Recuerden poner a grabar la clase

Clase 4

Funciones

Diplomatura UNTREF

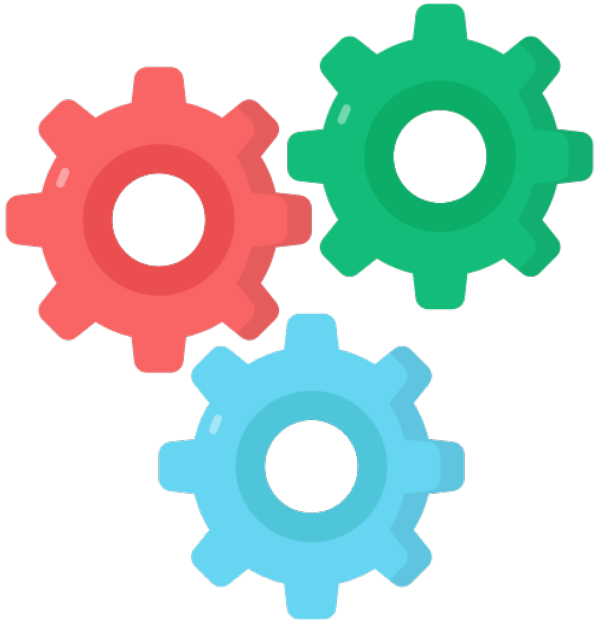
Temario

- Estructura de funciones
  - Definir la función
  - llamar a la función
- Funciones con parámetros
- Funciones con retorno
- Scope de una variable

Estructura de funciones

Funciones

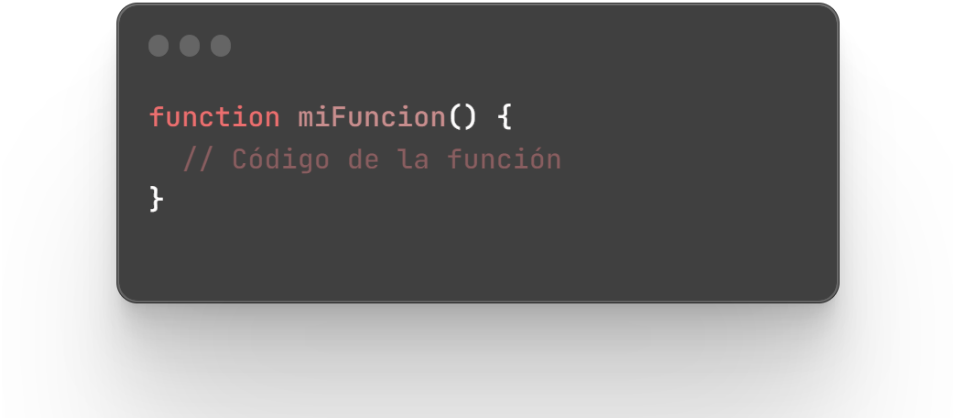
Las funciones son bloques de código reutilizables que se utilizan para realizar tareas específicas. Pueden aceptar argumentos (datos de entrada) y devolver un resultado. Las funciones en JavaScript se definen utilizando la palabra clave function y se pueden llamar en diferentes partes de un programa según sea necesario.



Funciones anatomía

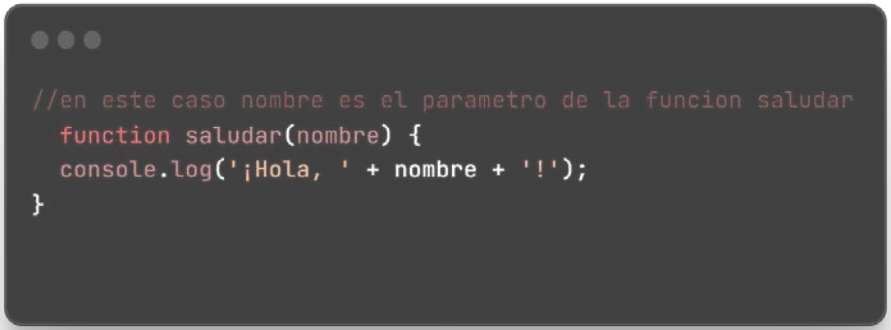
Vamos a describir las diferentes partes de una función:

Declaración de función: Una función se declara utilizando la palabra clave function, seguida del nombre de la función y un par de paréntesis ().



Funciones anatomía

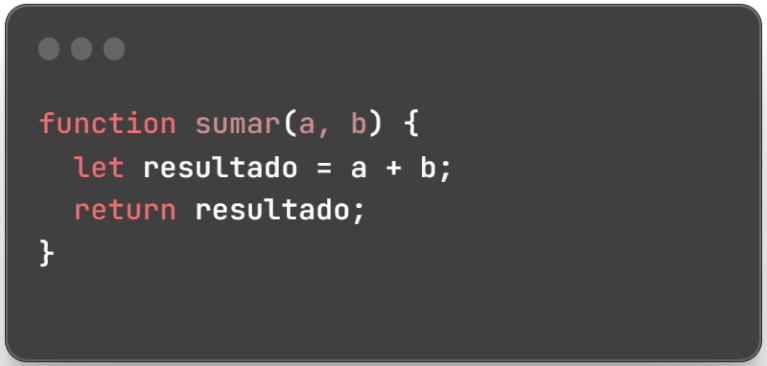
Parámetros: Los parámetros son variables que se definen dentro de los paréntesis de la declaración de la función. Representan los valores que se pasan a la función cuando se llama.



```
//en este caso nombre es el parametro de la funcion saludar
  function saludar(nombre) {
  console.log('¡Hola, ' + nombre + '!');
}
```

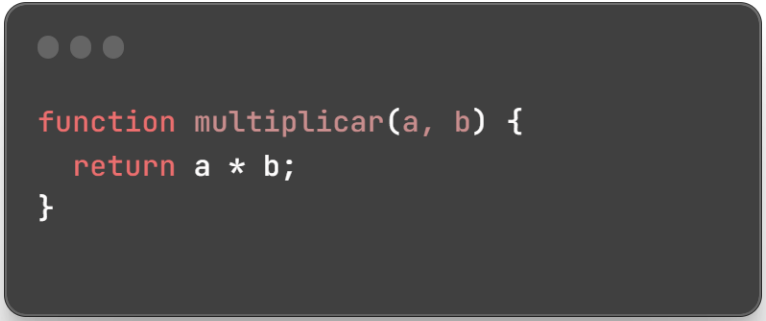
Funciones anatomía

Cuerpo de la función: El cuerpo de la función contiene el bloque de código que se ejecuta cuando se llama a la función. Está delimitado por un par de llaves {}.



Funciones anatomía

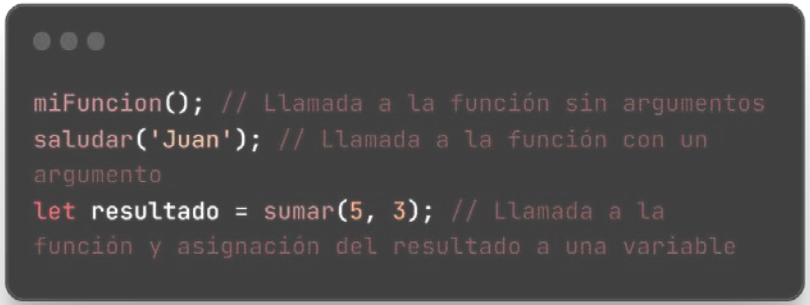
Retorno de valor: Una función puede devolver un resultado utilizando la palabra clave return. Esto permite obtener un valor de la función para su posterior uso.



```
function multiplicar(a, b) {
  return a * b;
}
```

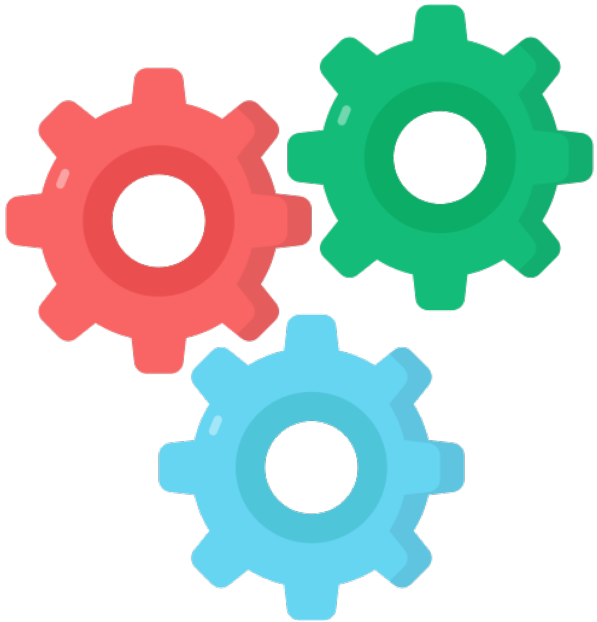
Funciones llamado

Una vez que una función está definida, se puede llamar o invocar en diferentes partes del programa utilizando su nombre y los paréntesis (). Al llamar a una función, se pueden proporcionar los argumentos necesarios en los paréntesis si la función espera recibir parámetros.



Funciones llamado

Para llamar a una función en JavaScript, simplemente debes escribir su nombre seguido de un par de paréntesis (). Si la función espera recibir argumentos, puedes proporcionarlos dentro de los paréntesis separados por comas.



Funciones llamado

Tenesmos diferentes maneras de llamar a una función

1. Llamada a una función sin argumentos:

```
miFuncion();
función y asignación del resultado a una variable
```

2. Llamada a una función con argumentos:

```
saludar('Juan', 'Pérez');
```

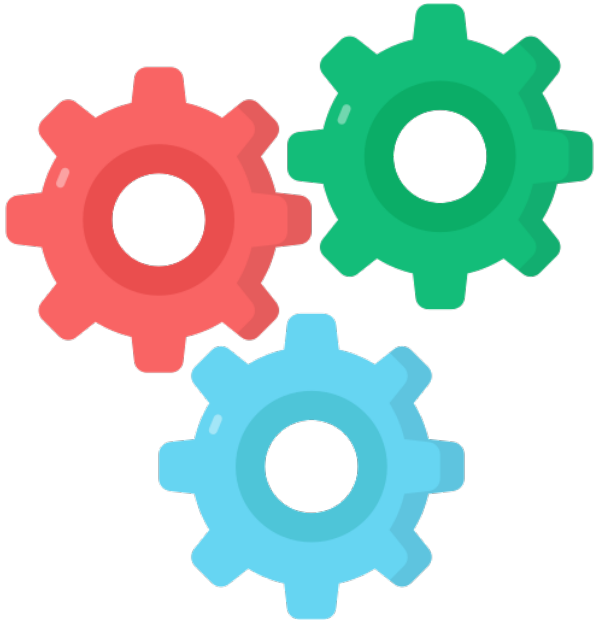
3. Llamada a una función y asignación del resultado:

```
let resultado = sumar(5, 3);
```

Funciones llamado

Recuerda que al llamar a una función, debes asegurarte de que su definición esté previamente disponible en el contexto en el que la llamas. Es decir, si declaras una función en un archivo o en una sección específica del código, debes asegurarte de que esté en un lugar accesible antes de llamarla.

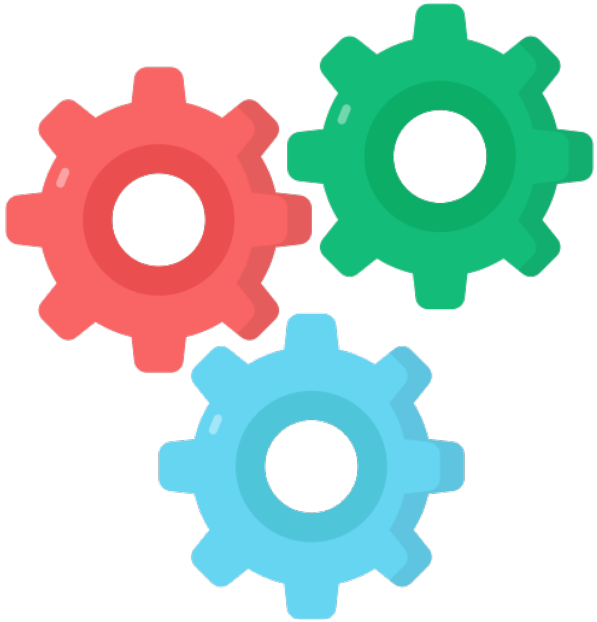
Además, ten en cuenta que las funciones también pueden retornar un valor utilizando la palabra clave `return`, y puedes capturar ese valor al llamar a la función y asignarlo a una variable u utilizarlo en otra parte de tu programa.



Funciones con parámetros

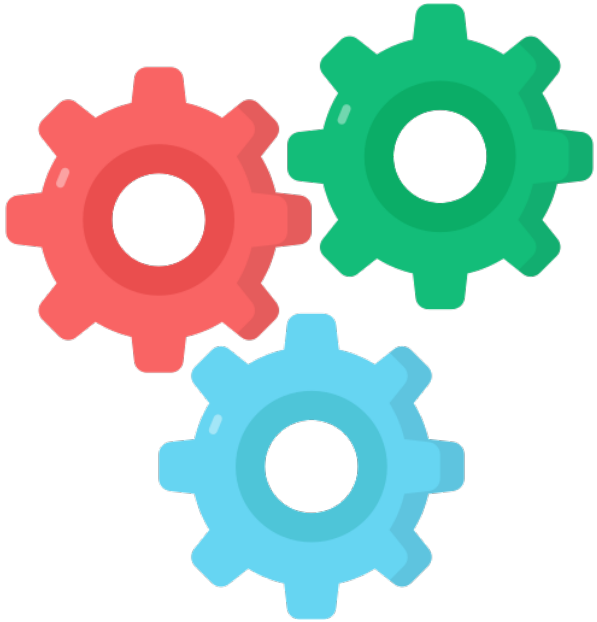
Funciones con parametros

Las funciones en JavaScript pueden recibir parámetros, que son valores que se pasan a la función al llamarla. Los parámetros permiten que una función acepte datos de entrada y los utilice dentro de su cuerpo para realizar operaciones o tomar decisiones.



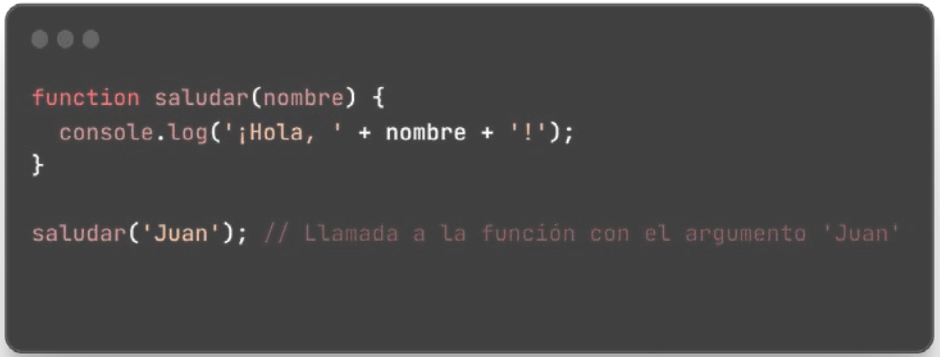
Funciones llamado

Cuando defines una función con parámetros, debes declarar los nombres de los parámetros dentro de los paréntesis en la declaración de la función. Estos nombres representan los valores que se pasarán a la función cuando se llame.



Funciones en acción

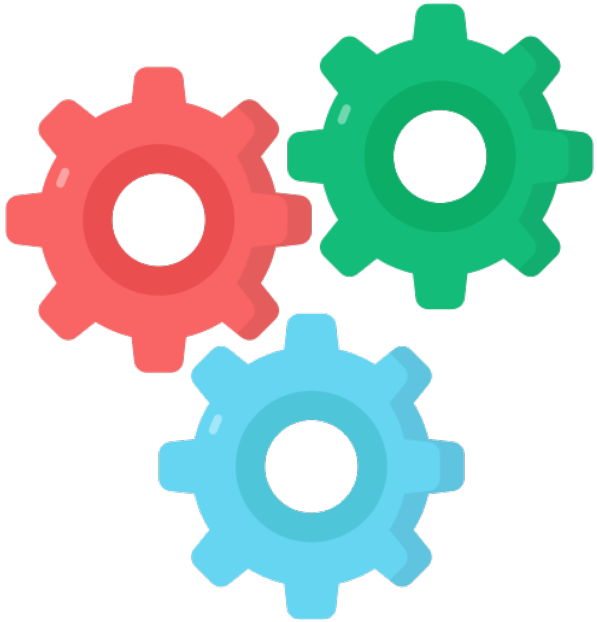
En este caso, la función saludar acepta un parámetro llamado nombre. Cuando se llama a la función con el argumento 'Juan', se imprime el saludo '¡Hola, Juan!', en la consola.



Funciones con retorno

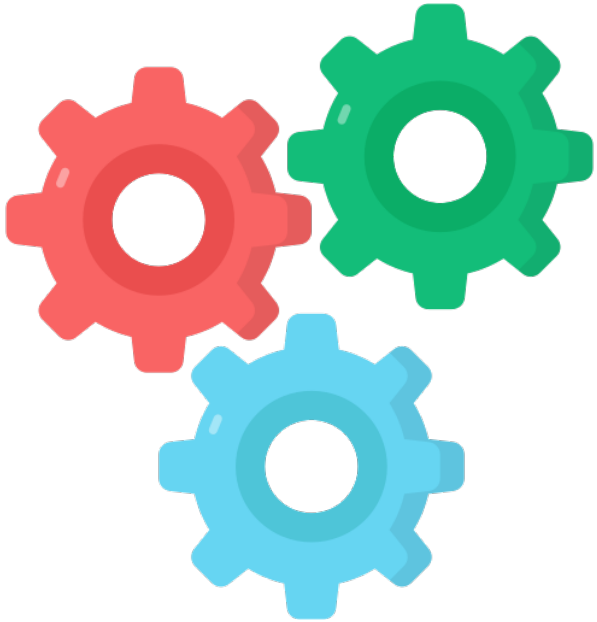
Funciones con retorno

Las funciones de retorno, también conocidas como funciones de devolución de llamada o "callback functions" en inglés, son funciones en JavaScript que se pasan como argumentos a otras funciones y se invocan dentro de esas funciones.



Funciones con retorno

Cuando una función recibe otra función como argumento y la utiliza en su lógica interna, se le llama función de retorno. Esto permite que la función externa pueda ejecutar la función interna en un momento adecuado, generalmente después de haber realizado ciertas operaciones o haber obtenido un resultado.



Funciones con retorno

Las funciones de retorno son especialmente útiles en situaciones en las que se necesita realizar tareas asincrónicas, como leer un archivo, realizar una solicitud HTTP o esperar a que ocurra un evento. En lugar de bloquear la ejecución del programa mientras se espera el resultado, se puede pasar una función de retorno para manejar el resultado una vez que esté disponible..

```
function realizarOperacion(a, b, callback) {
  let resultado = a + b;
  callback(resultado);
}

function mostrarResultado(resultado) {
  console.log("El resultado es: " + resultado);
}

realizarOperacion(5, 3, mostrarResultado);
```

Scope de una variable

Scoupe de una variable

El "scope" o alcance en JavaScript se refiere a la visibilidad y accesibilidad de variables, funciones y objetos en una determinada parte del código durante la ejecución del programa. En otras palabras, el scope determina dónde y cómo se pueden acceder a las variables en tu código.

```
function realizarOperacion(a, b, callback) {
  let resultado = a + b;
  callback(resultado);
}

function mostrarResultado(resultado) {
  console.log("El resultado es: " + resultado);
}

realizarOperacion(5, 3, mostrarResultado);
```

En JavaScript, existen dos tipos principales de scope: el scope global y el scope local.

Scoupe de una variable

Scope Global: Las variables declaradas fuera de cualquier función se consideran globales y están disponibles en todo el programa. Pueden ser accedidas y modificadas desde cualquier parte del código.

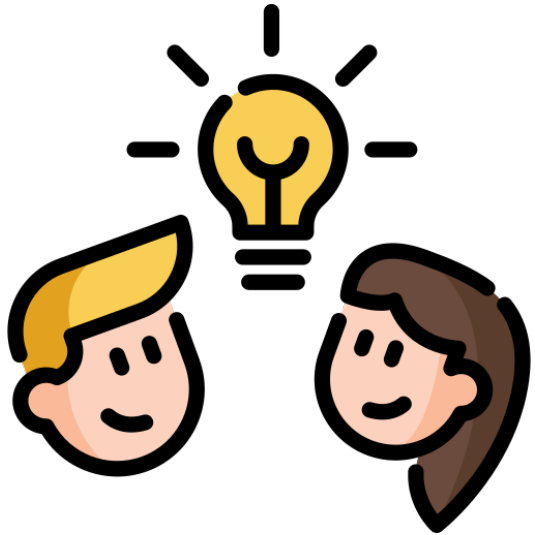
```
let nombre = "Juan"; // Variable global

function saludar() {
  console.log("Hola, " + nombre); // Acceso a variable global
}

saludar(); // Resultado: "Hola, Juan"
```

Imaginemos scope global

- Puedes imaginar el scope global como un "contenedor" que envuelve todo tu programa JavaScript.
- Visualmente, podrías representarlo como un círculo o un cuadro que abarca todo el código.
- Dentro de este contenedor global, todas las variables y funciones son accesibles desde cualquier parte del programa.



Scoupe de una variable

Scope Local: Las variables declaradas dentro de una función se consideran locales y solo son accesibles dentro de esa función, es decir, su alcance está limitado al bloque en el que fueron declaradas. Estas variables no están disponibles fuera de la función.

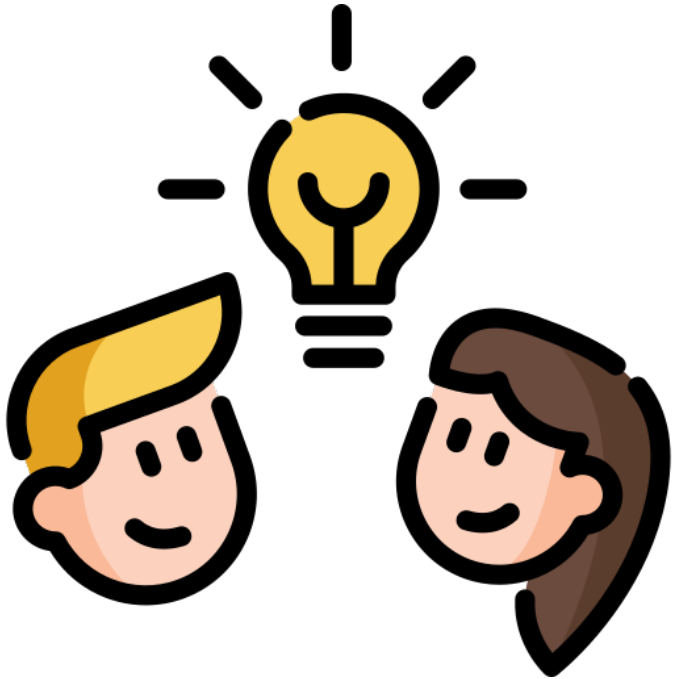
```
function saludar() {
  let nombre = "María"; // Variable local
  console.log("Hola, " + nombre); // Acceso a variable local
}

saludar(); // Resultado: "Hola, María"

console.log(nombre); // Error: la variable "nombre" no está definida en este scope
```

Imaginemos scope local

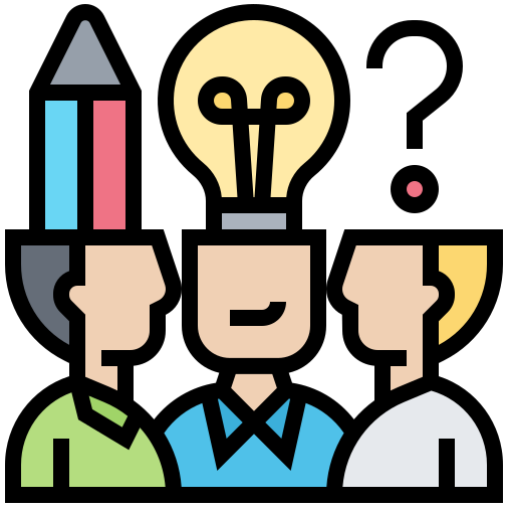
- Puedes visualizar el scope local como un "contenedor" más pequeño dentro del scope global.
- Cada función tiene su propio scope local, donde las variables declaradas dentro de la función son visibles.
- Puedes representar visualmente el scope local como un círculo o un cuadro más pequeño dentro del scope global.



A tener en cuenta

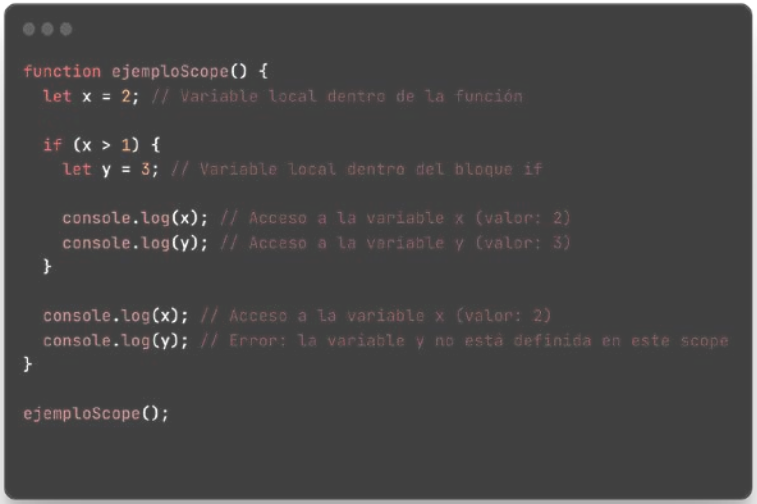
Es importante tener en cuenta que las variables locales en funciones anidadas también tienen un scope local y pueden ocultar (shadow) las variables con el mismo nombre en un scope superior.

El scope en JavaScript ayuda a evitar colisiones de nombres de variables y permite un mejor control sobre la privacidad y reutilización de código. Además, también existen scopes de bloque introducidos en versiones más recientes de JavaScript mediante el uso de las palabras clave let y const, que limitan la visibilidad de las variables a un bloque de código específico.



Scope de bloque

- Los scopes de bloque son introducidos por las palabras clave let y const en versiones más recientes de JavaScript.
- Puedes imaginar un scope de bloque como un "contenedor" que se aplica a un bloque de código delimitado por llaves {}.
- Este scope limita la visibilidad de las variables declaradas con let o const dentro del bloque en el que se encuentran.



```
function ejemploScope() {
  let x = 2; // Variable local dentro de la función

  if (x > 1) {
    let y = 3; // Variable local dentro del bloque if

    console.log(x); // Acceso a la variable x (valor: 2)
    console.log(y); // Acceso a la variable y (valor: 3)
  }

  console.log(x); // Acceso a la variable x (valor: 2)
  console.log(y); // Error: la variable y no está definida en este scope
}

ejemploScope();
```

¿Dudas o consultas?

¡Muchas Gracias!