# **Global Scope**

Se define cuando una variable está declarada de forma **global**, es decir, **fuera de una funcion o un bloque**.

**No es buena práctica declarar de forma global** ya que podemos declarar **dos** variables con el **mismo** nombre y **diferente** valor.

## **¿Qué es el SCOPE?**

Se puede definir como **el alcance que tienen nuestras variables dentro del documento**.

Es importante tener en cuenta que los **objetos** y las **funciones** también son variables, por lo tanto, **son afectadas por el** **scope**.

Para entender a la perfección como afecta el scope debemos tener en cuenta el “**funcionamiento de las variables**” que se pueden “declarar - asignar - declarar y asignar - redeclarar - reasignar”

Tenemos distintos tipos de Scope:

* **Global.**
* **Function.**
* **Block.**

# **Function Scope**

El **scope local de función** (function scope) es el entorno donde las variables locales solo se pueden acceder desde una función del programa.

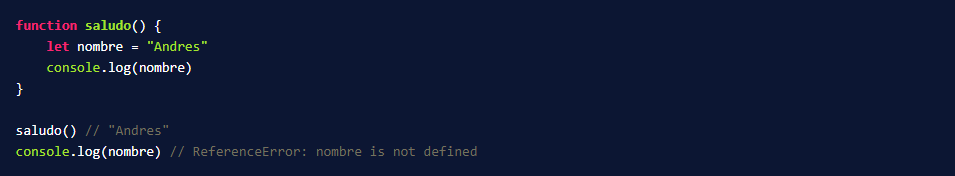
En este scope no importa que las variables sean declaradas con **var**, **let** o **const**. Sin embargo, ten presente que se puede redeclarar una variable con **var**, pero no con **let** y **const**.

## **Ejemplo utilizando scope de función**

Observa el siguiente código y piensa cuál será el resultado.

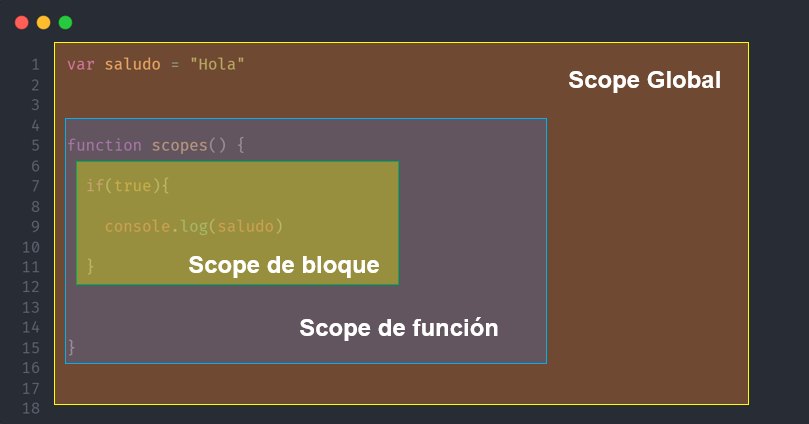


Primeramente, al invocarse la función saludo imprimirá "**Andres**" por consola, pero inmediatamente después existirá un error de referencia.



Esto sucede porque la variable nombre tiene un scope de función, por lo que solo se puede acceder dentro de la misma.

Según la cadena de scope, si existe una función dentro de otra función, la función hijo podrá acceder a las variables de la función padre, pero no en viceversa. Recuerda esto en el tema de los Closures.



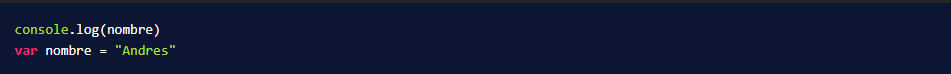
# **Block Scope**

El scope local de bloque es el entorno donde las variables locales únicamente pueden ser accedidas desde un bloque de código del programa. Un bloque de código es todo aquello que está dentro de los caracteres de llaves {}.

## **Hoisting en variables**

Hoisting es un término para describir que la declaración de variables y funciones son desplazadas a la parte superior del scope más cercano.

Mira el siguiente código y piensa cuál sería el resultado del console.log.



La respuesta es **undefined**, porque al hacer referencia a una variable que aún no está declarada, JavaScript crea esta variable y le asigna un valor de **undefined**. Es como hacer lo siguiente implícitamente.



De aquí el término de **Hoisting**, porque **eleva la declaración**. Pero esto solamente es cómo JavaScript interpreta el código, realmente las declaraciones siguen en el mismo lugar.



Este efecto solo ocurre con **var**, si ejecutamos el mismo código con **let** o **const**, mostrará un error de referencia.



## **¿Por qué “var” no tiene scope de bloque?**

En el scope de bloque, sí importa que las variables sean declaradas con **var**, **let** o **const**. Ya que **var** no tiene un scope de bloque. ¿Esto qué quiere decir?

Mira el siguiente código y piensa cuál sería el resultado del **console.log**.



La respuesta es primero "**hola**" y luego un **error de referencia**. Esto sucede por el **hoisting**, la declaración de la variable saludo se eleva fuera del bloque en un **scope superior**, que puede ser un **scope de función** o **global**. Sería como tener implícitamente lo siguiente.



Es por eso que **var** no tiene scope de bloque y se debe tener cuidado porque puede provocar errores en el código.

## **Ejemplo en un bloque de código**

Mira el siguiente código y piensa cuál sería el resultado de los **console.log**.



La respuesta es **2** y **2**, esto sucede nuevamente por el **hoisting**.



Para solucionar esto se utilizará **let** o **const**. La primera declaración tiene un **scope global**; y la segunda, un **scope de bloque**.



## **Ejemplo en un bloque de código de scopes diferentes**

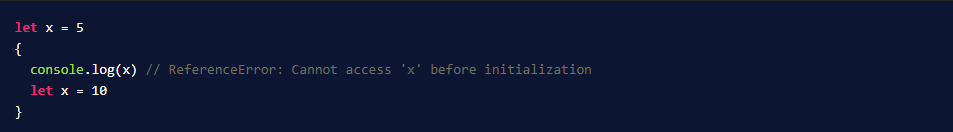
Hay una pequeña excepción cuando invocas una variable en un scope inferior del scope de la declaración de la variable con **let** y **const**, debes asegurarte que no exista una variable igual.

Mira el siguiente código y piensa cuál sería el resultado del **console.log**



La respuesta es **5**, porque si JavaScript no encuentra la variable **x**, sigue al **scope superior**.

Ahora, ¿qué pasaría si existe una variable **x** declarada con **let**, después del **console.log(x)**? Existirá un error de referencia, porque se está accediendo a una variable antes de su declaración.



## **Ejemplo en un bloque for**

Mira el siguiente código y piensa cuál sería el resultado del **console.log**.



La respuesta es **diez veces 10**, y sucede por el **hoisting**. La declaración de **i** se eleva hasta arriba de la función en el **scope de función**, por lo que cuando termine el ciclo este tendrá un valor de **10**. Es como tener implícitamente lo siguiente.



Para solucionar esto se utilizará **let** o **const**. La declaración del bloque **for** tiene un **scope de bloque**, en lugar de un **scope de función**.

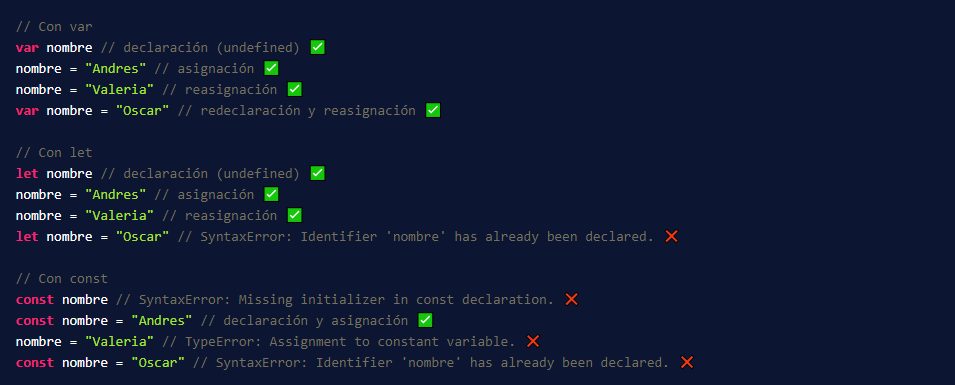


# **Reasignación y redeclaración**

La **redeclaración** es volver a declarar una variable, y la **reasignación** es volver a asignar un valor.

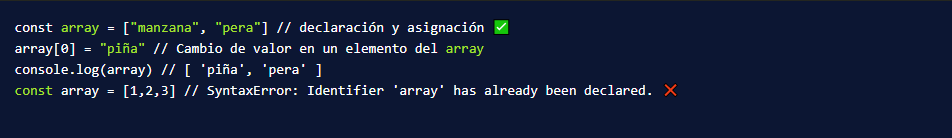
* Una variable declarada con **var** puede ser redeclarada y reasignada.
* Una variable declarada con **let** puede ser reasignada, pero no redeclarada.
* Una variable declarada con **const** no puede ser redeclarada, ni reasignada. Su declaración y asignación debe ser en una línea, caso contrario habrá un error.

Ejemplos de redeclaración y reasignación de variables



Estructuras de datos declaradas con **const**

Las estructuras de datos, como los **arrays** u **objetos**, declaradas con **const** pueden cambiar las referencias de sus elementos, a este concepto se lo denomina **mutabilidad**. Sin embargo, siguen las mismas reglas ya mencionadas, no puedes redeclarar ni reasignar una variable de estructura de datos.



En React, una librería de JavaScript, puedes utilizar declaraciones con **const** para el estado de un componente, porque, aunque cambie el valor, lo que sucede internamente es un re-renderizado y no una redeclaración.

# **Strict Mode**

El modo estricto es una funcionalidad que le permite al motor de JavaScript cambiar la manera en que ejecuta el código. En este modo, **se reduce las cosas que podemos hacer**, esto es bueno porque permite manejar errores que son poco perceptibles o que el motor de JavaScript sobreentiende y ayuda a su compilación para corregirlos.

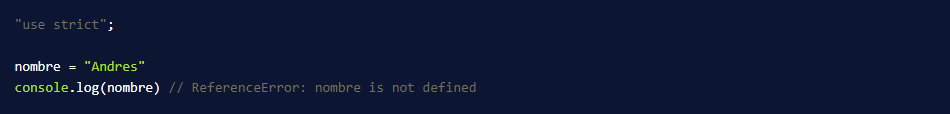
Este en el código colocando en la primera línea "**use strict**" para todo el archivo. También puede utilizarse en la primera línea de una función, pero no para un bloque en específico.

## **Ejemplo usando el modo estricto**

Si realizas el siguiente código que contiene una asignación sin una declaración, no va a ocurrir un error y el programa se va a ejecutar con normalidad.



En modo estricto, no te permitirá realizar esto y provocará un error.

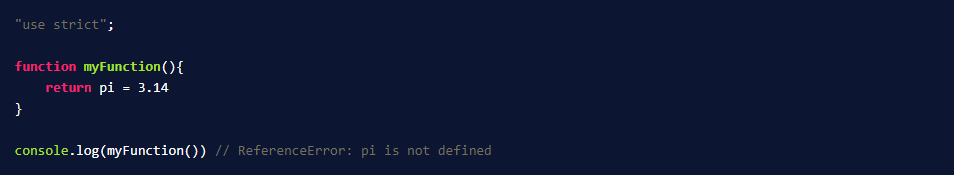


## **Ejemplo usando el modo estricto en una función**

Si realizas el siguiente código que retornas una variable no declarada, no va a ocurrir un error y el programa se va a ejecutar con normalidad.



En modo estricto, no te permitirá realizar esto y provocará un error.



# **¿Qué es un Closure?**

Un **closure** permite **acceder al scope de una función exterior desde una función interior**. En JavaScript, los closures se crean cada vez que una función se genera. A diferencia de otros conceptos como funciones, variables u otros, los closures no se utilizan todas las veces.

## **Construcción de un closure**

Para construir un closure necesitaremos los siguientes requisitos:

* Una función dentro de otra función.



* Una **variable** que se encuentre en la función con el **scope superior**, donde la función con **scope inferior** la utilice.



Invocar la función con **scope inferior** en otro scope del que fue escrita. Esto lo podemos hacer retornando la función entera y asignar la función de **scope superior** a una variable.



De esta manera obtendremos una función (scope inferior) que tiene una referencia a la variable que se encontraba en un scope superior, que a su vez recordará el contexto donde fue creada.

## **Ámbito léxico**

El ámbito léxico se refiere al **alcance de una variable**, siguiendo la cadena de scopes. Es decir, una variable puede ser accedida desde un **nivel inferior** hasta **uno superior**, pero no al contrario.



Podemos asignar la función contadora a una variable (**\_closure\_** y **\_closure2\_**). Dependiendo del valor que tenga como parámetro, cada variable cambiará el valor inicial por el ámbito léxico, por la que fue originada la variable acumulador que está vinculada con la función aumentar.

# **Practicando Closures**

Ahora que ya entiendes qué es un closure, te mostraré un ejemplo para emplearlo en un proyecto.

## **Cómo utilizar Closure para construir una alcancía**

Si creamos una alcancía de la siguiente manera, solamente mostrará el valor enviado, no guardará la información del dinero que le enviamos.



Si utilizamos un **closure**, entonces la variable en que estará en un scope más elevado que la función interior, recordará el ámbito léxico que fue creada. Por lo tanto, cada vez que invoquemos cada función, mostrará el dinero ahorrado en la alcancía.



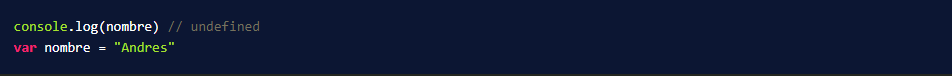
# **¿Qué es el Hoisting?**

**Hoisting** es un término para describir que las declaraciones de **variables** y **funciones** son desplazadas a la parte superior del scope más cercano, **scope global** o de **función**. Esto sucede solamente con las **declaraciones** y no con las asignaciones.

El código permanece igual, solo es una interpretación del motor de JavaScript. En el caso de las variables solamente sucede cuando son declaradas con var.

## **Hoisting en variables declaradas con var**

En el siguiente código, la respuesta del **console.log** es **undefined**, porque al hacer referencia a una variable que no está declarada aún, JavaScript crea esta variable antes de declararla y le asigna un valor de **undefined**.





## **Hoisting en scope de bloque**

Mira el siguiente código y piensa cuál sería el resultado del **console.log**.



La respuesta es primero "**hola**" y luego un **error de referencia**. Esto sucede por el hoisting, la declaración de la variable **saludo** se eleva fuera del bloque en un **scope superior**, que puede ser un **scope de función** o **global**.



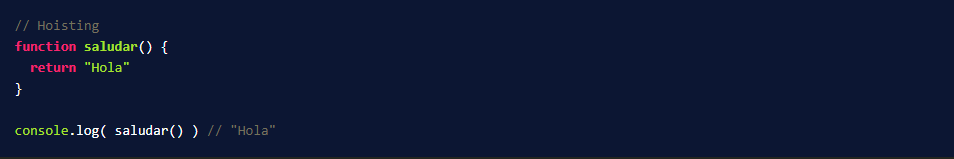
Es por eso que var no tiene scope de bloque, y se debe tener cuidado porque puede provocar errores en el código.

## **Hoisting en funciones**

Mira el siguiente código y piensa cuál sería el resultado del **console.log**.



La respuesta es "**Hola**", porque al invocar una función que no está declarada, JavaScript la **eleva** y por eso podemos invocar una función **antes** de declararla.



Pero, lo que realmente sucede es que JavaScript **guarda la función en memoria en la fase de creación de un contexto de ejecución**, no asigna **undefined** como con las variables.

## **Hoisting de variables dentro de una función**

El hoisting desplaza las declaraciones a la parte superior del scope más cercano, en el caso de una función **dentro** de otra función, seguiría el siguiente comportamiento.



JavaScript lo interpretaría como lo siguiente:

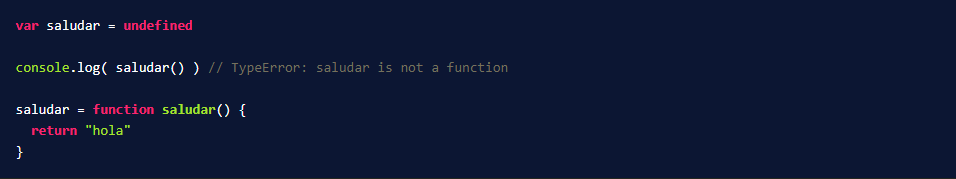


## **Hosting en funciones asignadas a variables**

Mira el siguiente código y piensa cuál sería el resultado del **console.log**.



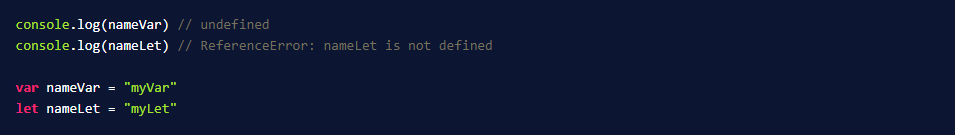
La respuesta es un **error de tipo** porque si asignas una función a una variable declarada con **var**, y la invocas antes declararla, la variable será de tipo **undefined** y no de función por el **hoisting**.



## **Hoisting con let y const**

Aunque te haya dicho que el **hoisting** solo ocurre con declaraciones con **var**, no es totalmente cierto. El hoisting hará que el intérprete de JavaScript eleve las declaraciones con **let** y **const** a la **Temporal Dead Zone**.

La **Temporal Dead Zone** es una región del código donde la variable está declarada, pero no es posible acceder a esta, provocando un error de tipo **ReferenceError**.



## **Buenas prácticas para usar hoisting**

* No utilices **var** en las declaraciones de variables.
* Escribe primero las funciones y luego su invocación.

# **Debugging**

**Debugging** es el término para solucionar bugs. Los bugs (“bichos” en inglés) son errores en la aplicación, saber cómo solucionarlos de manera eficiente es clave para tu desarrollo como profesional.

Todo navegador dispone de **Dev tools** o herramientas de desarrollador, que es un conjunto de características del código de la página web, una de estas es el **debugging**.

La consola del navegador es importante para ver qué está pasando con el código generado. La consola se muestra con la combinación de teclas F12 / Ctrl + Shift + I / Cmd+Opt+I o clic derecho e “Inspeccionar” en tu navegador preferido (de preferencia Google Chrome).

## **Código de ejemplo**

El código de ejemplo que se usará en las Dev Tools será el siguiente:



## **Palabra reservada debugger**

La palabra reservada **debugger** sirve para detener la ejecución del programa, pero solo funciona si el panel de las herramientas de desarrollo está abierto. Este panel te mostrará información sobre el código hasta la línea del debugger.

Ejecuta el código de prueba que contenga la palabra reservada **debugger** en la consola de tu navegador, puedes hacerlo en una página en blanco tan solo poniendo **about:blank** como una URL.

