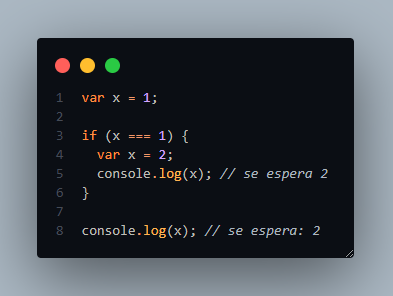
**A. Actividades de reflexión inicial.**

**1.Que son las variables var y las variables let (de un ejemplo):**

**VAR**

Esta forma declara una variable que puede ser inicializada opcionalmente con un valor, esto significa que puedo escribirla como var x o var x = 1, y su valor puede ir cambiando durante la ejecución de tu programa. En el siguiente código te muestro como el valor de x cambia imprimiéndole usando console log, usa tu editor favorito y cambia los valores asignados para que veas como cambia lo que se esta mostrando en consola.



**LET**

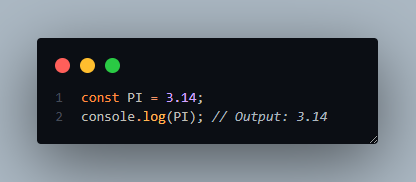
Las variables declaradas con let tienen un ámbito de bloque limitado a su contenedor (bloque {}). Esto significa que no son accesibles fuera de ese bloque.

No se pueden redeclarar dentro del mismo ámbito.

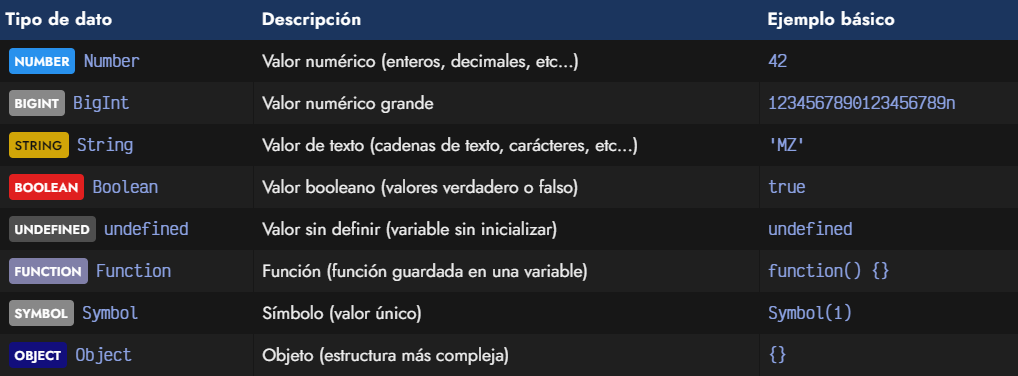


**Que son las constantes (conts) (de un ejemplo)**

con const definimos variables de sólo lectura (no confundir con inmutables), esto quiere decir que, cuando asignamos una variable, el nombre de esta va estar asignada a un puntero en memoria, el cual no puede ser sobreescrito o reasignado.



**TIPOS DE DATOS**

****

**Que es NaN**

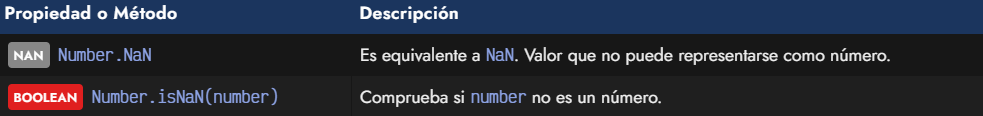
El acrónimo NaN es un valor especial de Javascript que significa literalmente Not A Number (No es un número). Sin embargo, no hay que dejarse llevar por su significado literal, ya que nos podría dar lugar a malentendidos. El valor NaN, a pesar de su significado, se usa para representar valores numéricos (y ahora es donde viene el matiz) que son indeterminados o imposibles de representar como número.

Dentro de esa categoría hay varios:

* **Indeterminación matemática:** Por ejemplo, 0 / 0.
* **Valores imposibles**: Por ejemplo, 4 - 'a', ya que es imposible restar una letra a un número.
* **Operaciones con NaN:** Por ejemplo, NaN + 4, ya que el primer operando es NaN.

Uno de los más frecuentes es el segundo, ya que Javascript es un lenguaje flexible que no requiere tipos. Eso, unido a que la mayoría de datos que extraemos de una página web se obtienen como , da como resultado operaciones de ese tipo, o concatenaciones inesperadas.

Por aquí puedes encontrar las propiedades y métodos relacionados con NaN que existen:



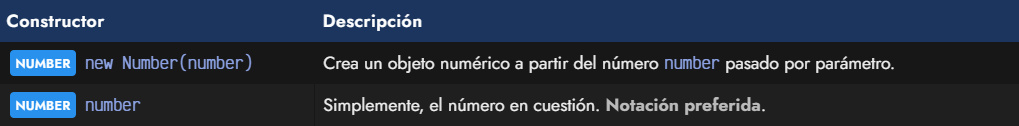
Texto

Descripción generada automáticamente

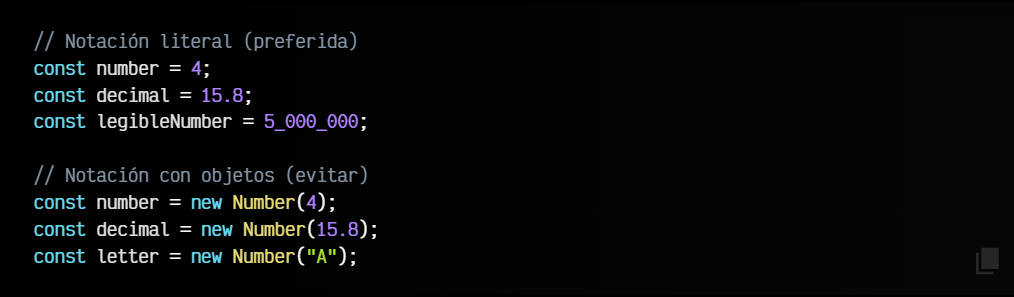
**QUE SON LOS NUMBERS**

**Pero como VARIABLE:**

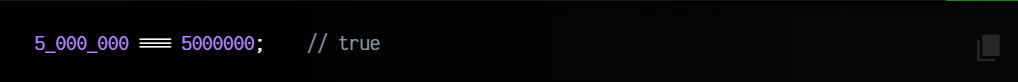
En Javascript, los números son uno de los tipos de datos básicos (tipos primitivos), que, para crearlos, simplemente basta con escribirlos literalmente. No obstante, como en Javascript todo se puede representar con objetos (como veremos más adelante) también se pueden declarar mediante la palabra clave new:



Sin embargo, aunque existan estas dos formas de declararlas, no se suele utilizar la notación new con números, ya que es bastante más tedioso y complicado, por lo que lo preferible es utilizar la notación de literales:

****

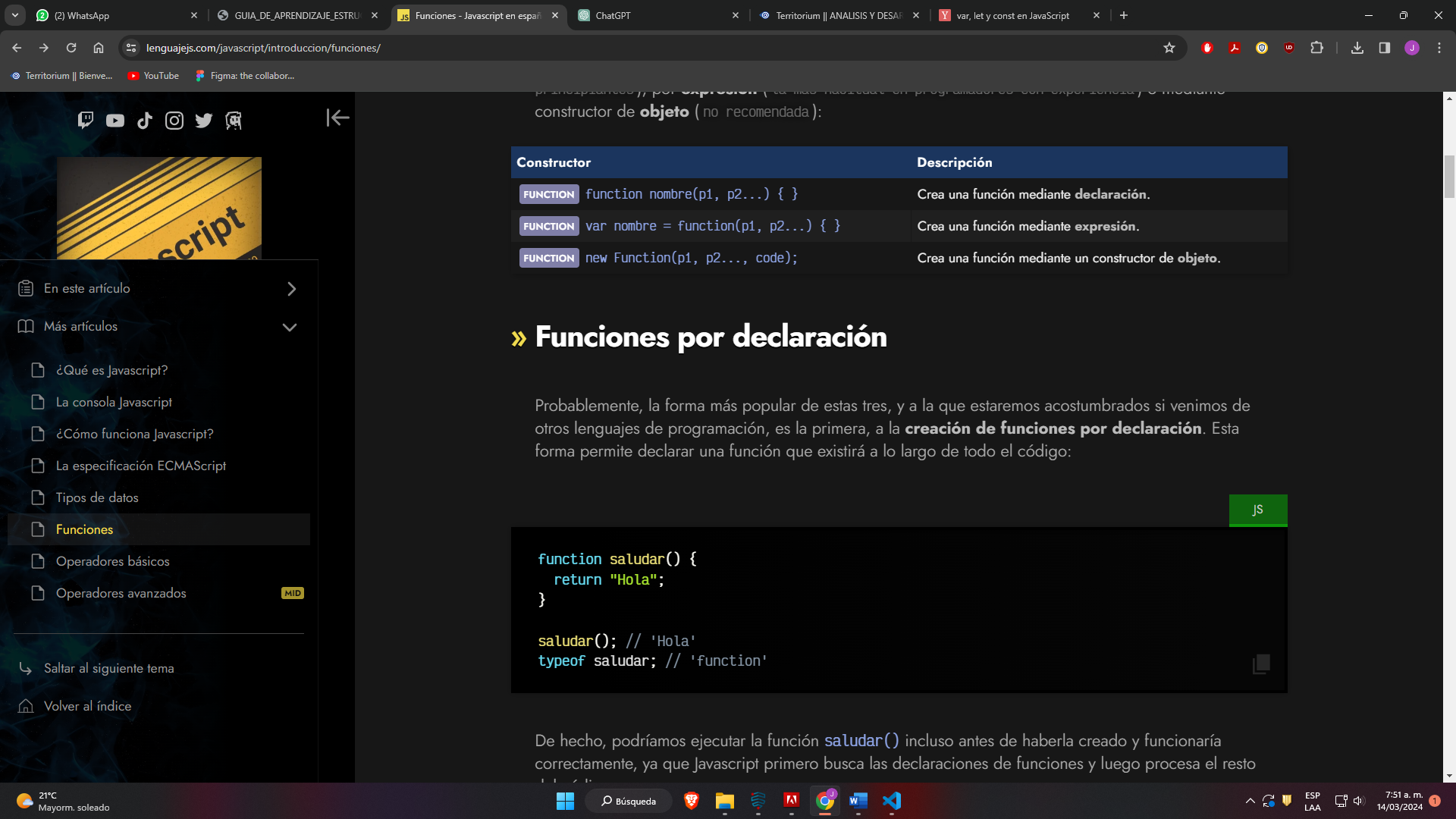
Observa que los números con decimales, en Javascript los separamos con un punto ( . ), mientras que de forma opcional, podemos utilizar el guión bajo ( \_ ) para separar visualmente y reconocer las magnitudes que usamos, teniendo en cuenta que para Javascript es lo mismo:



**Que son las funciones y que tipos de funciones existente en JavaScript (de ejemplos)**

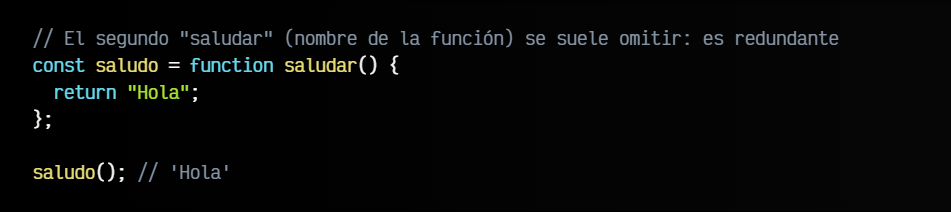
**Funciones por declaraciones:**

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



**Funciones por expresión**

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

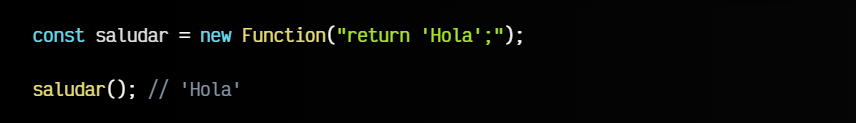


Con este nuevo enfoque, estamos creando una función en el interior de una variable, lo que nos permitirá posteriormente ejecutar la variable (como si fuera una función). Observa que el nombre de la función (en este ejemplo: saludar) pasa a ser inútil, ya que si intentamos ejecutar saludar() nos dirá que no existe y si intentamos ejecutar saludo() funciona correctamente.

¿Qué ha pasado? Ahora el nombre de la función pasa a ser el nombre de la variable, mientras que el nombre de la función desaparece y se omite, dando paso a lo que se llaman las funciones anónimas (o funciones lambda).

**Funciones como objetos**

Como curiosidad, debes saber que se pueden declarar funciones como si fueran objetos. Sin embargo, es un enfoque que no se suele utilizar en producción. Simplemente es interesante saberlo para darse cuenta que en Javascript todo pueden ser objetos:



**Funciones anónimas**

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

Texto

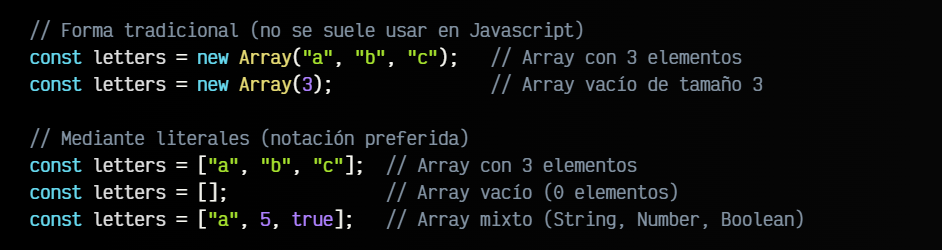
Descripción generada automáticamente con confianza media

**Que son los arreglos (Arrays) (de ejemplos)**

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

Texto

Descripción generada automáticamente



**Acceso a elementos del array**

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

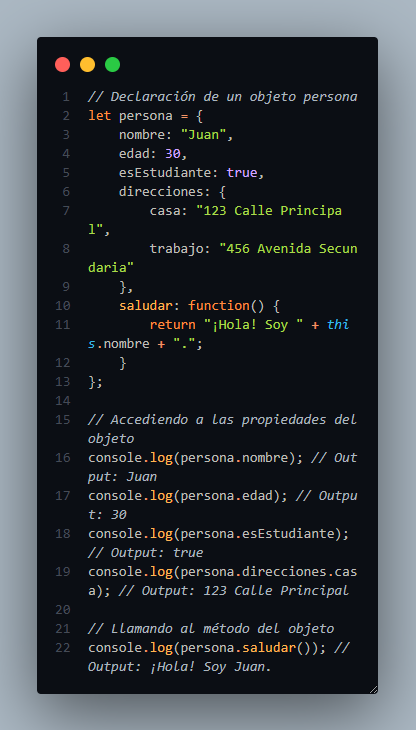
Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

<https://lenguajejs.com/javascript/arrays/que-es/>

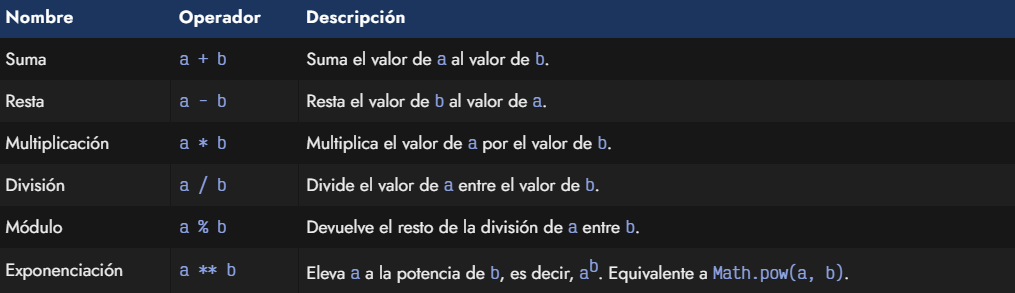
**Que son los objetos en JavaScript (de un ejemplo)**

En JavaScript, los objetos son estructuras de datos que pueden contener múltiples valores en forma de pares clave-valor. Estos valores pueden ser primitivos (como números, cadenas, booleanos, etc.) o incluso otras estructuras de datos como otros objetos, matrices o funciones. Los objetos en JavaScript son fundamentales ya que proporcionan una forma de organizar y manipular datos de manera estructurada.



**Estructuras de control:**

**Tipos de operadores (de un ejemplo de cada uno)**

****

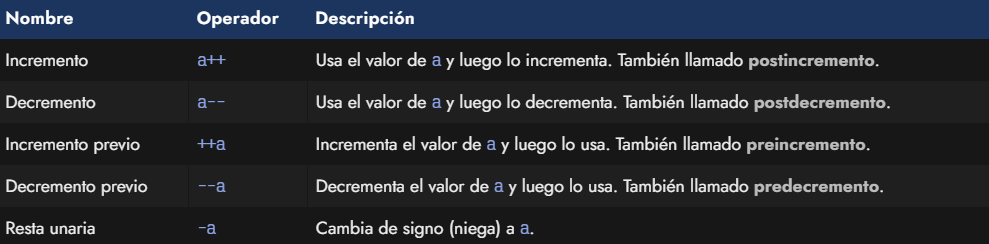
**Operadores de asignación**

**Captura de pantalla de computadora

Descripción generada automáticamente**

**Operadores unarios**

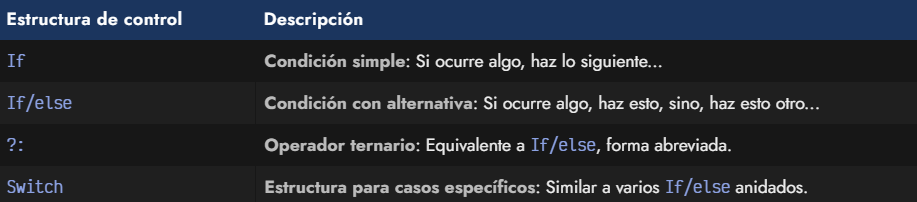
Los operadores unarios son aquellos que en lugar de tener dos operandos, como los anteriores, sólo tienen uno. Es decir, se realizan sobre un sólo valor almacenado en una variable.



[**https://lenguajejs.com/javascript/introduccion/operadores-basicos/**](https://lenguajejs.com/javascript/introduccion/operadores-basicos/)

**Condicionales (de un ejemplo)**

**CONDICIONALES:**

****

**Condicional If**

Quizás, el más conocido de estos mecanismos de estructura de control es el if (condicional). Con él podemos indicar en el programa que se tome un camino sólo si se cumple la condición que establezcamos.

Observa el siguiente ejemplo, donde guardamos en el «compartimento» nota, un valor numérico:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

**Condicional If / else**

Se puede dar el caso que queramos establecer una alternativa a una condición. Para eso utilizamos el if seguido de un else. Con esto podemos establecer una acción A si se cumple la condición, y una acción B si no se cumple.

Vamos a modificar el ejemplo anterior para mostrar también un mensaje cuando estamos suspendidos, pero en este caso, en lugar de mostrar el mensaje directamente con un console.log vamos a guardar ese texto en una nueva variable calificacion:

Texto

Descripción generada automáticamente

**Condicional Switch**

La estructura de control switch permite definir casos específicos a realizar cuando la variable expuesta como condición sea igual a los valores que se especifican a continuación mediante cada case:

Texto

Descripción generada automáticamente

**Hay varias puntualizaciones que aclarar sobre este ejemplo, así que vamos a explicarlo:**

* La sentencia switch establece que vamos a realizar múltiples condiciones analizando la variable nota.
* Cada condición se establece mediante un case, seguido del valor posible de cada caso.
* El switch comienza evaluando el primer case, y continua con el resto, hacia abajo.
* Observa que algunos case tienen un break. Esto hace que deje de evaluar y se salga del switch.
* Los case que no tienen break, no se interrumpen, sino que se salta al siguiente case.
* El caso especial default es como un else. Si no entra en ninguno de los anteriores, entra en default.

**Operador ternario**

El operador ternario es una alternativa al condicional if/else de una forma mucho más compacta y breve, que en muchos casos resulta más legible. Sin embargo, hay que tener cuidado, porque su sobreutilización puede ser contraproducente y producir un código más difícil de leer.

La sintaxis de un operador ternario es la siguiente:



Para entenderlo bien, vamos a reescribir el ejemplo de los temas anteriores utilizando este operador ternario. Primero, recordemos el ejemplo utilizando estructuras if/else:

Texto

Descripción generada automáticamente

Ahora, vamos a reescribirlo utilizando un operador ternario:

Texto

Descripción generada automáticamente

**Repasemos el ejemplo:**

* Observa que guardamos en calificacion el resultado del operador ternario.
* La condición es nota < 5, se escribe al principio, previo al ?.
* Si la condición es cierta, el ternario devuelve "suspendido".
* Si la condición es falsa, el ternario devuelve "aprobado".

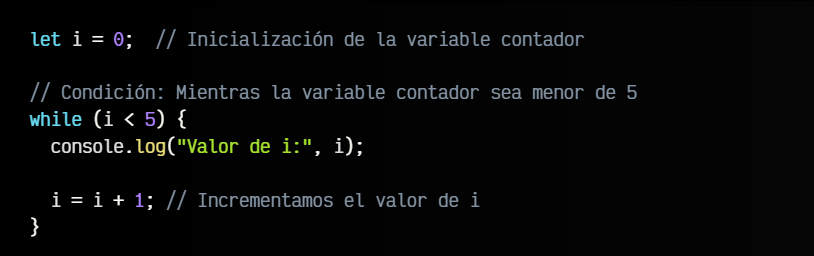
Este ejemplo hace exactamente lo mismo que el ejemplo anterior del if/else. La idea del operador ternario es que podemos condensar mucho código y tener un if en una sola línea. Es muy práctico, legible e ideal para ejemplos pequeños donde almacenamos la información en una variable para luego utilizarla.

<https://lenguajejs.com/fundamentos/estructuras-de-control/operador-ternario/>

**Ciclos (Loops) (de un ejemplo de cada uno)**

**Bucle while**

El bucle while es uno de los bucles más simples que podemos crear. Vamos a repasar el siguiente ejemplo y analizar todas sus partes, para luego analizar lo que ocurre en cada iteración del bucle. Empecemos por un fragmento sencillo del bucle:

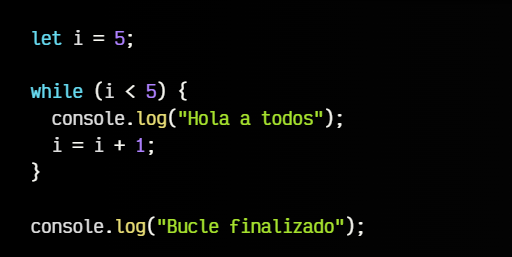


Antes de entrar en el bucle while, se inicializa la variable i al valor 0.

1. Antes de realizar la primera iteración del bucle, comprobamos la condición.
2. Si la condición es verdadera, hacemos las tareas que están indentadas dentro del bucle.
3. Mostramos por pantalla el valor de i.
4. Luego, incrementamos el valor de i sumándole 1 a lo que ya teníamos en i.
5. Terminamos la iteración del bucle, por lo que volvemos al inicio del while a hacer una nueva iteración.
6. Volvemos al punto 2) donde comprobamos de nuevo la condición del bucle.
7. Repetimos hasta que la condición sea falsa. Entonces, salimos del bucle y continuamos el programa.

**Bucle do ... while**

Existe una variación del bucle while denominado bucle do while. La diferencia fundamental, a parte de variar un poco la sintaxis, es que este tipo de bucle siempre se ejecuta una vez, al contrario que el bucle while que en algún caso podría no ejecutarse nunca.



Observa, que aún teniendo un bucle, este ejemplo nunca mostrará el texto Hola a todos, puesto que la condición nunca será verdadera, porque ya ha empezado como falsa (i ya vale 5 desde el inicio). Por lo tanto, nunca se llega a realizar el interior del bucle.

Con el bucle do while podemos obligar a que siempre se realice el interior del bucle al menos una vez:

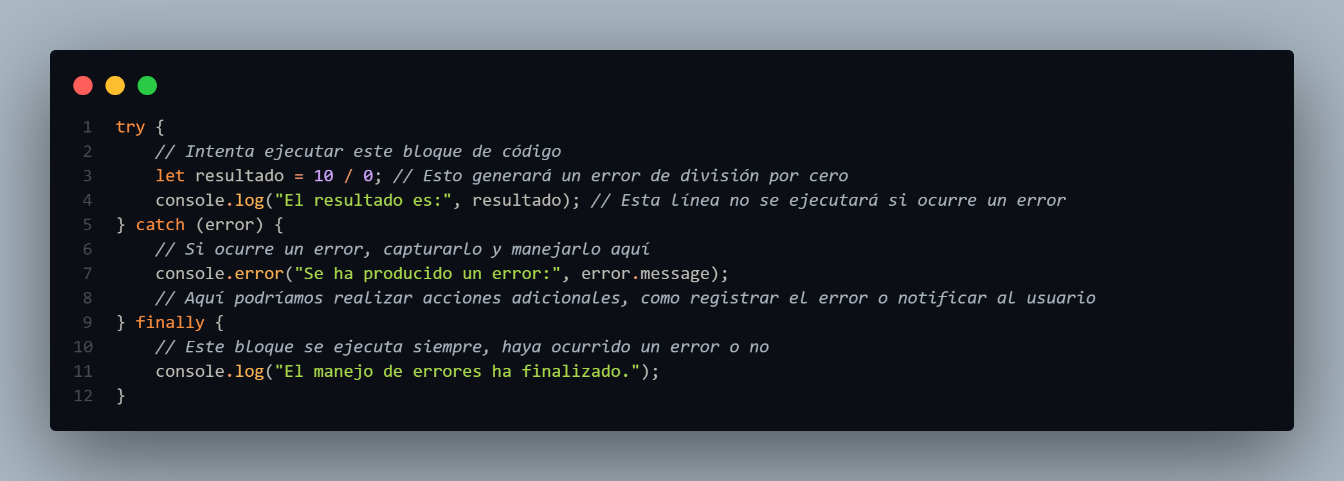
Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Observa los siguientes detalles de la variación do while:

* En lugar de utilizar un while desde el principio junto a la condición, escribimos do.
* El while con la condición se traslada al final del bucle.
* Lo que ocurre en este caso es que el interior del bucle se realiza siempre, y sólo se analiza la condición al terminar el bucle, por lo que aunque no se cumpla, se va a realizar al menos una vez

**MANEJO DE ERRORES**

****

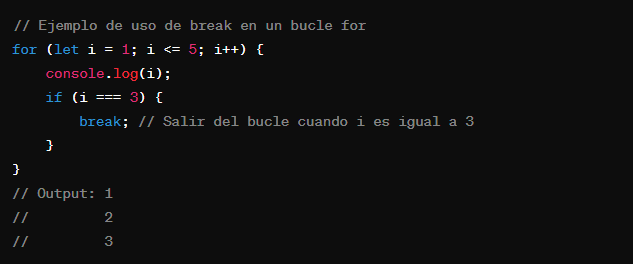
En JavaScript, el manejo de errores se realiza utilizando bloques try...catch. Estos bloques nos permiten intentar ejecutar un bloque de código y capturar cualquier error que pueda ocurrir durante la ejecución. Aquí tienes un

En este ejemplo, intentamos dividir 10 entre 0, lo que resulta en un error de división por cero. El bloque try intenta ejecutar esta operación, y si se produce un error, se captura y se maneja en el bloque catch, donde se imprime un mensaje de error. El bloque finally se ejecutará independientemente de si se produce un error o no, y en este caso, se imprime un mensaje indicando que el manejo de errores ha finalizado.

**Break & Continue (de un ejemplo de cada uno)**

**break:**

**La instrucción break se utiliza para salir inmediatamente de un bucle cuando se alcanza cierta condición.**

****

**continue:**

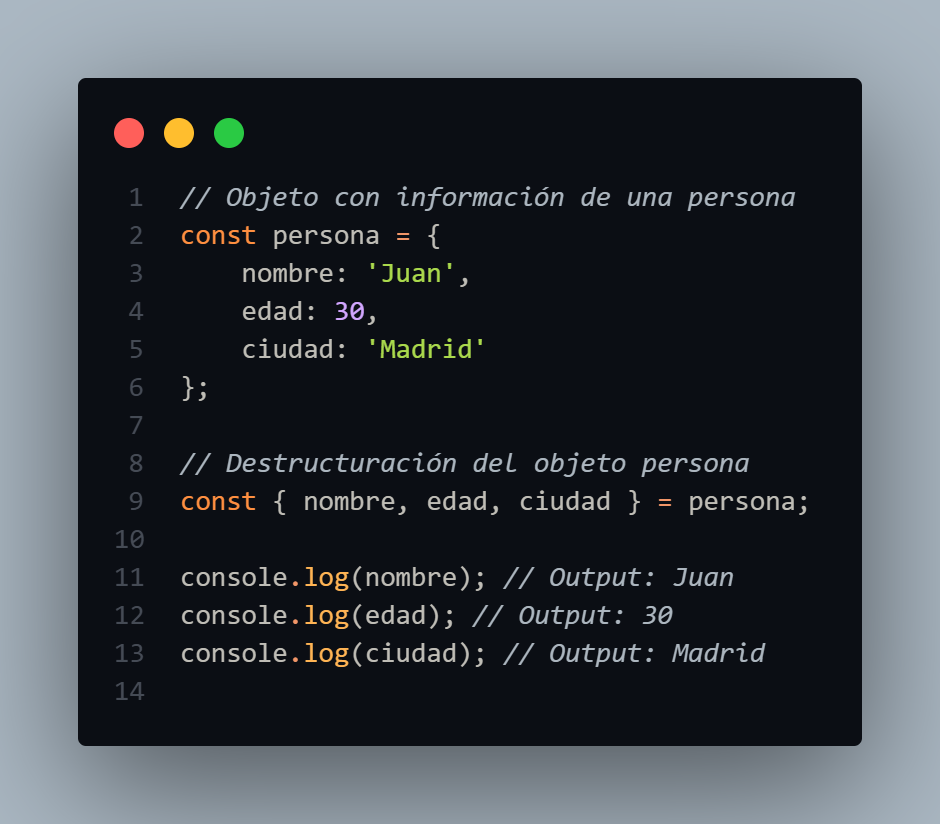
**La instrucción continue se utiliza para saltar a la siguiente iteración del bucle cuando se alcanza cierta condición, omitiendo cualquier código restante en el bloque del bucle para esa iteración.**

**Texto

Descripción generada automáticamente**

**Que es la destructuración (de un ejemplo)**

**La destructuración es una característica de JavaScript que permite descomponer una estructura de datos en partes más pequeñas, como variables individuales. Se utiliza principalmente con objetos y matrices, y es una forma conveniente de extraer valores de estos tipos de datos de una manera más concisa y legible.**

****

**Que son los objetos literales (de un ejemplo)**

**Los objetos literales en JavaScript son una forma de definir y crear objetos de manera directa utilizando una sintaxis sencilla y declarativa. Estos objetos se crean mediante la asignación de pares clave-valor entre llaves {}. Cada par clave-valor define una propiedad del objeto, donde la clave es el nombre de la propiedad y el valor puede ser cualquier tipo de dato válido en JavaScript, como un número, una cadena, un booleano, una función, otro objeto, etc.**

****

**Que son los parámetros REST y Operador Spread (de un ejemplo de cada uno)**

**Los parámetros REST y el operador spread (...) son características de JavaScript que nos permiten trabajar con un número variable de argumentos o elementos. Ambos se utilizan en funciones y en la manipulación de arreglos. Aquí tienes un ejemplo de cada uno:**

**Parámetros REST:**

**Los parámetros REST nos permiten representar un número variable de argumentos como un arreglo dentro de una función. Esto es útil cuando no sabemos cuántos argumentos serán pasados a la función.**

**Texto

Descripción generada automáticamente**

**Operador Spread:**

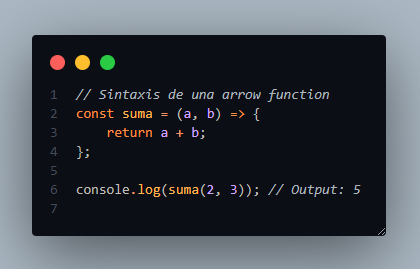
**El operador spread (...) se utiliza para expandir o desempaquetar elementos de un arreglo o un objeto. Se puede usar en la creación de arreglos, llamadas a funciones o en cualquier lugar donde se esperen múltiples elementos.**

**Captura de pantalla de computadora

Descripción generada automáticamente**

**LOS ARROW**

**Las arrow functions, o funciones flecha, son una característica introducida en ECMAScript 6 (también conocido como ES6) que proporciona una sintaxis más concisa para definir funciones en JavaScript. Estas funciones son especialmente útiles cuando se necesita una función anónima y tienen una sintaxis más corta y clara en comparación con las funciones tradicionales.**

****

**En este ejemplo, hemos definido una arrow function llamada suma que toma dos parámetros a y b, y devuelve la suma de estos dos parámetros. La sintaxis de la arrow function consiste en los parámetros (entre paréntesis), seguidos de la flecha =>, y luego el cuerpo de la función (entre llaves {}). En este caso, el cuerpo de la función simplemente contiene una expresión de retorno que suma a y b.**

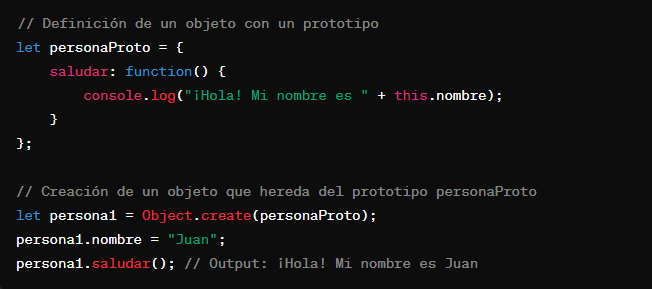
**Programación Orientada a Objetos (POO):**

**La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en la idea de "objetos" que pueden contener datos en forma de campos (también conocidos como propiedades o atributos) y código en forma de métodos (funciones asociadas al objeto). La POO se centra en la creación de objetos que interactúan entre sí para realizar tareas específicas.**

**Prototipos:**

**En JavaScript, los prototipos son un mecanismo que permite la herencia entre objetos. Cada objeto en JavaScript tiene un prototipo interno que actúa como un "padre" del objeto, del cual hereda propiedades y métodos. Los prototipos permiten que los objetos compartan funcionalidades y propiedades comunes sin tener que duplicar código.**

**Ejemplo:**

****

**Herencia Prototípica:**

**La herencia prototípica es un concepto en JavaScript que permite a un objeto heredar propiedades y métodos de otro objeto conocido como su "prototipo". Cuando se busca una propiedad o método en un objeto y no se encuentra, JavaScript lo busca en el prototipo del objeto y continúa buscando en la cadena de prototipos hasta encontrarlo o llegar al final de la cadena.**

**Ejemplo:**

**Texto

Descripción generada automáticamente**

**Métodos Estáticos, Getters y Setters:**

**Métodos Estáticos: Son métodos asociados a la clase misma, no a instancias específicas de la clase. Se invocan directamente desde la clase, no desde una instancia.**

**Ejemplo:**

**Texto

Descripción generada automáticamente**

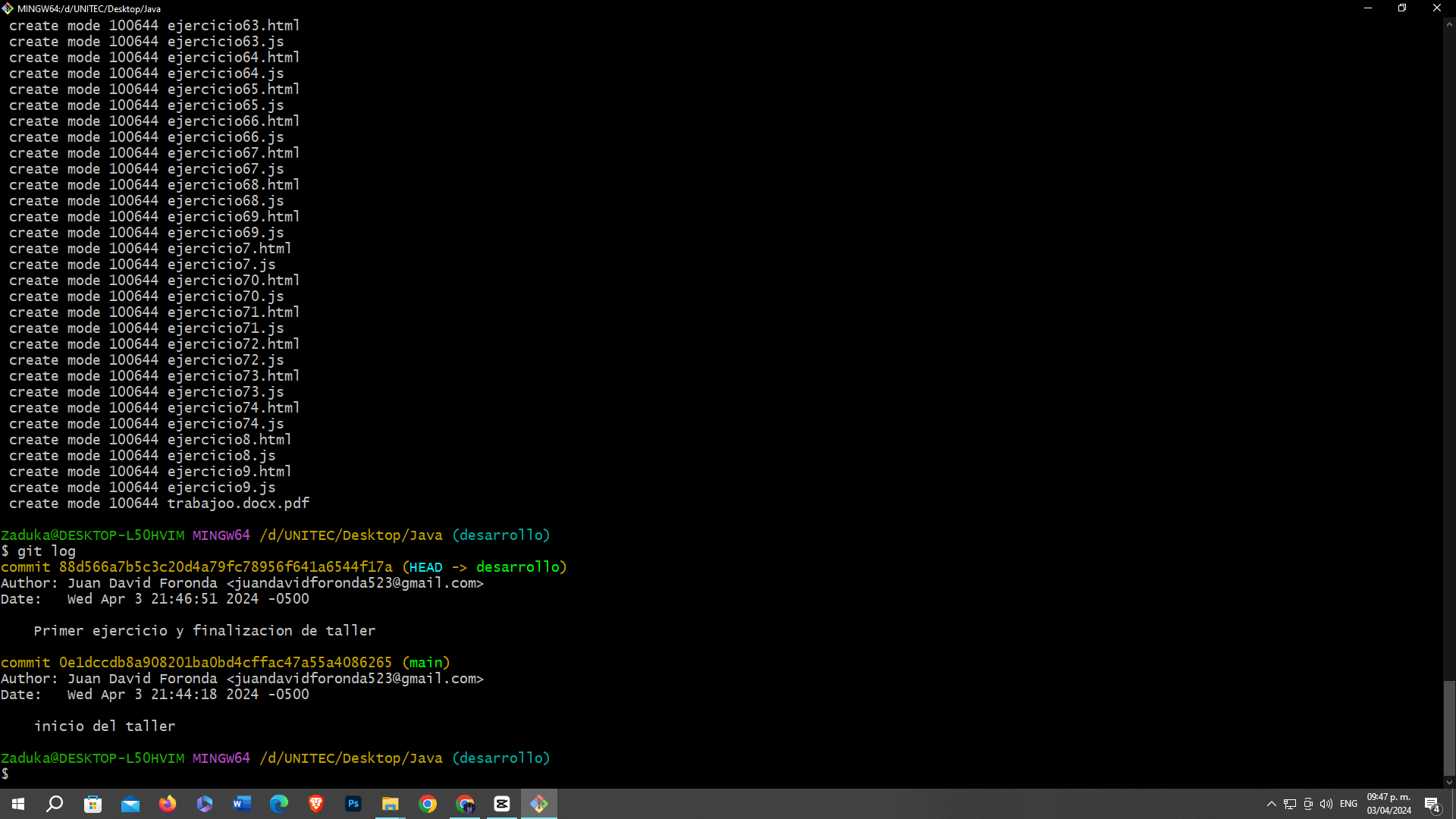
**Getters y Setters: Son métodos especiales que se utilizan para obtener y establecer el valor de propiedades de un objeto. Los getters se utilizan para obtener el valor de una propiedad, mientras que los setters se utilizan para establecer el valor de una propiedad.**

**Ejemplo:**

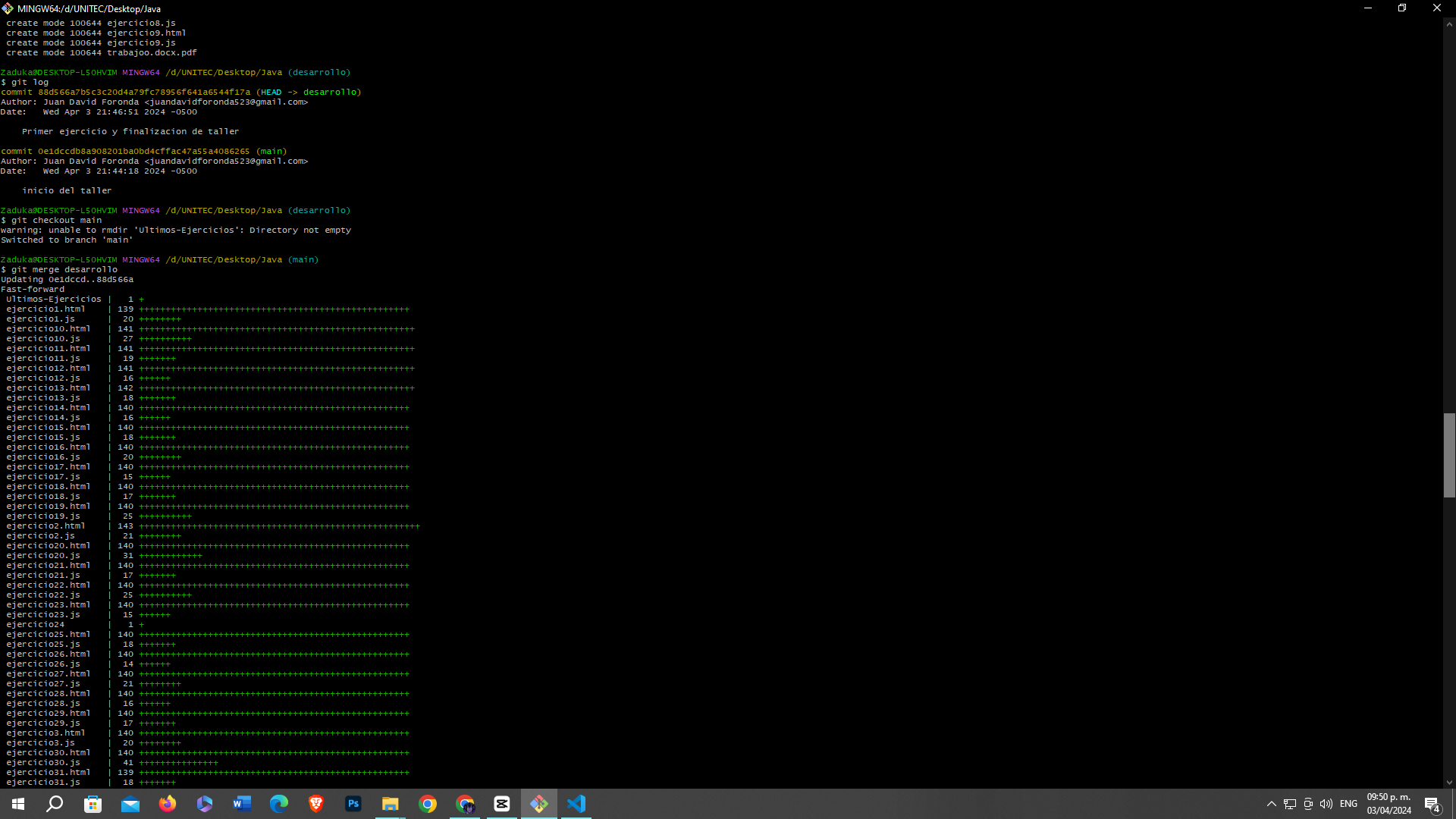
**Texto

Descripción generada automáticamente**

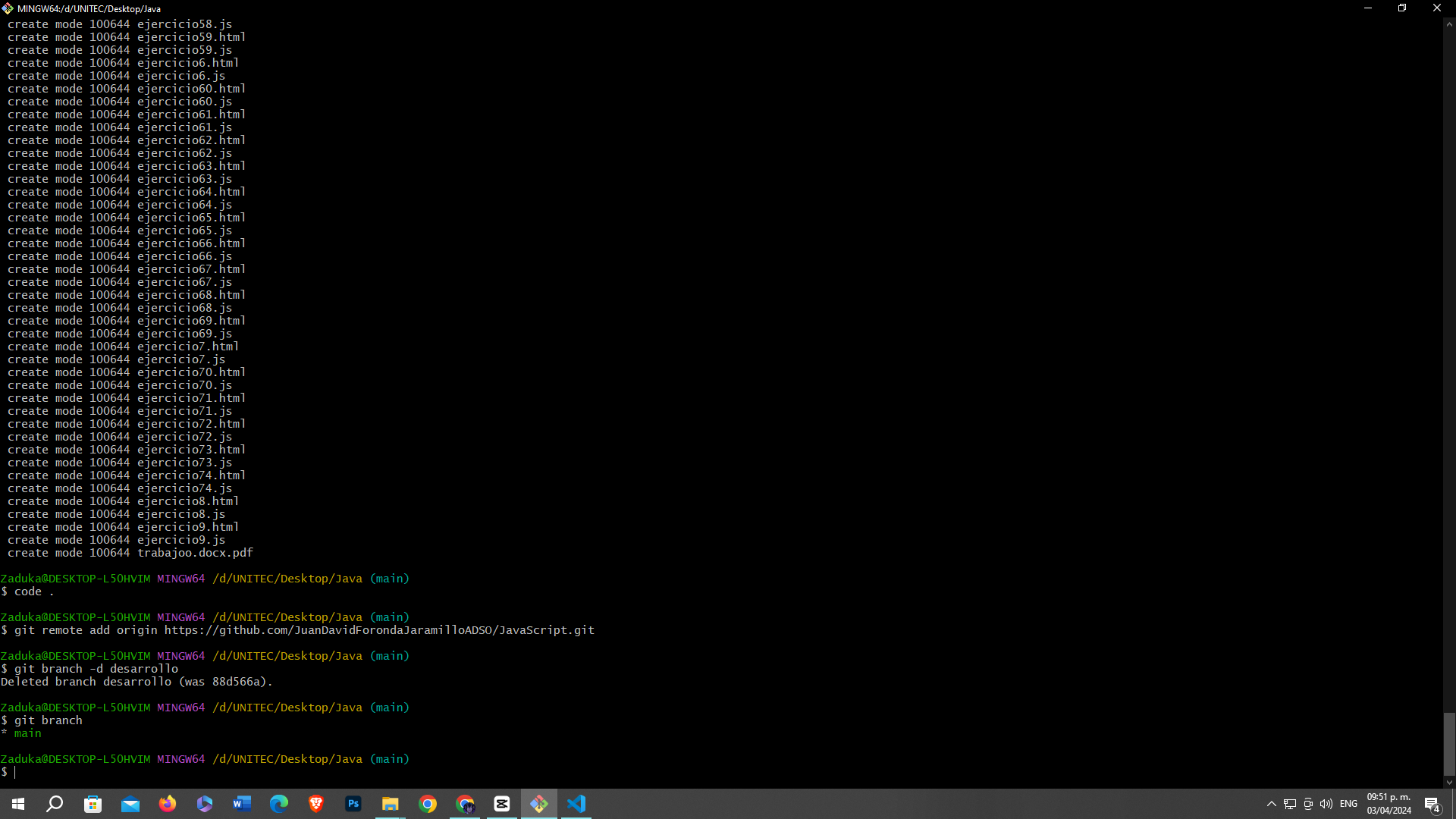
**LOGS:**

****

**GIT MERGE:**

****

**ELIMINADO DESAROLLO:**

****