Repaso de Funciones

Las funciones, en JavaScript, son un tipo de dato. Las podemos utilizar para agrupar una cierta funcionalidad para utilizarla las veces que necesitemos.

Para crear una función vamos a utilizar la palabra reservada function. Podemos ejecutar las funciones utilizando el nombre de la función y paréntesis

```
function saludar () {
 console.log('¡Hola mundo!');
}

saludar(); // ¡Hola mundo!
```

En este ejemplo vemos que podemos definir una función con el nombre **saludar** y luego la utilizamos llamándola con el **nombre y los paréntesis**.
Podemos ejecutar una función tantas veces como necesitemos.

Dado que las funciones en ECMASCript son un **tipo de dato** lo podemos asignar a una variable.
Al asignar una función a una variable no necesitamos darle un nombre ya que para eso podemos utilizar la variable.
Las funciones que no tienen nombre se denominan anónimas.

```
// Al asignar una función a una variable no hace falta ponerle nombre a la función
const saludar = function() {
 console.log('¡Hola mundo!');
}

saludar(); // Muestra en consola ¡Hola mundo!
saludar(); // Muestra en consola ¡Hola mundo!
```

Una función puede o no retornar un valor. Cuando una función no retorna un valor obtenemos **undefined**.
Para retornar un valor utilizamos la palabra reservada **return** dentro de la función.

```
function obtenerSaludo() {
 return 'hola';
```

```
}

// obtenerSaludo retorna un tipo de dato string y lo podemos
mostrar en console.log()
console.log( obtenerSaludo() );

// Podemos guardar el valor retornado de una función en una
variable y luego utilizarlo
const saludo = obtenerSaludo();
console.log( saludo )
```

Una función puede retornar cualquier tipo de dato.

```
function conseguirLaEdadDelUsuario() {
 return 30;
}

function usuarioLoggeado() {
 return true;
}
```

Las funciones también aceptan uno o varios valores como parámetros. De esta forma hacemos que nuestras funciones sean más flexibles y les podemos dar más funcionalidades.
Al declarar una función podemos establecer la cantidad de parámetros que acepta esa función y ponerles un nombre a cada uno. Dentro de la función podemos interactuar con los parámetros.

```
function saludar(nombrePersona) {
 console.log('¡Hola ' + nombrePersona + '!');
}

saludar('Ada'); // Muestra en consola: ¡Hola Ada!
saludar('Grace'); // Muestra en consola: ¡Hola Grace!
saludar('Hedy'); // Muestra en consola: ¡Hola Hedy!
```

Arrow Functions

Las **arrow functions** son uno de los cambios más impactantes en ES6, y se usan ampliamente en la actualidad. Se diferencian ligeramente de las funciones regulares vistas anteriormente.

Las arrow functions son una forma más concisa para escribir funciones. Utilizan el token **=>**, que parece una flecha. Las arrow functions **son anónimas**.

Las arrow functions hacen que nuestro código sea más conciso. Al utilizarlas, evitamos tener que escribir la palabra reservada **function**, la palabra reservada **return** (está implícita en las arrow functions) y las llaves.

```
// ES5
var saludar = function() {
 console.log('¡Hola mundo!');
};

// ES6
const saludar = () => {
 console.log('¡Hola mundo!');
};
```

Casos de uso

Funciones con múltiples parámetros

```
var multiplicar = function (x, y) {
 return x * y;
}// ES5



// ES6
const multiplicar = (x, y) => {
 return x * y;
}
```

En este ejemplo vemos como las arrow functions permiten llegar al mismo resultado con menos código.

Cuando la función tiene una sola expresión, podemos escribirla en una sola línea evitando las llaves y la palabra **return**

```
const multiplicar = (x, y) => x * y;
```

Funciones simples con un parámetro

Cuando una función tiene un solo parámetro, los paréntesis de los parámetros son opcionales.

```
// ES5
var saludar = function (nombre) {
 return `¡Hola, ${nombre}!`;
}

// ES6
const saludar = nombre => `¡Hola, ${nombre}!`;

console.log( saludar('Ada') ); // ¡Hola, Ada!
```

Funciones sin parámetros

Cuando una función **no** tiene parámetros, los paréntesis de los parámetros son obligatorios.

```
// ES5
var saludar = function () {
 return `¡Hola, mundo!`;
}

// ES6
const saludar = () => `¡Hola, mundo!`;

console.log( saludar() ); // ¡Hola, mundo!
```

Funciones con cuerpo

Cuando la función tiene más de una expresión, las llaves son obligatorias y es necesario el uso explícito de la palabra **return**.

```
const nombreLargo = nombre => {
 if (nombre.length > 10) {
   return true
 } else {
   return false
 }
}
```

En resumen, las arrow functions tienen 2 grandes beneficios:

1. Una sintaxis más reducida que las típicas funciones
2. No hace el binding de **this**

Parámetros por default

Los parámetros por default permiten que los parámetros de una función puedan ser definidos con un valor inicial.

Por defecto, todos los parámetros en ECMAScript son undefined. Los parámetros por default nos permiten cambiarles ese valor por defecto.

Supongamos que tenemos el siguiente ejemplo desarrollado con una versión más viaje de ECMAScript (ES5):

```
var sumar = function (a, b) {
 console.log(a + b);
}


sumar(2); // NaN
sumar(2, 5); // 7
```

Lo que está pasando es que nuestra función sumar espera que si o si le pasemos dos parámetros. En el ejemplo, como solo la ejecutamos con un parámetro,no puede realizar la suma y nos termina mostrando por consola el valor **NaN** (not a number).

Si utilizaramos default parameters, esto podríamos arreglarlo de la siguiente forma:

```
const sumar = function (a, b = 0) {
 console.log(a + b);
}

sumar(2); // 2
sumar(2, 5); // 7
```

En este último ejemplo, cuando escribimos `function (a, b = 0)` le estamos diciendo que la variable **b** tenga un valor inicial de 0. Si ejecutamos la función con un solo parámetro, entonces **b** va a tener el valor 0. Si ejecutamos la función con los dos parámetros, entonces **b** va a tener el valor que hayamos puesto al ejecutar la función.

Los parámetros por defecto pueden ser realmente útiles para asegurarnos que los datos que nos llegan en una función estén disponibles para realizar una acción.

```
const incrementar = (numero, cantidad = 1) => numero + cantidad;

console.log( incrementar(4) ) // 5
console.log( incrementar(5, 8) ) // 13
console.log( incrementar(7) ) // 8
```