

# Introducción a JS

## Funciones

Módulo: **Desarrollo Web en entorno cliente**

CFGs Desarrollo de Aplicaciones Web

Las funciones en JS son bloques de código ejecutable, a los que podemos pasar parámetros y operar con ellos. Nos sirven para modular nuestros programas y estructurarlos en bloques que realicen una tarea concreta, y normalmente nos devuelven un valor a través del “return”.

Si declaramos una función que espera un parámetro y este no es pasado al ser llamada, el parámetro es undefined y se trata como tal. Por ejemplo:

```
function hello(name) {  
    return ("Hello " + name + "!");  
}  
hello(); // Devuelve "Hola undefined!"
```

También podemos acceder a los parámetros que se pasan por argumento a través del array **arguments** sin indicarlo en la definición de la función, aunque no es muy recomendable su uso.

```
function hello () {  
    var type = arguments[0];  
    var name = arguments[1];  
    return ( type + ", " + name + "!");  
}  
hello("Bye", "Ana"); // Devuelve "Bye, Ana!"
```

## Parámetros por defecto

Una buena práctica para evitar errores o que se tome el valor undefined sin que podamos controlarlo, es utilizar los operadores booleanos para su asignación por defecto.

```
function hello (type, name) {  
  var type = type || "Hello";  
  var name = name || "Ana";  
  return ( type + ", " + name + "!");  
}  
hello(); // "Hello, Ana!"  
hello("Bye"); // "Bye, Ana!"  
hello("See you", "Pepe"); // "See you, Pepe!"
```

## Ámbito de la función

Por defecto, cuando declaramos una variable con var la estamos declarando de forma global y es accesible desde cualquier parte de nuestra aplicación. Tenemos que tener cuidado con los nombres que elegimos ya que si declaramos a una variable con el mismo nombre en varias partes de la aplicación estaremos sobrescribiendo su valor y podemos tener errores en su funcionamiento.

Si declaramos una variable dentro de una función, esta variable tendrá un ámbito local al ámbito de esa función, es decir, solo será accesible de la función hacia adentro. Pero si la definimos fuera de una función, tendrá un ámbito global.

En **ECMAScript 6** tenemos los tipos de variable **let** y **const** en lugar de var y definen unos ámbitos específicos. const crea una constante cuyo valor no cambia durante el tiempo y let define el ámbito de la variable al ámbito donde ha sido definida.

```
var valor = "global";  
function funcionlocal () {  
    var valor = "local";  
    return valor;  
}  
console.log(valor); // "global"  
console.log(funcionLocal()); // "local"  
console.log(valor); // "global"
```

Aunque tenemos definida fuera de la función la variable `valor`, si dentro de la función la declaramos y cambiamos su valor, no afecta a la variable de fuera porque su ámbito (o **scope**) de ejecución es diferente.

## Closures o funciones cierre

Los *Closures* o funciones cierre son un patrón de diseño muy utilizado en JS y son una de las llamadas *Good parts*.

## Funciones como objetos

Las funciones en JS son objetos, ya que todo en JS es un objeto, heredan sus propiedades de la clase `Object` y podemos tratarlas como tal. Por ejemplo, podemos guardar una función en una variable y posteriormente invocarla con el operador paréntesis `()`.

```
var hi = function (name) {  
    return "Hello " + name;  
};  
hi("Ana"); // "Hola Ana"
```

Si a la variable que guarda la función no la invocamos con el operador paréntesis, el resultado que nos devolverá es el código de la función:

```
hi; // Devuelve 'function(name) { return "Hello " + name };'
```

## Funciones anidadas

Las funciones pueden tener otras funciones dentro de ellas, produciendo nuevos ámbitos para las variables definidas dentro de cada una. Y para acceder desde el exterior a las funciones internas, tenemos que invocarlas con el operador `()()`.

```
var a = "OLA";
function global () {
  var b = "K";
  function local () {
    var c = "ASE";
    return a + b + c;
  }
  return local;
}
```

```
global(); // Devuelve la función local: "function local() { var c = "ASE" ... ""
global()(); // Devuelve la ejecución de la función local: "OLAKASE"
var closure = global();
closure(); // Devuelve lo mismo que global()(): "OLAKASE"
```

## Función cierre o closure

Un *Closure* es una función que encapsula una serie de variables y definiciones locales que únicamente serán accesibles si son devueltas con el operador `return`. JS al no tener una definición de clases como tal (como por ejemplo en Java, aunque con la versión ECMAScript6 esto cambia un poco) este patrón de creación de closures, hace posible modularizar nuestro código y crear algo parecido a las clases.

En el siguiente ejemplo creamos una función que tiene un variable local que guarda el valor de un número que será incrementado o decrementado según llamemos a las funciones locales que se devuelven y acceden a esa variable. La variable local `_contador` no puede ser accesible desde fuera si no es a través de esas funciones:

```
var miContador = (function () {  
    var _contador = 0; // Por convención, a las variables "privadas" se las llama con un  
  
    function incrementar () {  
        return _contador++;  
    }  
  
    function decrementar () {  
        return _contador--;  
    }  
  
    function valor () {  
        return _contador;  
    }  
  
    return {  
        incrementar: incrementar,  
        decrementar: decrementar,  
        valor: valor  
    }  
})();  
  
miContador.valor(); // 0  
miContador.incrementar();  
miContador.incrementar();  
miContador.valor(); // 2  
miContador.decrementar();  
miContador.valor(); // 1
```

**Nota:** A las variables privadas se les suele denominar con `_nombreVariable`. Hay gente a favor y gente en contra de esta convención.

## Funciones como clases

Un closure es muy similar a una clase, la principal diferencia es que una clase tendrá un constructor que cumple el mismo cometido que el closure. Al crear un objeto a partir de una clase debemos usar el parámetro new y si es un closure, al inicializar un nuevo objeto, se le pasa lo que le devuelve la función cierre.

Veamos un ejemplo de la misma función, codificada como closure y como, y como se crearían sus objetos.

```
function inventario (nombre) {  
  var _nombre = nombre;  
  var _articulos = {};  
  
  function add (nombre, cantidad) {  
    _articulos[nombre] = cantidad;  
  }  
  
  function borrar (nombre) {  
    delete _articulos[nombre];  
  }  
  
  function cantidad (nombre) {  
    return _articulos[nombre];  
  }  
  
  function nombre () {  
    return _nombre;  
  }  
  
  return {  
    add: add,  
    borrar: borrar,  
    cantidad: cantidad,  
    nombre: nombre  
  }  
}
```

Una vez construido la closure, podemos usar sus métodos como vemos a continuación:

```
var libros = inventario("libros");  
libros.add("AngularJS", 3);  
libros.add("JavaScript", 10);  
libros.add("NodeJS", 5);  
libros.cantidad("AngularJS"); // 3  
libros.cantidad("JavaScript"); // 10  
libros.borrar("JavaScript");  
libros.cantidad("JavaScript"); // undefined
```

```
function Inventario (nombre) {  
  this.nombre = nombre;  
  this.articulos = [];  
  
  this.add = function (nombre, cantidad) {  
    this.articulos[nombre] = cantidad;  
  }  
  
  this.borrar = function (nombre) {  
    delete this.articulos[nombre];  
  }  
  
  this.cantidad = function (nombre) {  
    return this.articulos[nombre];  
  }  
  
  this.getNombre = function () {  
    return this.nombre;  
  }  
}
```

Una vez definida la clase, crear objetos a partir de ella e invocar a sus métodos sería así:

```
var libros = new Inventario("Libros");  
libros.add("AngularJS", 3);  
libros.add("JavaScript", 10);  
libros.add("NodeJS", 5);  
libros.cantidad("AngularJS"); // 3  
libros.cantidad("JavaScript"); // 10  
libros.borrar("JavaScript");  
libros.cantidad("JavaScript"); // undefined
```

Esta forma de codificar las funciones como clases se conoce como **Factory Pattern** o **Template functions**.

## Uso de Prototype

Un problema importante que tiene este tipo de estructura, es que cuando creamos un nuevo objeto a partir de esta clase, reservará espacio en memoria para toda la clase incluyendo atributos y métodos. Con un objeto solo creado no supone mucha desventaja, pero imaginemos que creamos varios objetos:

```
var libros = new Inventario("Libros");  
var discos = new Inventario("discos");  
...
```



Esto supone que las funciones de la clase, add , borrar , cantidad y getNombre están siendo replicadas en memoria, lo que hace que sea ineficiente.

Para solucionar esto podemos hacer uso del objeto Prototype que permite que objetos de la misma clase compartan métodos y no sean replicados en memoria de manera ineficiente. La forma correcta de implementar la clase Inventario sería la siguiente:

```
function Inventario (nombre) {  
  this.nombre = nombre;  
  this.articulos = [];  
};  
  
Inventario.prototype = {  
  add: function (nombre, cantidad) {  
    this.articulos[nombre] = cantidad;  
  },  
  
  borrar: function (nombre) {  
    delete this.articulos[nombre];  
  },  
  
  cantidad: function (nombre) {  
    return this.articulos[nombre];  
  },  
  
  getNombre: function () {  
    return this.nombre;  
  }  
};
```

De esta manera, si queremos crear un nuevo objeto de la clase Inventario y usar sus métodos, lo podemos hacer como veníamos haciendo hasta ahora, sólo que internamente será más eficiente el uso de la memoria por parte de JS y obtendremos una mejora en el rendimiento de nuestras aplicaciones.

Creando de nuevo los objetos libros , discos , juegos y comics , su espacio en memoria es menor (ya no tienen replicados los métodos):

```
var libros = new Inventario('libros');  
libros.getNombre();  
libros.add("AngularJS", 3);  
...  
var comics = new Inventario('comics');  
comics.add("The Walking Dead", 10);  
...
```

## Clases en ECMAScript 6

Con la llegada de la nueva versión del estándar de JavaScript (ECMAScript 6) la definición de una función como clase ha cambiado. ES6 aporta un [azúcar sintáctico](#) para declarar una clase como en la mayoría de los lenguajes de programación orientados a objetos, pero por *debajo* sigue siendo una función prototipal.

El ejemplo anterior del Inventario , transformado a ES6 sería tal que así:

```
class Inventario {
  constructor(nombre) {
    this.nombre = nombre;
    this.articulos = [];
  }

  add (nombre, cantidad) {
    this.articulos[nombre] = cantidad;
  }

  borrar (nombre) {
    delete this.articulos[nombre]
  }

  cantidad (nombre) {
    return this.articulos[nombre]
  }

  getNombre () {
    return this.nombre;
  }
}
```

Utilizando la palabra reservada `class` creamos una clase que sustituye a la función prototipal de la versión anterior.

El método especial constructor sería el que se definía en la función constructora anterior. Después los métodos `add` , `borrar` , `cantidad` y `getNombre` estarían dentro de la clase y sustituirían a las funciones prototipales de la versión ES5.

Su utilización es igual que en la versión anterior:

```
var libros = new Inventario("Libros");
libros.add("AngularJS", 3);
libros.add("JavaScript", 10);
libros.add("NodeJS", 5);
libros.cantidad("AngularJS"); // 3
libros.cantidad("JavaScript"); // 10
libros.borrar("JavaScript");
libros.cantidad("JavaScript"); // undefined
```

Con esta nueva sintáxis podemos implementar herencia de una forma muy sencilla. Imagina que tienes una clase Vehículo de la siguiente manera:

```
class Vehiculo {
  constructor (tipo, nombre, ruedas) {
    this.tipo = tipo;
    this.nombre = nombre;
    this.ruedas = ruedas
  }
  getRuedas () {
    return this.ruedas
  }
}
```

```
arrancar () {  
    console.log(`Arrancando el ${this.nombre}`)  
}  
aparcar () {  
    console.log(`Aparcando el ${this.nombre}`)  
}  
}
```

Y quieres crear ahora una clase Coche que herede de vehículo para poder utilizar los métodos que esta tiene. Esto lo podemos hacer con la clase reservada `extends` y con `super()` llamamos al constructor de la clase que hereda:

```
class Coche extends Vehiculo {  
    constructor (nombre) {  
        super('coche', nombre, 4)  
    }  
}
```

Si ahora creamos un nuevo objeto Coche podemos utilizar los métodos de la clase Vehiculo:

```
let fordFocus = new Coche('Ford Focus')  
fordFocus.getRuedas() // 4  
fordFocus.arrancar() // Arrancando el Ford Focus
```

Para saber más ....

Closures