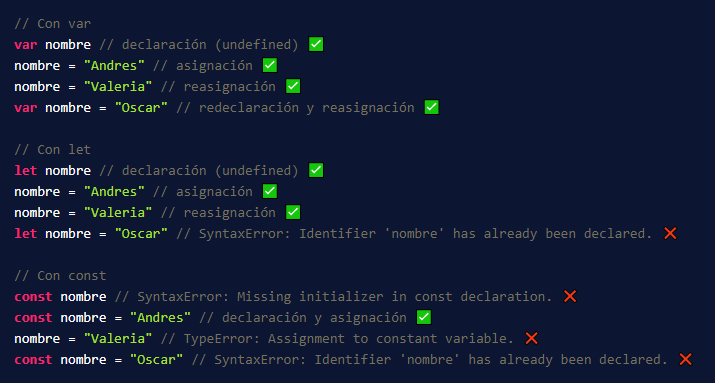
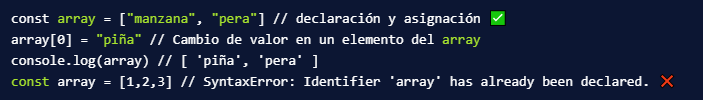
* **Reasignación y Re-declaración**



* **Const con Array y objetos:**

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**. Es decir que una **const** no se puede re-declarar ni reasignar, pero sí puedo usar métodos y editarle el contenido a la estructura de datos. Ej.:



**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:

nombre = "Andres"

console.log(nombre) // "Andres"

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

"use strict";

nombre = "Andres"

console.log(nombre) // ReferenceError: nombre is not defined

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.

**function** **myFunction**(){

**return** pi = 3.14

}

console.log(myFunction()) // 3.14

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

**function** **myFunction**(){

"use strict";

**return** pi = 3.14

}

console.log(myFunction()) // ReferenceError: pi is not defined

El modo estricto se aplica a un script completo o a funciones individuales. No se aplica a bloques entre corchetes {}; intentar aplicarlo en tales contextos no hace nada.

**CLOSURE**

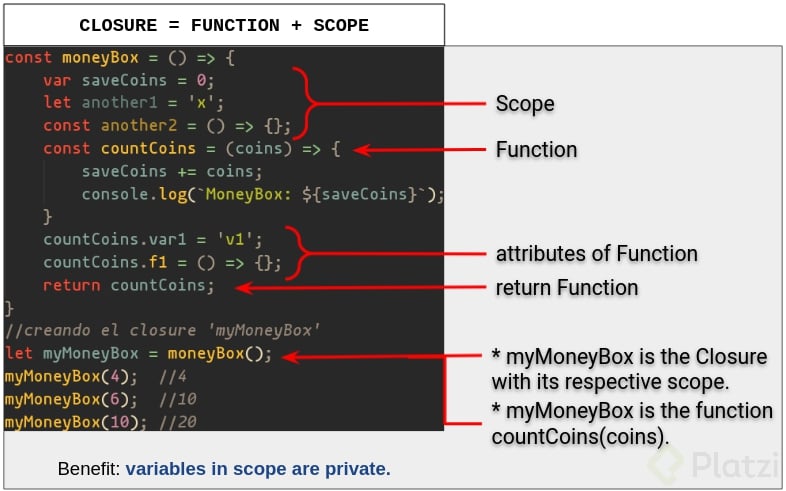
Un closure es la combinación entre una función y el ámbito léxico en el que esta fue declarada. Con esto, la función recuerda el ámbito con el cual se creó. Puedes entender los closures como: función interna + scope. Mira estos pasos:

Genera una función que retorna una función interna.

Esta función interna tiene un scope, el cual puede ser accedido únicamente por esta función, es decir, las variables, funciones, etc. definidas en el scope solo pueden ser accedidas por la función interna.

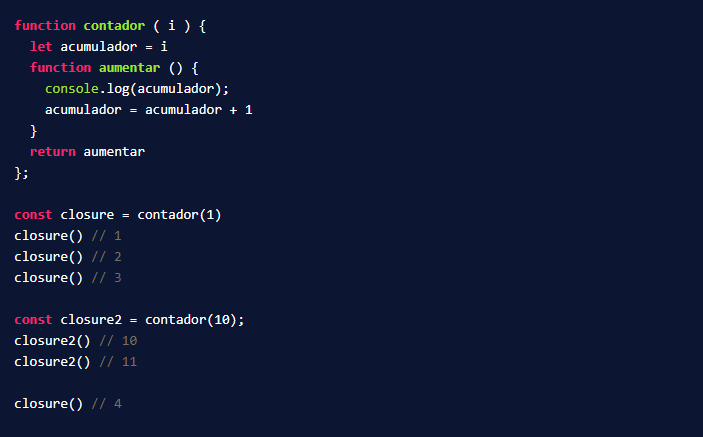
Como resultado, esta función interna retornada con su scope será nuestro closure.

Veamos la siguiente imagen para entenderlo mejor. En la imagen FUNCTION es la función interna de la cual hablamos.



Ambito léxico

El ámbito léxico se refiere al alcance de una variable siguiendo la cadena de scopes. Una variable se puede abordar desde un nivel inferior hasta uno superior, pero no al contrario.



**()**

Podemos asignar la función contador 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.

Ejemplo de closure

La mejor manera de entender un closure es por medio de un ejemplo práctico. Cuando declaramos una función dentro de nuestro global scope, estamos usando un closure.



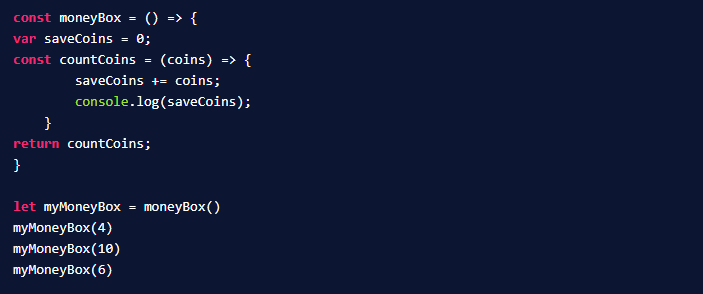
Los closures son básicamente cuando aprovechamos la habilidad de JavaScript de emplear las variables que están en el scope padre de nuestro bloque de código, por eso el global scope es un closure grande. El bloque myFunction puede utilizar TODAS las variables que están disponibles en el bloque inmediato anterior. Usando el ejemplo visto en el video:

Si tú declaras la variable saveCoins en el global scope, estarías usando el mismo principio que si emplearas la segunda función que escribe el profesor porque estás usando las variables que están en el scope padre.



Sin embargo, está mal visto modificar variables globales, por eso es que quieres crear variables dentro de un scope cerrado y que interactúen entre ellas. Entonces, declaras las variables que vas a usar dentro del scope padre del bloque que las va a modificar para que siempre pueda acceder a ellas.

Para eso originas un nuevo “global scope” ficticio que va a conservar todas las variables que tú quieras monitorear. Ahora mira las similitudes entre el código de arriba y el que está justo abajo de aquí:



¡Si remueves " const moneyBox = () => {} " serían exactamente las mismas líneas de código!



Lo que estás haciendo es simplemente bajar un nivel tu scope. Quieres que la función moneyBox regrese una función que estuvo declarada dentro de sí misma porque esa función tiene acceso a ese scope que ya no va a existir para que alguien más lo utilice, solamente lo podrá emplear la función countCoins.

Al guardar el resultado de moneyBox (countCoins) en otra variable estás generando el ámbito léxico que menciona el profesor, necesario para no dejar morir ese scope.