¿Qué es un JavaScript Closure? , esta es una pregunta clásica en las formaciones avanzadas de JavaScript .Vamos a introducir el concepto y ver en que situaciones se puede utilizar de forma práctica. Para ello vamos a partir del siguiente bloque de código.

```
var mensaje="hola";

function f1(){

 var otroMensaje="hola2";

 function f2() {

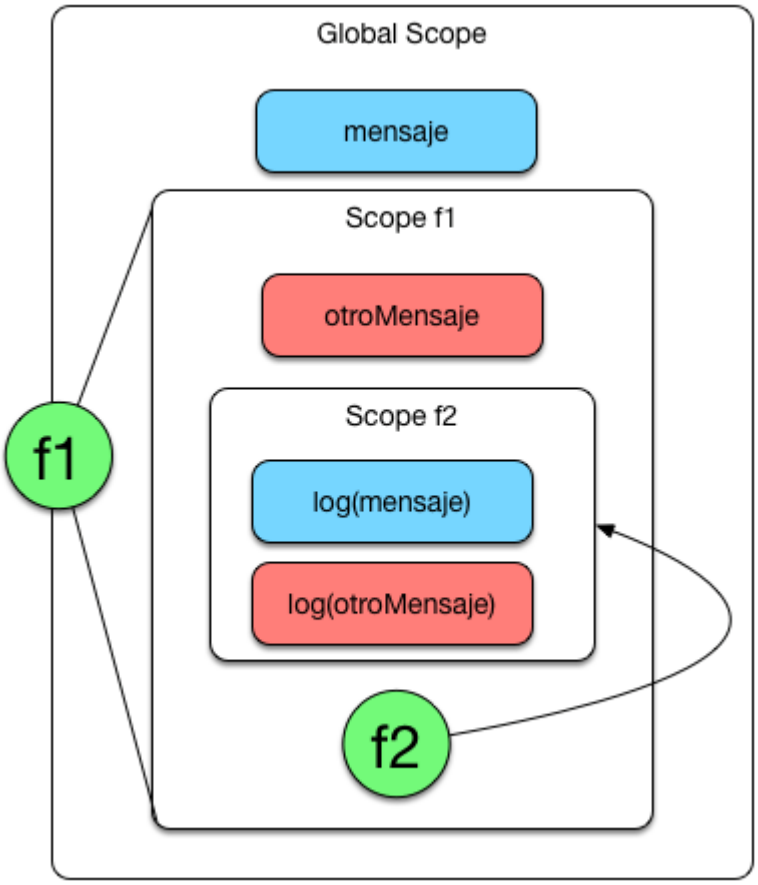
 console.log("mensaje:"+mensaje);
 console.log("otromensaje:"+otroMensaje);

 }

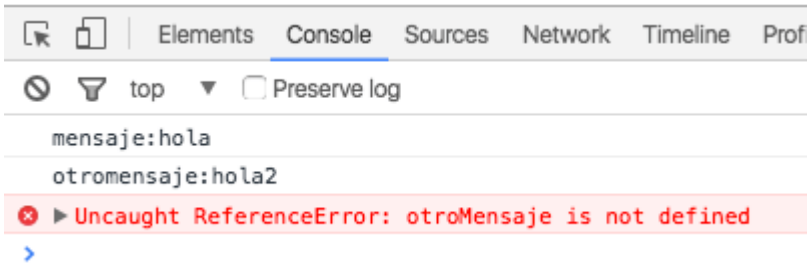
 f2();
}

f1();
// no existe fuera de ambito
console.log(otroMensaje);
```

El código dispone de dos funciones anidadas, recordemos que cada función en JavaScript define un Scope , por lo tanto disponemos del Scope de f1 , el scope de f2 y el Scope global:



Invocamos a f1() y nos imprime por pantalla el siguiente resultado:



La función f1 llama a la función f2 y nos imprime los mensajes sin problemas .
Evidentemente desde el scope global no tenemos acceso a la variable otroMensaje ya que se

encuentra dentro del scope de f1 . Vamos a realizar una modificación en el código.

```
var mensaje="hola";
//genero una referencia a f2()
var referencia;
function f1(){

 var otroMensaje="hola2";

 function f2() {

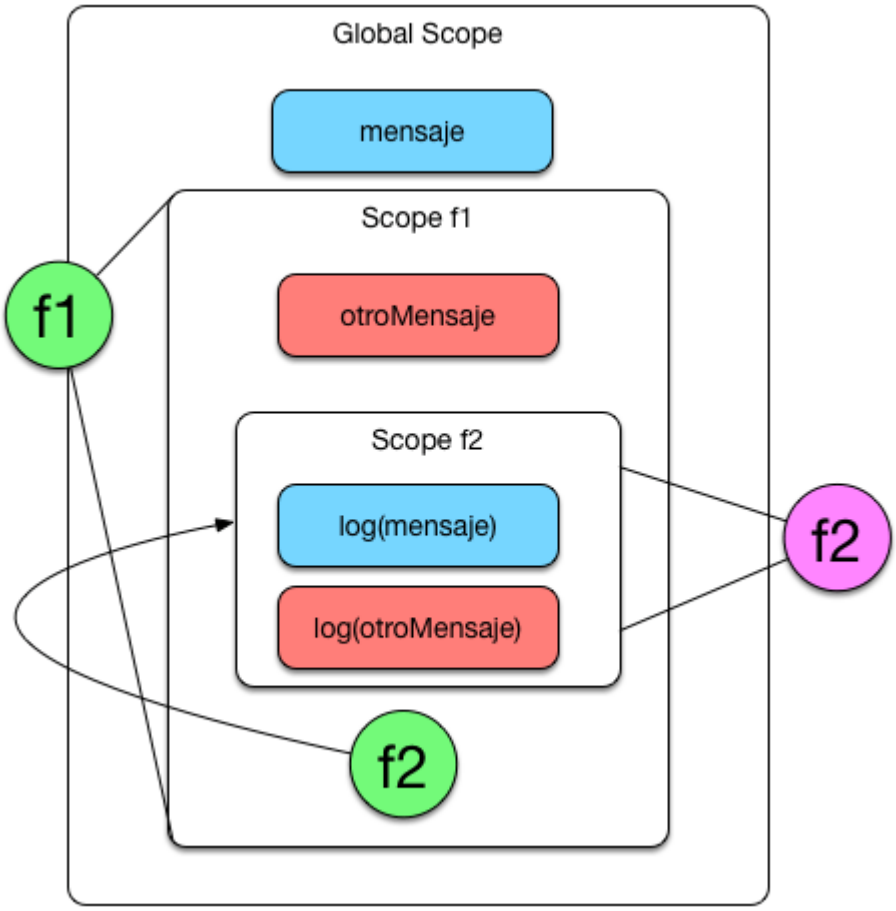
 console.log("mensaje:"+mensaje);
 console.log("otroMensaje"+otroMensaje);

 }


 f2();
 referencia=f2;
}

f1();
referencia();
```

Acabamos de crear una variable que referencia a la función f2 desde el scope global.

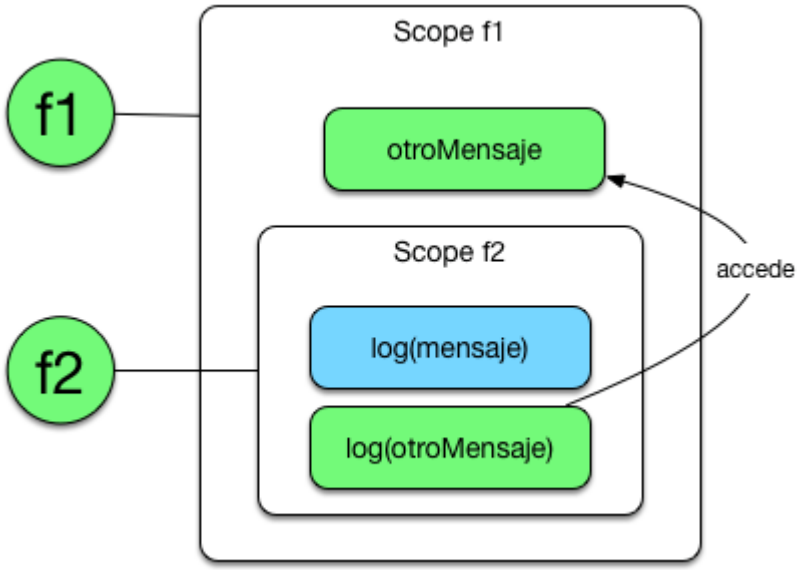


Es momento de invocar la la función f1 y a la referencia que acabamos de crear y ver que resultado nos imprime la consola.

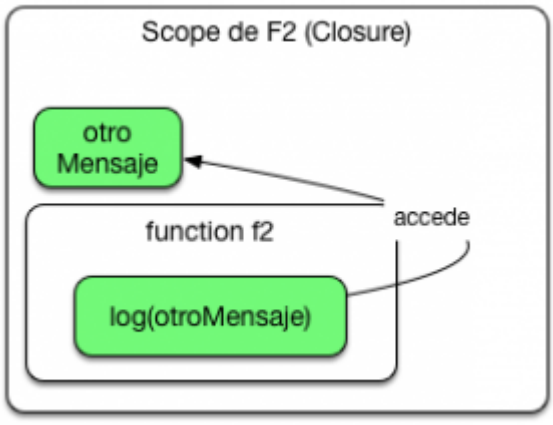


Los dos primeros mensajes son sencillos de entender imprimen los valores de las variables .

Sin embargo el segundo bloque de mensajes genera más dudas ya que invocamos directamente a f2 y la función f1 ya termino su ejecución por lo tanto la variable “otro mensaje” debería imprimir undefined. ¿Qué esta pasando?.



JavaScript se da cuenta de que la función f2 debe acceder a datos que están en el Scope de la función f1 y mantiene en memoria esta variable (creando un scope) aunque no accedemos de forma directa a ella sino a través de f2.



Por lo tanto el Scope de f2 incluirá la variable que tenemos en f1. ¿Hasta cuando vivirá la variable otroMensaje.? . Su ciclo de vida se ampliará hasta que la función f2 termine de ejecutarse.

JavaScript Closure y su uso

Acabamos de ver como se construye un JavaScript Closure , lo difícil muchas veces es entender para que podemos usarlo. Vamos a crear un bucle for para entender mejor los conceptos.

```
for (var i = 1; i <= 5; i++) {

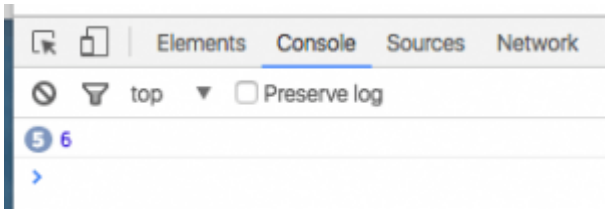
 setTimeout(function retraso() {

 console.log(i);

 }, 2000);


}
```

Este bucle debiera imprimir 1,2,3,4,5 por pantalla cada dos segundos . Sin embargo el resultado es muy diferente , al cabo de 2 segundos nos imprime :



Esto se debe a que la variable i se incrementa hasta 6 y luego después de dos segundos el setTimeOut se ejecuta 5 veces imprimiendo 6 por pantalla. ¿Cómo podemos solventar este problema? . Vamos a crearnos un JavaScript Closure generando un nuevo ámbito o scope.

```
for (var i = 1; i <= 5; i++) {

 (function (variable) {

 setTimeout(function retraso() {

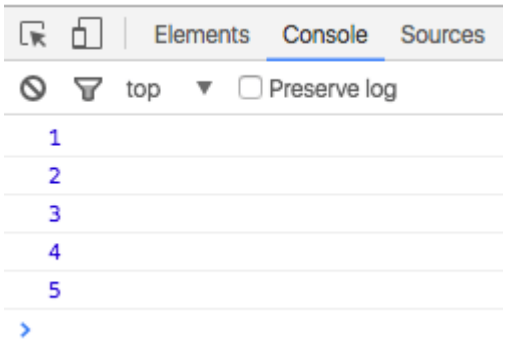
 console.log(variable);

 }, 2000);

 }) (i);

}
```

Ahora la variable i es pasada de la función en formato IIFE a la función retraso . Al usar esta función la variable i , esta variable se mantendrá con vida en cada iteración. El resultado será mucho más cercano al que queremos.



Por último podemos cambiar el tiempo de TimeOut para cada iteración de tal forma que se ejecute cada dos segundos.

```
for (var i = 1; i <= 5; i++) {

 (function (variable) {

 setTimeout(function retraso() {

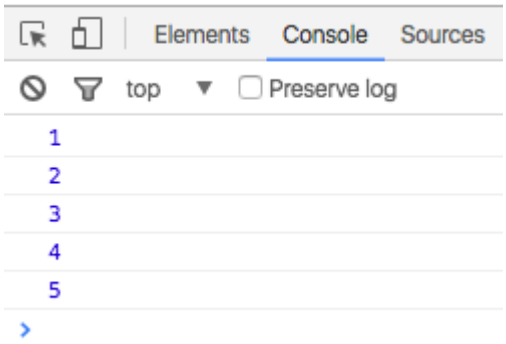
 console.log(variable);

 }, i*2000);

 }) (i);

}
```

Veremos pasar cada dos segundos una iteración.



Es lo que necesitábamos , sin los JavaScript closures no habría sido posible. Es aquí donde vemos que JavaScript puede llegar a ser muy complejo.

Otros artículos relacionados : JavaScript Prototypes JavaScript Stream vs Promises , JavaScript NameSpace