# Funciones

Las funciones en JavaScript 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. De esta manera nuestro código es más legible y mantenible.

Las funciones normalmente, al acabar su ejecución devuelven un valor, que conseguimos con el parámetro `return`. Se declaran con la palabra reservada `function` y a continuación suelen llevar un nombre, para poder invocarlas más adelante. Si no llevan nombre se les llama funciones anónimas.

Veamos un ejemplo de función:

```
function saludar (nombre) {
  return ("Hola " + nombre + "!");
}

saludar("Carlos"); // Devuelve "Hola Carlos!"
```

La función del ejemplo se llama `saludar`, y se le pasa un único parámetro, entre paréntesis `(...)`, que es `nombre`. Ese parámetro funciona como contenedor de una variable que es utilizada dentro del bloque de código delimitado por las llaves `{...}`. El comando `return` devolverá el String que concatena texto con el valor que contiene el parámetro `nombre`.

Si no pasásemos ningún valor por parámetro, obtendríamos el valor `undefined`.

```
function saludar (nombre) {
  return ("Hola " + nombre + "!");
}

saludar(); // Devuelve "Hola undefined!"
```

También podemos acceder a los parámetros que se pasan por argumento a través del array `arguments` sin indicarlo en en la defínción de la función, aunque esta opción no es muy utilizada. Ejemplo:

```
function saludar () {
  var tipo = arguments[0];
  var nombre = arguments[1];
  return (tipo + ", " + nombre + "!");
}

saludar("Adios", "Carlos"); // Devuelve "Adios, Carlos!"
```

# Parámetros por defecto

Una buena práctica para evitar errores o que se tome el valor `undefined` sin que podamos controlarlo, es utilizar algunos de los operadores booleanos que vimos en capítulos anteriores. Si tomamos el operador OR `||` podemos asignar un valor por defecto si no está definido. Veamos un ejemplo:

```
function saludar (tipo, nombre) {
  var tipo = tipo || "Hola";
  var nombre = nombre || "Carlos";
  return (tipo + ", " + nombre + "!");
}

saludar();                      // "Hola, Carlos!"
saludar("Adios");               // "Adios, Carlos!"
saludar("Hasta luego", "Pepe"); // "Hasta luego, Pepe!"
```

# Ámbito de una 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 la versión 6 de ECMAScript 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 (por ejemplo en una función).

Con un ejemplo lo veremos más claro:

```
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. Una definición de variable local tapa a una global si tienen el mismo nombre.

# Closures

Los *Closures* o funciones cierre son un patrón de diseño muy utilizado en JavaScript y son una de las llamadas *Good parts*. Para poder comprender su funcionamiento veamos primero unos conceptos.

## Funciones como objetos

Las funciones en JavaScript son objetos, ya que todo en JavaScript es un objeto, heredan sus propiedades de la clase `Object`. Entonces pueden ser tratadas como tal. Podemos guardar una función en una variable y posteriormente invocarla con el operador paréntesis `()`. Ejemplo:

```
var saludar = function (nombre) {
  return "Hola " + nombre;
};

saludar("Carlos"); // "Hola Carlos"
```

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

```
saludar; // Devuelve 'function(nombre) { return "Hola " + nombre };'
```

## 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 doble paréntesis `()()`. Veamos un ejemplo

```
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"
```

Vistos estos conceptos ya podemos definir lo que es un `closure`.

## 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`. JavaScript 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.

Veamos un ejemplo de closure con la siguiente función. Creamos una función que tiene un variable local que guarda el valor de un numero 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 u

  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
```

# 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 clase y como closure, 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
```

Ahora veamos como sería esto mismo pero codificado como Clase:

```
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");
var juegos = new Inventario("juegos");
var comics = new Inventario("comics");
...
```

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

```
> libros
◀ ▼ Inventario {nombre: "libros", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "libros"
    ▶ __proto__: Inventario
> discos
◀ ▼ Inventario {nombre: "discos", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "discos"
    ▶ __proto__: Inventario
> comics
◀ ▼ Inventario {nombre: "comics", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "comics"
    ▶ __proto__: Inventario
> juegos
◀ ▼ Inventario {nombre: "juegos", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "juegos"
    ▶ __proto__: Inventario
>
```

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 JavaScript 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 o ECMAScript 2015) 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
```