

María Sofía Álvarez López – 201729031

Ejercicio 1 – JavaScript

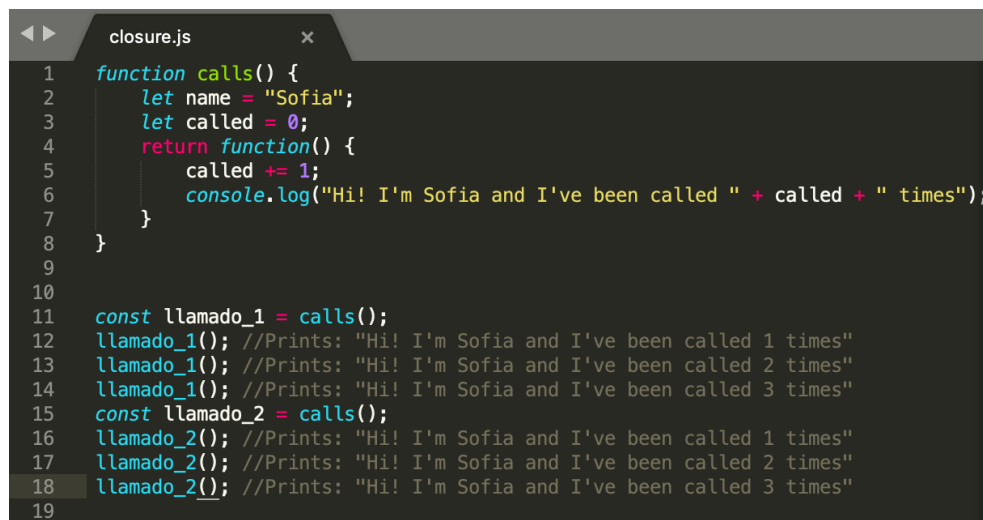
Nota: Todos los ejemplos asociados a este ejercicio los encuentra en el siguiente repositorio:

<https://github.com/sofiaalvarezlopez/Laboratorios-Web/tree/main/Semana%202>

1. ¿Qué es un closure?

Un *closure* es una función que tiene acceso al *scope* (conjunto de variables que pueden ser accedidas en un bloque de código) de una (o varias) función(es) padre, o superiores – incluso si esta función padre ya ha sido cerrada. Una función que sea *closure* recordará el conjunto de variables a las que puede acceder, incluso si dicha función se invoca en otro lugar del código.

Un ejemplo se presenta a continuación:



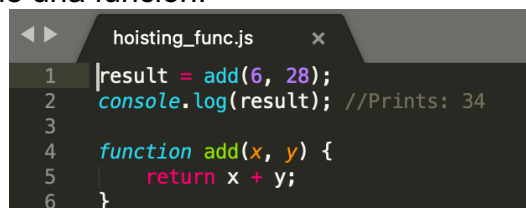
```
closure.js
1 function calls() {
2   let name = "Sofia";
3   let called = 0;
4   return function() {
5     called += 1;
6     console.log("Hi! I'm Sofia and I've been called " + called + " times");
7   }
8 }
9
10
11 const llamado_1 = calls();
12 llamado_1(); //Prints: "Hi! I'm Sofia and I've been called 1 times"
13 llamado_1(); //Prints: "Hi! I'm Sofia and I've been called 2 times"
14 llamado_1(); //Prints: "Hi! I'm Sofia and I've been called 3 times"
15 const llamado_2 = calls();
16 llamado_2(); //Prints: "Hi! I'm Sofia and I've been called 1 times"
17 llamado_2(); //Prints: "Hi! I'm Sofia and I've been called 2 times"
18 llamado_2(); //Prints: "Hi! I'm Sofia and I've been called 3 times"
19
```

En este caso, la función retornada en la línea.4 es un closure de la función *calls*. Note que esta función retornada tiene acceso a la variable *name* (Sofía) de la función padre y a la variable *called*. Cuando se llama a la función en otra parte del código, se recuerda el conjunto de variables a las que puede acceder (tanto *name* como *called*), y puede incrementar en uno el contador de *called* al recordar que puede acceder a dicha variable.

2. ¿Qué es hoisting?

El hoisting es el comportamiento de JavaScript que mueve todas las declaraciones al inicio del scope de declaración. En este caso, las **funciones** (pero no las expresiones de funciones) y las variables definidas con la palabra **var** (pero no las que se declaran con let o const) pueden ser invocadas antes de ser declaradas. Este comportamiento no es usual de otros lenguajes de programación

Veamos el ejemplo de una función:



```
hoisting_func.js
1 result = add(6, 28);
2 console.log(result); //Prints: 34
3
4 function add(x, y) {
5   return x + y;
6 }
```

En este caso, la función `add(6, 28)` es llamada (invocada) antes de ser declarada gracias al comportamiento de hoisting.

Para una variable declarada con **var**:

```
hoisting_var.js
// Invocar la variable antes de declararla (con var).
function funciona(){
  x = 5;
  console.log('La variable es:', x);
  var x;
}

funciona(); //Prints: La variable es: 5
```

En este caso, la variable es invocada antes de ser declarada. No obstante, al ser declarada con **var**, funciona perfectamente.

En el caso en que se use **let** o **const** (como se muestra en el ejemplo a continuación), el compilador arroja un error de sintaxis (falta inicialización en la declaración de **const**), puesto que el hoisting no está permitido para este tipo de variables.

```
hoisting_not_works.js
1 // No funciona con const o con let
2 function noFunciona(){
3   y = 'No funciona';
4   console.log(y);
5   const y;
6 }
7
8 noFunciona(); // SyntaxError: Missing initializer in const declaration
```

3. Explique los distintos contextos en que puede usarse el objeto **this**

En JavaScript, el contexto se refiere a un objeto. Dentro de un objeto, la palabra reservada **this** hace referencia a dicho objeto (a sí mismo). El objeto **this** puede usarse en 4 contextos: default binding, implicit binding, explicit binding, new binding y lexical binding.

- Default binding: También llamado invocación directa, corresponde a la forma en que funciona **this** si se usa directamente en una función. Si no se usa un punto, o las funciones `call()` o `apply()`, **this** será en este caso el objeto `window` (si se está en el navegador) o el objeto `global` (si se está en Node).

De esta forma, el siguiente fragmento de código:

```
defaultBinding.js
1 function helloWorld(){
2   console.log('Hello, world!', this);
3 }
4
5 helloWorld();
```

Al ser ejecutado en node, imprimirá el objeto `global`, que es a lo que corresponde **this** en este contexto:

```

Hello, world! <ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  performance: Performance {
    nodeTiming: PerformanceNodeTiming {
      name: 'node',
      entryType: 'node',
      startTime: 0,
      duration: 36.15451765060425,
      nodeStart: 2.145482063293457,
      v8Start: 3.141098976135254,
      bootstrapComplete: 27.59179162979126,
      environment: 14.724379062652588,
      loopStart: -1,
      loopExit: -1,
      idleTime: 0
    },
    timeOrigin: 1662133579385.412
  },
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}

```

- Implicit binding: También llamado invocación de método, corresponde al caso en que se invoca al método de un objeto, tal que **this** corresponde a dicho objeto invocado. Dicho en otras palabras, **this** será lo que haya a la izquierda del punto de invocación del método. De esta forma:

```

implicit_binding.js
1  const animal = {
2    species: "Sea lion",
3    saySpecies: function (language) {
4      if (language === "English") {
5        console.log(`I am a ${this.species}`);
6      }
7      else {
8        console.log(`Soy un ${this.species}`);
9      }
10   }
11 };
12
13
14 animal.saySpecies("English"); //Prints: I am a Sea lion.
15

```

En el contexto del ejemplo mostrado previamente, siendo **this** lo que hay a la izquierda del punto de invocación, en este caso se tiene **this** como el objeto animal. Invocando el método saySpecies() sobre el objeto, se tiene que en dicho contexto, **this** será el animal (león marino) en cuestión.

- **Explicit binding:** Es decir, por invocación indirecta, ocurre cuando quiere invocarse un método en particular (que pertenece a un objeto A, con una propiedad X) sobre otro objeto B que tenga la misma propiedad que X. En este caso, se puede invocar el método para el objeto B usando las funciones `call()` o `apply()` sobre el método deseado del objeto A. En el contexto del ejemplo del león marino (objeto A), supongamos que tenemos un objeto delfín. Puedo invocar el método `saySpecies()` para B usando la invocación indirecta por medio de `call()` o `apply()`. De esta forma:

```

explicit_binding.js
1  const animal = {
2    species: "Sea lion",
3    saySpecies: function (language) {
4      if (language === "English") {
5        console.log(`I am a ${this.species}`);
6      }
7      else {
8        console.log(`Soy un ${this.species}`);
9      }
10   }
11 };
12
13 const dolphin = {species: "Dolphin"};
14
15 animal.saySpecies.call(dolphin, "English"); //Prints: I am a Dolphin.
16 animal.saySpecies.apply(dolphin, ["English"]); //Prints: I am a Dolphin.
17

```

Como se ve, en este caso, se usa la función `call` o `apply` (la diferencia es que `apply` requiere que los argumentos de la función sean pasados como arreglo) para el objeto B (el delfín) sobre el método (`saySpecies`) del objeto A (el animal – león marino).

En este contexto, **this** hace referencia al objeto sobre el cual se aplica el método `call()/apply()`. Es decir, el león marino.

- **New binding:** Ocurre cuando hay instanciación de objetos. En este caso, supongamos que se tiene un constructor para un objeto. Sea este un animal, como se verá más adelante en el ejemplo. Al instanciar un nuevo objeto, se crea un objeto vacío usando la palabra reservada **new** que invoca a la función constructora del objeto (en este caso, el animal) como valor de **this**. Lo que sucederá es que **this** terminará siendo el nuevo objeto instanciado. En otras palabras, se agrega al objeto la propiedad **this**. Veamos el ejemplo:

```

new_binding.js
1  function Animal(species) {
2    this.species = species;
3  }
4
5  const seaLion = new Animal("Sea Lion");
6  console.log(seaLion.species); //Prints: Sea Lion

```

Aquí, se está instanciando un nuevo objeto de tipo `Animal`, con el nombre de especie león marino. En este caso, el crear el objeto con **new** hace que **this** sea este nuevo objeto instanciando (el león marino).

- **Lexical binding:** Funciona cuando se utilizan arrow functions. Debido a que las arrow functions se ejecutan en el mismo contexto en que fueron creadas, el objeto **this** seguirá refiriéndose al mismo objeto, así se esté dentro de otra función. Veamos un ejemplo:

```

lexical_binding.js
1  const animal = {
2    name: "Señor Cinco",
3    species: "Sea Lion",
4    residence: "Vancouver aquarium",
5    introduce: function () {
6      const placeOfLiving = () => {
7        console.log(`I live in the ${this.residence}`);
8      };
9      console.log(`Hi! My name is ${this.name} and I am a ${this.species}`);
10     placeOfLiving();
11   },
12 };
13
14 animal.introduce();
15 // Prints: Hi! My name is Señor Cinco and I am Sea Lion
16 //          I live in the Vancouver aquarium

```

En este caso, el lexical binding está ocurriendo en el contexto de la arrow function definida en la línea 6. En la línea 7, **this** seguirá haciendo referencia al animal (del scope de la función externa), debido a que las arrow functions se ejecutan en el mismo contexto en que fueron creadas. Entonces, no se cambia el contexto de ejecución. Esto NO ocurre si no se tiene una arrow function, debido a la forma en la que funciona JavaScript.

Para resumir, el objeto **this** se puede usar en varios contextos. En el default, corresponderá al objeto window (en un navegador) o al objeto global (en node). En el implicit binding, hace referencia al objeto que invoca a un método en particular (es decir, lo que va antes del punto de invocación). En explicit binding, hace referencia al **objetoX** sobre el que se aplica un método a través de call o apply métodos call() o apply(). Es decir objetoA.metodo.call(**objetoX**, param) u objetoA.metodo.apply(**objetoX**, [params]). En new binding, hace referencia a un objeto nuevo que ha sido instanciado, al invocar a la función constructora usando new. Finalmente, en lexical binding, para funciones arrow internas, **this** hace referencia al mismo objeto del contexto exterior de la función arrow.

4. Arrow function para calcular el MCD

```

arrow_punto4.js
1  // Esta funcion recibe dos numeros enteros positivos a,b y retorna su MCD.
2  const mcd = (a, b) => (a === b) ? a : (mcd(Math.min(a, b), Math.abs(a - b)));
3  console.log(mcd(45, 100)); //Prints 5

```