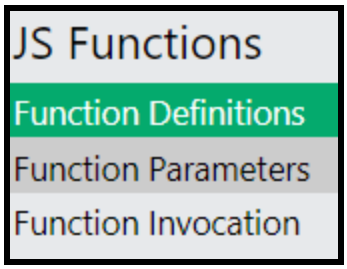
JS-Tarea9-Funciones(II)-Definiciones de funciones de JavaScript

DAM2 - DWEB

Objetivo:

- ➢ diferenciar y desarrollar en JavaScript:
  - **funciones declaradas,**
  - **funciones anónimas** *(funciones definidas como expresión),*
  - **funciones autoinvocadas** *(IIFE Immediately-invoked function expressions)*
  - **funciones flecha** ES6 *(funciones definidas como expresión de ES5)*
- ➢ toma de contacto con el uso de Templates Literals

Funciones en JavaScript *(teoría en: w3schools)*



1. **Ejemplo de función declarada:**

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Functions</h2>

<p>Este ejemplo llama a una función que realiza un cálculo y devuelve el
resultado:</p>

<p id="demo"></p>

<script>
var x = myFunction(4, 3);
document.getElementById("demo").innerHTML = x;

function myFunction(a, b) {
  return a * b;
}
</script>

</body>
</html>
```

2. **Ejemplo de función anónima** *(función definida como una expresión)*
   Se declaran sin nombre de función y se alojan en el interior de una variable. Haremos referencia a dicha variable cada vez que queramos invocar la función.

```
<!DOCTYPE html>
<html>
<body>

<p>Una función se puede almacenar en una variable:</p>
<p id="demo"></p>

<script>
var x = function (a, b) {return a * b};
document.getElementById("demo").innerHTML = x(4,3);
</script>

</body>
</html>
```

   La diferencia fundamental entre las funciones por declaración y las funciones por expresión es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si *«ejecutamos la variable»* antes de declararla nos dará un error.

3. **Ejemplo de función flecha (ES6):**
   Es la forma abreviada de escribir funciones que aparece en Javascript a partir de ECMAScript 6. Básicamente, se trata de reemplazar/eliminar la palabra ***function*** y añadir los caracteres **“=>”** antes de las llaves que contendrán el cuerpo de la función:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Arrow Functions</h2>

<p>Las funciones de flecha no son compatibles con IE11 o versiones
anteriores</p>

<p id="demo"></p>

<script>
let x = (x, y) => { return x * y };
document.getElementById("demo").innerHTML = x(5, 5);
</script>

</body>
</html>
```

***ojo!!**:*

- ➔ *Las funciones flecha **no** se "izan": sino que se deben definir antes de su uso*
- ➔ *Las funciones flecha siempre son anónimas.*

Las funciones flechas tienen algunas ventajas interesantes a la hora de simplificar código:

- Si el cuerpo de la función sólo tiene una línea podemos omitir las llaves ***{}***. Además en ese caso, automáticamente se hace un return de esa única línea, por lo que podemos omitir también la palabra reservada *return*. La definición del ejemplo anterior se vería así:

  ```
  let x = (x, y) => x * y;
  ```

- En el caso de que la función no tenga parámetros, se indica solamente con ***() =>***. Ejemplo:

  ```
  let x=()=>(console.log("hola caracola"));
  console.log(x());
  ```

  provoca en consola la salida: *hola caracola*

- En el caso de que la función tenga un solo parámetro, se puede indicar simplemente el nombre del mismo sin los paréntesis ***nombreParametro =>***. Ejemplo: los siguientes dos códigos son equivalentes:

  ```
  let x=(y)=>(console.log("hola " + y));
  console.log(x("Pepito"));
  ```

  lo escribiremos de forma abreviada así:

  ```
  let p=y=>(console.log("hola " + y));
  console.log(p("Pepito"));
  ```

  ambos códigos provocan en la consola la misma salida: *hola Pepito*

- En el caso de que la función tenga 2 ó más parámetros, se indican como en el primero de los ejemplos, entre paréntesis: *(a, b) =>*.

- Si queremos retornar un objeto, puesto que coincide la sintaxis de llaves se han de incluir paréntesis. Ejemplo:

  ```
  let x = id => ({id: id, name: "Temp"});
  console.log(x(8));
  ```

  provoca la salida: *{id: 8, name: 'Temp'}*

4. **Ejemplo de función autoinvocada** *(IIFE Immediately-invoked function expressions):*
Son un tipo de funciones que se caracterizan por ser invocadas automáticamente cuando las creamos, es decir que se ejecutan tan pronto como se definen. Es un patrón de diseño también conocido cómo **función autoejecutable.**

```
<!DOCTYPE html>
<html>
<body>
	<p>Las funciones se pueden invocar automáticamente sin ser llamadas:</p>
	<p id="demo"></p>

	<script>
	(function () {
	  document.getElementById("demo").innerHTML = "Hello! I called myself";
	})();
	</script>

</body>
</html>
```

Utilidad de las funciones autoinvocadas
Este tipo de funciones las encontramos a menudo en patrones y bibliotecas de Javascript con la intención de encapsular un bloque de código dentro de un ámbito local en busca de **privacidad de los datos**.

Por un lado, las variables que definamos en su interior no van a poder ser accedidas desde fuera de la función IIFE. Por otro lado, los paréntesis que la encierran convierten a la función en una *expresión de función* que se ejecuta inmediatamente.

```
var saludo = "hola";

(function(nombre){
  var saludo = "buenos días";
  console.log(saludo + " " + nombre);

}("Pepe"));  //>> buenos días Pepe
```

En el ejemplo, la IIFE se ejecuta automáticamente después de ser definida y muestra por la consola un saludo. En este caso, dentro de la IIFE la variable saludo contiene el texto "buenos días" , sin afectar a la variable saludo global.

En muchos programas y bibliotecas, verás que se encierra el código JavaScript en una IIFE para que sus variables y funciones no colisionen con otras definidas en otros archivos. Esto es así, porque aunque en ES6 es posible declarar ámbitos a nivel de bloque, en ES5 no lo era, y es por ello que se emulaba este comportamiento con el uso de IIFEs.

En pocas palabras, para conseguir un encapsulamiento *(técnica de ocultación para aislar el objeto o en este caso la variable del exterior)* antes de ES6 se hacía de esta manera:

```
(function () {
   var x = 10;
})();
console.log(x);
```

provoca la salida: *Reference Error: x is not defined*

Pero en ES6 es posible usar lo siguiente:

```
{
   let q = 10;
};
console.log(q);
```

provoca la salida: *Reference Error: q is not defined*

Esta nueva característica *block scoping* es posible usando las palabras reservadas *let* y *const* que se introdujeron en el nuevo estándar de Javascript. Sin embargo, *var* sigue existiendo por cuestiones de compatibilidad.

Por lo tanto, dado que *var* tiene alcance a nivel de función y no a nivel de bloque, si queremos encapsular la variable x del ejemplo declarada con *var,* hace falta usar IIFEs como la del primer ejemplo ya que unas llaves no serán suficiente:

```
{
   var x = 10;
};
console.log(x);
```

provoca la salida: 10

Tarea: Realiza las siguientes ***prácticas guiadas*** prestando especial atención a cómo defines en cada caso la función, así como a los elementos ES6 incorporados (***let y const***). En las pruebas propuestas se introducen también los Templates Literals *(Plantillas de cadena de texto)* que nos permiten insertar variables y expresiones dentro de una cadena de texto (nótese el uso de **backtips** ` `, en lugar de **comillas simples ‘ ’** y para insertar mi variable emplea la notación **${miVariable}**)

Prueba1.html

```
<!DOCTYPE html>
<html><head>
<meta content="text/html; charset=utf-8">

<script>

    function saludo(nombre)
    {
        document.write(`Hola ${nombre} `);
    }

</script>

</head>
<body>

<input type="button" onclick="saludo('Ana')" value="Pulsa aquí">

</body>
</html>
```

1. Ejecuta, revisa y estudia el código propuesto.
2. ¿En qué momento se "dispara" la función?
3. Coloca un punto de interrupción en la línea de código ***document.write (`Hola ${nombre} `)*** y otro punto de interrupción en la línea ***<input type>***. Empezar la depuración e ir ejecutando "paso a paso" (F10) Ver el orden en que se ejecutan las instrucciones. Ayuda-Depuración de errores con puntos de interrupción (Chrome)

Prueba2.html Ejecutar el siguiente código y comprueba el efecto de la falta de parámetro...

```
<!DOCTYPE html>
<html><head>
<meta content="text/html; charset=utf-8">

<script>

function Calcula(numero=1)
{
  let calculo = numero * numero;
  document.write(`<p/> Resultado = ${calculo} </p>`);
}

</script>

</head>
<body>

<input type="button" onclick="Calcula()" value="Calculo">
<input type="button" onclick="Calcula(5)" value="Otro Calculo ">

</body>
</html>
```

Parámetro (numero) , si no recibe recibe valor, vale 1 .
Se utiliza como variable local en la función.

Prueba3.html

En base al siguiente código añade función para restar dos números

```
<!DOCTYPE html>
<html><body>
<script>
    let multipli = function(x,y){
        return x*y
    }

    let expon = function (x,y){
        return Math.pow(x, y);
    }

    let suma = (x,y) => {return x+y};


var result = multipli(5,2);
alert (result);

result = suma (3,4);
alert (result);

</script>
</body>
</html>
```

Prueba4.html

En base al siguiente código añade función para pasar de Fahrenheit a Celsius

```
<!DOCTYPE html>
<html><body>
<script>
let pasaraFarenheit = (grado) => {return Fahrenheit = grado * 1.8 + 32;};

var result = pasaraFarenheit(22);
alert (result);

</script>
</body>
</html>
```

Prueba5.html

En base al siguiente código, realizar programa que muestra en pantalla un saludo al nombre de dos personas introducido por teclado.

```
<!DOCTYPE html>
<html><head>
<meta content="text/html; charset=utf-8">
<script>

function Saludo(persona1,persona2) {
document.getElementById("salida").innerHTML = `Hola ${persona1} y ${persona2}`;
}
</script>
</head>

<body>
<h1>Función con dos parámetros</h1>
<p>Saluda a dos personas.</p></br>
<p id="salida">Aquí el resultado</p>

<script>
    Saludo("Ana","Pedro");
</script>

</body>
</html>
```