CLASE 02 | JS AVANZADO I

POR ALEJO BENGOECHEA

JAVASCRIPT AVANZADO I

▼ SINGLE THREADED Y SINCRÓNICO

Un **thread** (o hilo de ejecución) es la secuencia de instrucciones más pequeña que puede ser manejada por un *planificador de recursos* (se encarga de repartir el tiempo disponible de los recursos del sistema entre todos los procesos).

JavaScript es **Single Threaded** y **Sincrónico**, es decir que sólo puede hacer un sólo comando o instrucción en cada momento y que lo hace en orden, empieza la instrucción siguiente cuando termina la anterior.

▼ SYNTAX PARSER

Lee el código línea por línea y determina lo que hace cada parte. También chequea si la gramática es correcta o no. El ***Syntax Parser*** es el intérprete entre tu código y la computadora. Traduce tu código a un lenguaje que la máquina puede entender.

▼ LEXICAL ENVIROMENT

El ***Lexical Environment*** tiene que ver con *dónde* están declarados ciertos statements o expresiones en tu código. No será lo mismo hacerlo en un lugar que en otro.

```
function hola() {
  var foo = "Hola!";
}
var bar = "chao!";
```

Por ejemplo, para el interprete las dos declaraciones de variable del arriba tendrán signicados muy distintos. Si bien la operación es igual en los dos (asignación) al estar en lugares distintos (una dentro de una función y la otra no) el interprete las parseará de forma distinta.

▼ GLOBAL ENVIROMENT

El **Global Enviroment** es la ventana “padre” del código. El la parte más externa, la que envuelve a todo el código. Dentro de este contexto podremos encontrar otros contextos.

▼ EXECUTION CONTEXT

El contexto de ejecución contiene información sobre **QUÉ** código se está ejecutando en cada momento. Esto se da en las *functions*.

```
var sayHello = "Hola Mundo!";           //GLOBAL CONTEXT

function persona() {                    //EXECUTION CONTEXT
  var first = "Maico";
  var last = "Rosa";

  function firstName() {                //EXECUTION CONTEXT
    return first;
  }

  function lastName() {                 //EXECUTION CONTEXT
    return last;
  }
}
```

Dentro del ***Execution Context*** existen distintos elementos. Estos son

Código | Global Object | ‘This’ | Outer Enviroment

▼ HOISTING

Hoisting es un comportamiento de JavaScript en el que “mueve las declaraciones al principio”.

Lo que sucede aquí es que antes de ejecutar el código, JavaScript le da una primera lectura para “tomar nota” de todas las variables (**var, —let y const NO —**) y todas las functiones (**function**) que tenga el código en su contexto global. Así podrá tener una referencia de la memoria a utilizar.

```
bar();
console.log(foo);

var foo = "Hola, me declaro";
function bar() {
  console.log("Soy una función");
}
```

En el ejemplo de arriba, estamos ejecutando una función y una línea de texto antes de declarar cualquiera de las dos. Cuando se realiza la etapa de ***hoisting*** se distingue una variable (foo) y una función (bar). Aquí las moverá al principio.

```
//var foo = undefined;
//function bar() {
//  console.log("Soy una función");
//}

bar();
console.log(foo);

var foo = "Hola, me declaro";
function bar() {
  console.log("Soy una función");
}
```

Todas las variables declaradas no estarán definidas en el hoisting. Las funciones declaradas como variables tampoco. Solo las funciones declaradas como *statement* serán reconocidas como una función completa.

▼ CREACIÓN Y EJECUCIÓN

Cuando el intérprete inicia el corrido del código entra en *execution context* e inicia dos etapas. La primera (**creation phase**) y la segunda (**execution phase**). En la ***creation phase*** suceden varios procesos. Los dos más importantes son el *hoisting* y la creación del nuevo entorno. En la ***execution phase*** se ejecuta el código.

▼ EXECUTION STACK

El ***execution stack***, se produce el fenómeno de “pila” o “last in first out”.

▼ GLOBAL OBJECT

El ***Objeto Global*** es creado en la primera fase de creación del **global enviroment**. A este objeto le asigna la variable **This.** En Google Chrome, la palabra *window* es el objeto global, y *this* es su referencia. Este objeto proporciona variables y funciones que están disponibles en cualquier lugar.

En el primer recorrido del código se crea un **Global Execution Context**. Aquí se comenzaran a hacer la etapa de creación y la de ejecución.

```
function b() {
  console.log("B!");
```

```
}

function a() {
  b();
}

a();
```

En este ejemplo vemos que primero entramos en fase de creación. Cuando se produce el *hoisting* se encuentran dos funciones definidas. Luego se crea el primer entorno que es el global. Aquí entramos a la segunda etapa que es de ejecución. Se ejecutará el código, y su primera movilización es la ejecución de la función A.

Como el intérprete se encuentra con una función que se quiere ejecutar, inicia el ***execution context*** de A(). Comienza la etapa de creación y *hoisting*. Como dentro de la función no hay variables u otras funciones, el hoisting queda nulo. Aquí crea el entorno de A(). Posteriormente, se ejecuta el código, ejecutando a su vez la función B().

Aquí se repite el proceso. Primero se hace un *hoisting* que queda nulo, luego se crea el entorno, y finalmente se ejecuta la función.

Es importante señalar que el entorno de A() se cerrará sólo cuando se finalize el entorno y ejecución de la función B(). Posteriormente que finaliza el entorno de A() se finaliza el código y termina con el contexto global.

▼ **COMENTARIOS SOBRE LOS CONTEXTOS**

Los contextos o *enviroments* del bloque global y de cada función mantienen una relación unilateral, donde las *inner functions* podrán tomar variables externas, pero no así al revez.

```
var foo = "Hola amigos";

function persona() {
  var saludo = "Qué tal muchachos";
}

function otraPersona() {
  var saludo = "Qué tal muchachos";
```

En este caso, nosotros podremos utilizar la variable **foo** en el contexto global y en las funciones. Pero las variables **saludo** solo pueden utilizarse en su función propia. Cabe destacar que, aunque ambas variables se llamen igual y tengan el

mismo valor, no se pisaran y ambas son completamente diferentes, ya que están en contextos distintos. Es decir, si yo cambiara la variable **saludo** de alguna de las dos funciones, esta no cambiaría en la otra.

▼ RECORRIDO POR ENTORNOS

Si recorremos el siguiente código la terminal mostraría lo siguiente.

```
var global = 'Hola!';

function a() {
  console.log(global);
  global = 'Hello!';
}

function b(){
  var global = 'Chao';
  console.log(global);
}

a();                          // 'Hola!'
b();                          // 'Chao'
console.log(global);          // 'Hello'
```

▼ SCOPE

El **Scope** de una variable hace referencia al lugar donde esta va a vivir , o podrá ser accesible. Podríamos decir también que **Scope** es el alcance que determina la accesibilidad de las variables en cada parte de nuestro código.

Cada contexto maneja sus propias variables, y son independientes de los demás. Justamente por eso, podemos usar los mismos nombres de variables dentro de funciones que creamos sin que *pisen* las demás.

También sabemos que podemos acceder a una variable declarada en el contexto global dentro de una función. Esto se debe a que JavaScript primero busca una variable dentro del contexto que se está ejecutando, si no la encuentra ahí, usa la referencia al `outer context` para buscarla dentro de ese contexto.

```
var global = 'Hola!';

function b(){
  var global = 'Chao';
  console.log(global); // Chao
  function a() {
    // como no hay una variable llamada global en este contexto,
    // busca en el outer que es scope de b;
```

```
    console.log(global); //Chao
    global = 'Hello!'; // cambia la variable del contexto de b()
  }
  a();
}

//a(); Ya no puedo llamar a a desde el scope global, acá no existe.
b();
console.log(global); // 'Hola!
```

La variable `global` está definida en dos `scopes` distintos, uno es el `scope global` y el otro es el `scope` de la función `b`, esto quiere decir que, a pesar de tener el mismo nombre, estas dos variables son distintas.

Si el intérprete llega el scope Global sin encontrar la variable, entonces va a tirar un error.

▼ TIPOS DE DATOS

Existen dos tipos de lenguajes en programación. Los de tipado **dinámico** y los de tipado **estático**. JavaScript es de tipado dinámico. En este lenguaje no es necesario declarar el tipo de dato cuando creamos una variable. En el tipado estático, como en JAVA, cuando creamos una variable tendremos que definir qué tipo de dato será, y eso no se podrá cambiar.

DATOS PRIMITIVOS: *string*, *boolean*, *null*, *undefined*, *number*, *objeto*.

▼ OPERADORES

Todos los operadores son funciones simples. El intérprete tomará el miembro de la izquierda y el de la derecha, y realizara la operación.

El operador **+** tiene una exclusión. Para que opere como una suma, los datos de los lados deben ser ambos números. En el caso de que alguno sea otro tipo de dato (**string**, por ejemplo), el operador concatenará ambas variables.

▼ PRECEDENCIA Y ASOCIACIÓN

La ***precedencia de operadores*** es el orden en que se van a llamar las funciones de los operadores. O sea que si tenemos más de un operador, el intérprete va a llamar al operador de mayor precedencia primero y después va a seguir con los demás.

La ***asociatividad de operadores*** es el orden en el que se ejecutan los operadores cuando tienen la misma precedencia, es decir, de izquierda a derecha o de derecha a izquierda.

```
3 + 4 * 5
//23
```

Para resolver esa expresión y saber qué resultado nos va a mostrar el intérprete deberíamos conocer en qué orden ejecuta las operaciones. La multiplicación tiene mayor precedencia que la suma. Por lo tanto el intérprete primero va a ejecutar la multiplicación y luego la suma con el resultado de lo anterior

```
   let a = 1;
   let b = 2;
   let c = 3;

   a = b = c
// 3 = 3 = 3
```

Aquí la precedencia es la misma, por lo que recurrimos a la asociación. En este caso ocurre de derecha a izquierda(←). Por lo que primero **b** obtendrá el valor de **c**, y luego **a** obtendrá el valor de **b**.

Tabla de igualdades

```
6 / "3"                    //2
"2" * "3"                  //6
4 + 5 + "px"               //9px
"$" + 4 + 5                //$45
"4" - 2                    //2
"4px" - 2                  //NaN
7 / 0                      //Infinity
{}[0]                      //undefined
parseInt("09")             //9
5 && 2                     //2
2 && 5                     //5
5 || 0                     //5
0 || 5                     //5
[3]+[3]-[10]               //23
3>2>1                      //false
[] == ![]                  //true
true                       //1
false                      //0
```

▼ FUNCIONES

En JavaScript, las funciones son de tipo `first class`. Esto quiere decir que las funciones pueden ser tratadas igual que cualquier otro tipo de valor. Es

decir, que podemos pasar una función como argumento, podemos asignar una función a una variable, podemos guardarla en un arreglo, etc..

Las funciones en JavaScript son un tipo especial de objetos. Este objeto, además de poder tener cualquier propiedades adentro (como cualquier objeto) tiene dos propiedades especiales: la primera es el nombre (`name`), que contiene el nombre de la función, esta propiedad es opcional (funciones anónimas). La segunda propiedad se llama `code` (código) y en ella se guarda el código que escribiste para la función.

▼ EXPRESIÓN Y STATEMENT

Una ***Expresión*** es una unidad de código que evalúa a un valor. Por ejemplo, `a = 3`, es una expresión que devuelve el número `3`. `1 + 2` también es una expresión que retorna `3`.

Los ***Statements***, no producen un valor directamente, si no que *hacen algo*, generalmente tienen adentro expresiones. Según el statement que usemos vamos a tener un comportamiento distinto, ejemplos de statements son `if`, `while`, `for`, etc...

Respecto a las ***funciones***, también existen de dos tipos.

```
// FUNCTION STATEMENT
function saludo(){
  console.log('hola');
}

// FUNCTION EXPRESSION
var saludo = function(){
  console.log('Hola!');
}
```

▼ VALOR Y REFERENCIA

```
var a = 3;
var b = 5;
    a = b
//  5 = 5

b = 324
a = 5
```

En este caso estamos pasando variables por **valor**. Al comienzo **a** y **b** son distintas. Luego **a** se hace "***fotocopia***" de **b** y adquiere su valor. Si luego

cambiamos el valor de **b**, el valor de **a** se seguirá manteniendo igual, ya que mantiene independencia. Las variables que se pasan por valor son los *datos primitivos*.

```
var obj = {Nombre: "Alejo", Apellido: "Bengo"};
var newObj = obj;

obj.Edad = 21;
//obj = {Nombre: "Alejo", Apellido: "Bengo", Edad: 21}

<newObj>
{Nombre: "Alejo", Apellido: "Bengo", Edad: 21};
```

Cuando pasamos objetos, arreglos o funciones en variables lo haremos por **referencia**. En ese caso hay un "***reflejo***" de los cambios que hagamos en cualquiera de las variables. En el ejemplo primero creamos un objeto, y luego definimos un nuevo objeto que es igual al primero. Cuando al primero le agregamos la propiedad *Edad*, esta se agregará automáticamente a la segunda.

▼ THIS

Cuando se crea el `execution context`, el interprete reserva el espacio de memoria para las variables (hoisting), guarda la referencia al `outer enviroment` y además "setea" la variable `this`. Esta variable va a apuntar a distintos objetos dependiendo en cómo fue invocada la función.

▼ GLOBAL CONTEXT

Este es el caso cuando ejecutamos código en el contexto global (afuera de cualquier función). En este caso `this` hace referencia al objeto `global`, en el caso del browser hace referencia a `window`.

▼ FUNTION CONTEXT

Cuando estamos dentro de una función, el valor de `this` va a depender de cómo sea invocada la función.

▼ SIMPLE CALL

En este caso, el interprete le da a `this` una referencia al objeto `global`.

```
function a() {
  function b() {
    return this;
  }
  return b();
```

```
}
console.log(a());
```

▼ MÉTODO DE UN OBJETO

Si tenemos una función como propiedad de un objeto, la palabra clave **this** va a hacer referencia al objeto.

```
//--------------EJEMPLO 1---------------
var o = {  prop: 37, f: function() {return this.prop;}  };
console.log(o.f()); // logs 37
// this hace referencia a `o`

//--------------EJEMPLO 2---------------
var o = {prop: 37};
// declaramos la función
function loguea() {return this.prop;}
//agregamos la función como método del objeto `o`
o.f = loguea;
console.log(o.f()); // logs 37
// el resultado es le mismo!
```

▼ EVENT LOOP

setTimeout() es una función integrada en Javascript que nos permitirá retrasar el proceso de una función. Existe un espacio llamado "***Web Apis***" en el que es enviado el proceso de esta función. De esta forma JavaScript puede seguir procesando el código sin perder el tiempo en esa función.

Hay que destacar que, una vez enviado el **setTimeout** al **Web Apis**, JavaScript procesará el resto del código, y sólo recibirá el proceso terminado del Web Apis una vez terminada la lectura del código.

Una vez que el **setTimeout** se terminó de procesar en el Web Apis, pasará al **Callback Queue**. Aquí serán enviados todos los procesos que JavaScript deriva, y como dijimos, una vez que el intérprete termina de leer el código, recién ahí serán reintegrados al **Call Stack**.

```
function saludarMasTarde(){
var saludo = 'Hola';
setTimeout( function(){
console.log(saludo);
},3000)
};
saludarMasTarde();
```

En este **ejemplo**, el proceso es el siguiente. Primero se ejecuta la función **saludarMasTarde**, y luego, cuando se ejecuta el **setTimeout**, se lo envía al Web Apis. Cuando termina de procesarse y de transcurrirse el tiempo, se lo envía a el **Callback Queue**. Durante todo este tiempo, JavaScript sigue ejecutando el código hasta terminarlo. Una vez terminado, recibe la función del Callback Queue.