegúrese de realizar pruebas cuidadosamente y esté alerta ante cualquier comportamiento potencialmente inesperado.

**Cumpla con las normas** : para que su código sea fácil de leer y mantener, cumpla con las convenciones aceptadas y las mejores prácticas al implementar objetos Proxy.

**o Que son las propiedades dinámicas de los objetos (de un ejemplo)**

Las propiedades dinámicas de los objetos en JavaScript son aquellas que pueden ser creadas, modificadas o eliminadas en tiempo de ejecución. Esto significa que no están definidas en la estructura inicial del objeto, sino que pueden ser añadidas o modificadas en cualquier momento durante la ejecución del programa.



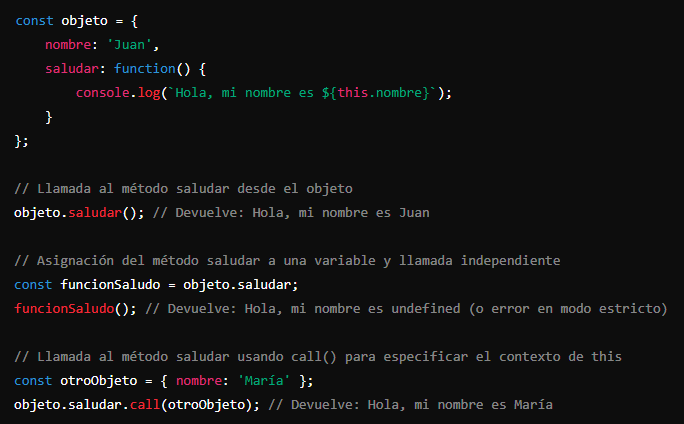
En este ejemplo, primero creamos un objeto **persona** con dos propiedades: **nombre** y **edad**. Luego, añadimos una nueva propiedad **profesion** de forma dinámica utilizando la sintaxis de punto (**persona.profesion = 'Ingeniero'**). Después, modificamos la propiedad **edad** de forma dinámica (**persona.edad = 35**). Finalmente, eliminamos la propiedad **profesion** de forma dinámica utilizando el operador **delete**.

Este ejemplo demuestra cómo las propiedades de un objeto pueden ser manipuladas dinámicamente durante la ejecución del programa, lo que proporciona una flexibilidad adicional al trabajar con objetos en JavaScript.

**• This en JavaScript**

**o Que es el método this y cuál es su alcance (de un ejemplo)**

El método this en JavaScript no es un método en sí mismo, sino una palabra clave que se refiere al objeto actual en el cual se está ejecutando el código. El alcance de this depende del contexto de ejecución en el que se encuentra.

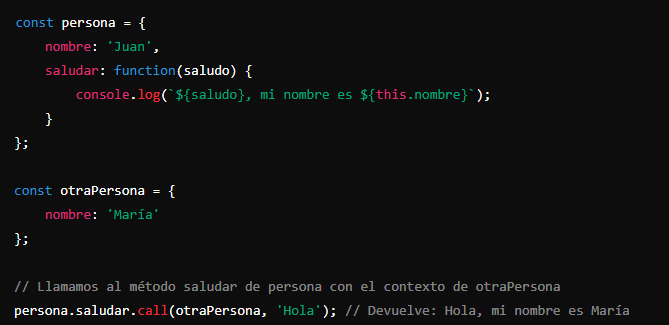


En este ejemplo, **this** dentro del método **saludar** hace referencia al objeto **objeto** cuando se llama al método directamente desde el objeto (**objeto.saludar()**). Sin embargo, cuando asignamos el método **saludar** a una variable (**funcionSaludo**) y lo llamamos de forma independiente (**funcionSaludo()**), **this** ya no se refiere a **objeto**, sino al objeto global (**window** en el navegador) o a **undefined** en modo estricto.

Para solucionar este problema y asegurar que **this** se refiera al objeto adecuado, podemos usar el método **call()** para especificar explícitamente el valor de **this**. En el último ejemplo, llamamos al método **saludar** utilizando **call()** y pasamos **otroObjeto** como argumento para establecer el contexto de **this** como **otroObjeto**. Esto hace que **this** dentro de la función **saludar** se refiera al objeto **otroObjeto**.

**o Que son los métodos call (de un ejemplo)**

El método call es un método disponible en todas las funciones en JavaScript que nos permite llamar a una función con un valor específico para this, además de argumentos individuales pasados por separado. Este método es útil cuando queremos ejecutar una función en un contexto específico diferente al contexto en el que fue definida originalmente.



En este ejemplo, tenemos un objeto **persona** con un método **saludar** que espera un saludo como argumento. Luego, creamos otro objeto **otraPersona** con un nombre diferente. Utilizando el método **call**, podemos llamar al método **saludar** de **persona**, pero estableciendo el contexto de **this** como **otraPersona**, lo que hace que **this.nombre** dentro del método **saludar** se refiera al nombre de **otraPersona**.

**o Que son los métodos aplay (de un ejemplo de cada uno)**

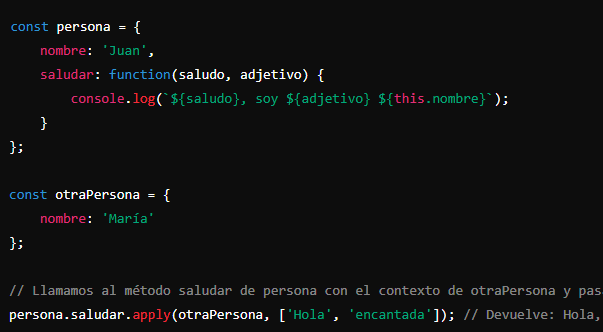
El método apply es una función de alto orden que se utiliza para llamar a una función con un valor específico para this, así como para pasar un conjunto de argumentos a la función llamada en forma de array.

**La sintaxis del método apply es la siguiente:**

* funcion: La función que se va a llamar.
* valorThis: El valor que se utilizará como this dentro de la función llamada.
* argsArray: Un array que contiene los argumentos que se pasarán a la función llamada.

Cuando llamamos a apply en una función, la función especificada como primer argumento se ejecuta, y el contexto de this dentro de esa función se establece como el valor pasado como primer argumento. Además, los argumentos contenidos en el array pasado como segundo argumento se pasan a la función como argumentos individuales.

Aquí hay un ejemplo de cómo usar el método apply:



En este ejemplo, persona.saludar.apply(otraPersona, ['Hola', 'encantada']) llama al método saludar de persona, pero establece el contexto de this como otraPersona. Además, pasamos dos argumentos adicionales ('Hola' y 'encantada') mediante un array como segundo argumento de apply. Esto permite que el método saludar los utilice correctamente dentro de su cuerpo.

**QUE ES BIND**

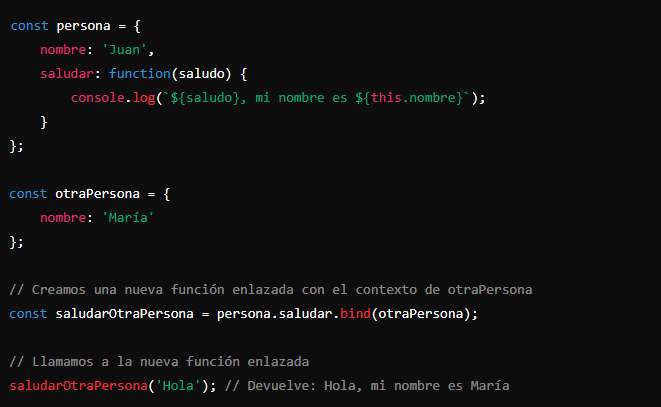
El método **bind** es un método que está disponible en todas las funciones en JavaScript. Se utiliza para crear una nueva función que, cuando se llama, tiene un valor de **this** predefinido, así como cualquier cantidad de argumentos predefinidos.

La sintaxis del método **bind** es la siguiente:



* **funcion**: La función original que se va a enlazar.
* **valorThis**: El valor que se utilizará como **this** dentro de la función enlazada.
* **arg1**, **arg2**, ...: Argumentos opcionales que se pasarán a la función enlazada cuando se llame.

EJEMPLO:



En este ejemplo, utilizamos **bind** para crear una nueva función llamada **saludarOtraPersona**, que tiene el contexto de **this** establecido como **otraPersona**. Luego, llamamos a esta nueva función **saludarOtraPersona** con el argumento **'Hola'**, lo que produce la salida correcta: "Hola, mi nombre es María".