Desarrollo web en entorno cliente

Tema 1: Introducción al desarrollo web en el lado del cliente

1. Introducción

Las normas básicas de la sintaxis de JavaScript son las siguientes:

- No tiene en cuenta espacios en blanco ni las nuevas líneas.
- Distingue entre mayúsculas y minúsculas.
- No define el tipo de las variables.
- No es necesario terminar cada sentencia con ";" pero es conveniente.

Inclusión de JavaScript en una página web

Las tres opciones para incluir JavaScript en un documento HMTL son:

- En el cuerpo del documento:

```
<span onclick="alert('Hola');">Clic aqui</span>
```

- Entre las etiquetas <script> </script>, incluido en el head o en el body:

```
<script>
    function alerta() { alert('Hola');}
</script>
```

- En un documento aparte que se referenciará desde el documento HTML. Esta referencia se incluirá en el head o en el body del documento:

```
<script src="../scripts/javascript.js"></script>
```

En caso de que se incluya el código JavaScript en un fichero diferente (que es lo recomendable), si se pone la referencia al final del body, no se detendrá el renderizado de la página mientras se descarga y ejecuta el código. En caso de que se ponga en el head, es recomendable hacer uso de los atributos async y/o defer para evitar problemas:

- Los scripts async descargarán el script sin bloquear la renderización de la página y lo ejecutarán tan pronto como el script termine de descargarse. Se recomienda usar async cuando los scripts de la página se ejecuten independientemente unos de otros.

```
<script async src="../scripts/scripts1.js"></script>
```

RENDERIZAR: proceso de convertir el código fuente de una página web en la representación visual que vemos en el navegador

- defer cargará los scripts en el orden en que aparecen en la página y los ejecutará tan pronto como el script y el contenido sean descargados:

```
<script defer src="../scripts/scripts1.js"></script>
```

Comentarios

Los comentarios introducidos en un documento JavaScript no se visualizan por pantalla, pero sí que se envían al navegador por lo que el usuario podría acceder a los mismos. Se definen como sigue:

- /Comentarios de línea:

```
//Esto es un comentario de una sola línea. No hay que cerrarlo
```

- Comentarios de bloque:

```
/* Este comentario puede ocupar más de una línea.
   --------------------
Se deberá cerrar al terminar */
```

Mensajes de alerta

Mediante las siguientes funciones JavaScript permite pedir o mostrar información al usuario a través de ventanas modales:

- **alert**(“Mensaje”): Mediante este método se mostrará un mensaje por pantalla al usuario. Este método no devuelve ningún valor.
- **prompt**(“Mensaje”[, valorPredeterminado]): Utilizando este método se mostrará al usuario un mensaje solicitando información. Devuelve el valor introducido si el usuario pulsa Aceptar, o null en caso contrario.
- **confirm**(“Mensaje”): Al utilizar este método se mostrará por pantalla un mensaje de alerta para su confirmación o cancelación. Este método devuelve true si el usuario acepta el mensaje y false en caso contrario.

Si lo que se quiere es sacar mensajes de log que se mostrarán en la consola del navegador:

- **console.log**(“Mensaje”): El mensaje incluido aparecerá en la consola del navegador.
- **console.error**(“Mensaje”): El mensaje se mostrará en la consola de la misma forma que con la opción console.log(), pero se mostrará como si se tratara de un error JavaScript.

EJERCICIO 1

- Visualiza el siguiente video para entender la diferencia entre los parámetros async y defer: https://somostechies.com/async-vs-defer/
- Busca que más opciones de mensajes por consola ofrece console.
- Lanza diferentes mensajes de alerta desde la consola del navegador para probar su uso.

2. Variables y tipos de datos

2.1. Declaración de variables y alcance

Las variables son contenedores para almacenar valores que se declaran con una de las siguientes palabras clave: **var**, **let** o **const**(aunque no es obligatorio si es recomendable). El valor se asignará en la misma creación o más tarde:

```
var variable1 = "valor";
let variable2;
variable2 = "valor";
```

Hay que saber que un bloque es una estructura que agrupa sentencias, delimitada por un par de llaves. Puede ser una función, una sentencia condicional, un bucle, etc.

La declaración con **var** define una variable en el ámbito local actual (función) y se hereda a elementos descendientes por referencia (bloques, por ejemplo). Si la variable es declarada fuera de una función, o dentro de ella, pero sin la palabra **var**, la variable será global.

Si se declara una variable con **let** o **const** tendrá alcance de bloque y no será visible para la función que lo contenga. La variable declarada con **const** no podrá ser reasignada, pero si su valor es mutable se pueden cambiar los valores de sus elementos. Por ejemplo:

```
const user = { name: 'Juan' };
user.name = 'Manolo';
```

En general, **let** se usaría dentro de un bloque o función, **const** para variables que no van sufrir reasignación y **var** para el resto.

El nombre de una variable podrá contener letras, números, guiones bajos o símbolos dólar ($) y no podrá comenzar por número. Es recomendable que se siga la sintaxis lowerCamelCase(esteEsUnEjemploDeNombre). Su contenido podrá ser:

- Numérico: cualquier tipo de número real o entero.
- Lógico o booleano: true, false, 1 o 0.
- Cadena de caracteres: cualquier combinación de caracteres (letras, números, signos especiales y espacios). Se delimitan mediante comillas dobles o simples.

2.2. Tipos de datos

Los tipos de datos que define el estándar son:

- Tipos primitivos:
  - **Boolean**: tomará los valores *true* o *false*.
  - **Null**: tendrá el valor *null*.
  - **Undefined**: variable sin valor asignado. Tendrá el valor *undefined*.
  - **Number**: cualquier tipo de número entero o decimal. El carácter para la coma decimal es el ".". Este tipo tiene tres valores simbólicos: *+Infinity*, *-Infinity* y *NaN*(Not A Number).
  - **String**: Cadena de texto. Irá entre comillas.

- **Symbol**: Tipo nuevo de la última versión. Se trata de un valor primitivo único e inmutable y puede ser usado como la clave de una propiedad de un Object.
- Tipos **Object**:
  - lista de pares clave-valor donde la clave será un String o un Symbol.
  - Existe un conjunto de objetos especiales como el "objeto global", el "objeto prototipo", los arrays, las funciones, las clases definidas por el programador, las clases predefinidas JavaScript, etc.

En javascript no es necesario declarar el tipo de variable antes de usarla, ya que éste será determinado automáticamente cuando el programa comience a ser procesado y podrá cambiar a lo largo de la ejecución del mismo.

2.3. Conversión entre tipos

Para las operaciones, JavaScript realiza de manera automática las conversiones necesarias:

- 8 * true = 8, ya que la operación que se realiza es 8*1.
- 15 + '5' = 155, ya que convierte 15 a cadena y concatena.

Luego hay elementos, como la función prompt(), que devuelven siempre cadenas, y para casos como este existen funciones de conversión de datos. Algunas de las más usadas son:

- **Number**("cadena"): Convierte una cadena a valor numérico.
- **parseInt**(valor): convierte el valor pasado como parámetro a un número entero. Siempre que el valor empiece por un número lo convertirá:

```
parseInt(5.7) // Devuelve 5
parseInt('5.7') // Devuelve 5
parseInt('5 coches') // Devuelve 5, Number devolvería NaN
```

- **parseFloat**(valor): Igual que el anterior, pero a número decimal.
- **Boolean**(valor): Convierte cualquier valor a booleano. Cualquier valor se evaluará a *true* excepto *0*, *NaN*, *null*, *undefined* o cadena vacía ('') que se evaluarán a *false*.

EJERCICIO 2

Realiza los siguientes ejercicios en la consola del navegador:

- Comprueba, utilizando el operador typeof(), el tipo de dato de 3, "3", tres y 3.6 .
- Comprueba el resultado de las siguientes conversiones: parseInt("345.87"), parseFloat("345.87"), parseInt('8 manzanas') y Number('8 manzanas').
- Comprueba el tipo de dato del resultado de las conversiones del punto anterior.
- Comprueba el resultado de convertir a booleano una cadena de caracteres y un valor numérico.

3. Operadores

Los tipos de operadores más comunes en JavaScript son:

- Operadores aritméticos: Para operaciones aritméticas con valores y/o variables.
- Operadores comparativos: Para hacer comparaciones entre valores y/o variables.
- Operadores lógicos: Para establecer condiciones.

3.1. Operadores aritméticos

+	Realiza la suma de dos numerales o concatena cadenas. Si se intenta sumar un número y una cadena, JavaScript los concatena como dos cadenas
-	Resta
*	Producto
/	Asigna al resultado el cociente de dos numerales
%	Asigna al resultado el resto de la división de dos numerales
++	Incrementa un valor en una unidad
--	Decremento de un valor en una unidad

Solo el operador Adición tiene una función distinta de las aritméticas, para el resto, si se emplean tipos de variable que no sean números, el resultado es *NaN*.

3.2. Operadores lógicos

>	var1 > var2 →true si var1 mayor que var2
<	var1 < var2 →true si var1 menor que var2
==	var1 == var2 →true si var1 igual que var2
!=	var1 != var2 →true si var1 distinta que var2
>=	var1 >= var2 →true si var1 mayor o igual que var2
<=	var1 <= var2 →true si var1 menor o igual que var2
===	var1 === var2 →true si ambas son idénticas, es decir, si sus valores coinciden y son del mismo tipo
!==	var1 !== var2 →true si no son idénticas, es decir, si sus valores no coinciden o no son del mismo tipo

Hay que tener en cuenta que:

- == y !=: Comparan las series de caracteres variables, y su orden.
- >, <, >=, <=: Comparan el orden alfabético de los caracteres, para ver cuál sería anterior, característica que se iguala a mayor.

3.3. Operadores condicionales

!	NOT	true si no se cumple la condición
&&	AND	true si se cumplen ambas condiciones
||	OR	true si se cumple una de las dos condiciones

En los casos de AND y OR, cada condición que se establezca deberá ir entre paréntesis:

```
if((var1>0)&&(var2<0)) { … }
```

EJERCICIO 3:

Piensa que resultado darían las siguientes expresiones y comprueba si estás en lo cierto ejecutándolas desde la consola del navegador:

- let num1=5, num2=8, resultado1, resultado2;
  resultado1=((num1+num2)*200)/100;
  resultado2=resultado1%3;
  resultado1=++num1;
  resultado2=num2++;
  resultado1=-resultado2;

- let var1=4, var2=5;
  alert("var1==var2 es " + (var1==var2));
  alert("var1!=var2 es " + (var1!=var2));
  alert("var1>var2 es " + (var1>var2));
  alert("var1<=var2 es " + (var1<=var2));

- alert("false&&false es " + (false&&false));
  alert("true&&false es " + (true&&false));
  alert("false||true es " + (false||true));
  alert("true||true es " + (true||true));
  alert("!false es " + (!false));

4. Arrays

Un array se utiliza para almacenar más de un valor en una sola variable. En un array se usan índices numéricos. Las dos formas siguientes de declaración de arrays hacen exactamente lo mismo, aunque es más simple y rápida en ejecución la primera de ellas.

```
let variables1 = ["var1","var2","var3","var4"];
let variables2 = new Array("var1","var2","var3","var4");
```

Se accede a un elemento del array indicando su posición dentro del mismo, teniendo en cuenta que las posiciones empiezan en 0. Al array completo se puede acceder por su nombre:

```
let num3 = variables[2];
alert(num3);
```

Un array puede almacenar valores de distintos tipos en cada elemento:

```
let array1 = [1,2,3,4,5];
array1[1] = ["cadena1","cadena2", "cadena3"];
array1[2] = "pepito";
```

EJERCICIO 4:

Prueba las siguientes instrucciones en la consola del navegador:

- let array1=new Array();
- array1[0]="Rocío"; array1[1]=15; array1[2]="13";
- array1;
- let array2=["9.78",7,"Zaragoza"];
- array2;
- alert(array1[1]);
- alert(array2[0]);
- alert(array1 + "___" + array2);

5. Estructuras de control y bucles

5.1. if / if..else

Si se cumple la condición especificada, se ejecutará el bloque entre llaves, en caso contrario se ejecutará el bloque indicado por el else (en caso de que se haya incluido):

```
if (condición)
        sentencia1;
}else {
        sentencia2;
        sentencia3;
}
```

5.2. switch

Permite listar una serie de bloques de enunciados que se ejecuten dependiendo del valor de una variable. El valor se comprobará usando comparación estricta(===).

```
switch (variable){
        case 1:
                sentencias;
                break;
        …
        case n:
                sentencias;
                break;
        default:
                sentencias;
                break;
}
```

Si no se incluye algún break, el programa continuará ejecutando el siguiente case. Si hay más de un case seguidos (sin sentencias ni break entre ellos), para todos se ejecutará la sentencia detallada para el último de ellos:

```
switch (var1) {
        case ‘uno’:
        case ‘dos’:
        case tres':
                console.log(menor que 5'); break;
        case ‘cinco’:
        case ‘seis’:
                console.log(mayor o igual que 5'); break;
}
```

5.3. for / for…in

La sintaxis de un blucle **for** es la siguiente:

```
for(expresionInicial ; condición ; variaciónContador){
        sentencias;
}
```

```
for (let i = 0; i < 9; i++) { n += i; }
```

Al recorrer los datos de un objeto con **for..in** se mostrará el nombre de la propiedad, para mostrar su valor habría que acceder a través de ese nombre utilizándolo como índice.

```
let obj1 = { var1 : 1, var2 : 2 };
for( var property in obj1 ){
    console.log( property );
}
```

5.4. do…while

El bucle do…while tiene el mismo comportamiento que un bucle for solo que primero se ejecutan las sentencias y después se evalúa la condición:

```
do{
        sentencias
} while (condición);
```

En este caso, habría que declarar el contador fuera del bloque para poder utilizarlo.

5.5. while

Mismo comportamiento que en el caso do…while, salvo que en este caso la sentencia se evalúa antes de la ejecución de cualquier sentencia.

```
while (condicion) {
        sentencias
}
```

5.6. Instrucciones break y continue

En la ejecución de un bucle for, while o do-while, se puede modificar la secuencia lógica de ejecución mediante el uso de las instrucciones break y continue.

La instrucción **break** dentro de un bucle hace que la ejecución del mismo se interrumpa inmediatamente y continúe el programa inmediatamente a continuación del final del bucle.

Por su parte, el uso de la instrucción **continue** en un bucle hace que se retorne a la cabecera del bucle, volviendo a ejecutar la condición o a incrementar los índices cuando sea un bucle for, permitiendo saltarse recorridos del bucle.

En este primer ejemplo se pide al usuario la introducción de la palabra “Coche”, permitiéndole 5 intentos. En caso de introducir la palabra correcta no se seguirá ejecutando:

```
let numVeces = 1;
let palabra;
while (numVeces <= 5) {
    palabra = prompt("Escribe la palabra Coche ");
    console.log("Intento número " + numVeces);
    numVeces++;
    if (palabra == "Coche") {
        console.log(palabra);
        break;
    }
}
```

En el siguiente ejemplo, dependiendo de la elección del usuario se ejecutarán las sentencias siguientes o se volverá a la cabecera del bucle:

```
let eleccion;
for (let i = 0; i < 10; i++) {
    console.log(" Valor del contador: " + i);
    eleccion = confirm("¿Terminar la ejecución de la iteración?");
    if (eleccion)
        continue;
    console.log("Se realiza un incremento extra del contador");
    i++;
}
```

EJERCICIO 5:

- Prueba, en la consola del navegador, los dos ejemplos de las sentencias break y continue.
- Define una función donde se pida, repetidamente, al usuario que introduzca un color. Utiliza las sentencias break y continue donde corresponda:
  - Si el color introducido es el rojo, la ejecución saldrá del bucle.
  - Si el color indicado es el verde, se volverá al comienzo del bloque sin ejecutar las sentencias siguientes.
  - Si se trata de cualquier otro color, se incluirá éste en un array. Si el color ya estuviera se repite.
  - Tras terminar la ejecución del bucle se mostrará por pantalla el número de veces que el usuario ha introducido un color y todos los valores almacenados en el array.

```
function pedirUnColor(){

  let colores = [];
  let contador = 0;
  let color = "";

 while (true) {
     color = prompt("Introduce un color: ")
     contador ++;

     if (color == "rojo"){
       break;
     } if (color == "verde"){
       continue;
     }
  else {

     colores.push(color);
 }

  alert(contador + "-- " + colores);

 }

pedirUnColor();
```

6. Funciones

Una función es un fragmento de código que puede ser invocado. El uso de funciones ayuda en la resolución de problemas complejos, evita la repetición de código y facilita el mantenimiento y correcciones futuras.

En JavaScript, las funciones son objetos, es decir, se pueden manipular y transmitir al igual que cualquier otro objeto. Concretamente son objetos de tipo **Function**.

6.1. Definición de una función

Una función se declara con la palabra function y, declarada de esta manera, se cargará antes de la ejecución del código:

```
function función (par1, par2, …, parN) {
        sentencias…
}
```

Una expresión de función es una función anónima, que se asigna a una variable y que no estará disponible hasta que no se ejecute esa línea:

```
var variable = function(){
        sentencias…
}
```

6.2. Parámetros de entrada y salida

Los parámetros de entrada se incluyen en la definición de la función solo con el nombre, sin indicar tipo, entre paréntesis y separados por comas. En Javascript los parámetros siempre se pasan por valor, por lo que si dentro de la función se modifica su valor, será un cambio local.

Si la función va a devolver algún valor se hará uso de la sentencia **return**. No hay que declarar en la cabecera de la función el tipo de dato del retorno:

```
function suma(a,b){
        return a+b;
}
```

Si a una función se le pasan más parámetros de los que ha definido, se ignorarán los sobrantes. Por el contrario, si se pasan menos de los necesarios, los que faltan tendrán valor undefined.

Se puede dar un valor por defecto a los parámetros, por si no son pasados en una llamada al a función:

```
function suma(a=3,b=5){
    return a+b;

}
```

En caso de que no se vaya a saber cuántos parámetros van a ser pasados, se puede acceder a los mismos desde el array **arguments[]**:

```
function mostrarCiudades() {
    for (const i in arguments) {
        alert(arguments[i]);
    }
}
mostrarCiudades("Madrid", "Bilbao", "Salamanca");
```

Una función, al ser un objeto, se pasar como parámetro a otra o asignarla a una variable:

```
function funcionParam(dato, funcion) {
    return funcion(dato);
}
```

6.3. Recursividad

Una función recursiva es una función que se llama a sí misma, evitando el uso de bucles, hasta que se cumpla una condición establecida.

Esta forma de programación suele simplificar el código y hacerlo más intuitivo, aunque también baja el rendimiento, ya que por cada llamada se crea una nueva entrada en la pila de llamadas y se puede llegar a producir un desbordamiento de pila. Para evitar esto, hay que tener bien definida la condición de parada.

```
function miFuncion(n) {
        ...
        miFuncion (n-1);
        …
}
```

EJERCICIO 6:

- Prueba, en la consola del navegador:
  - Un ejemplo de establecimiento de valor por defecto a algún parámetro de entrada.
  - el ejemplo de la función mostrarCiudades().
- Implementa una solución que calcule la secuencia de Fibonacci, tomando como límite un número que el usuario introduzca por pantalla. En ambos casos se irá mostrando cada número de la secuencia por consola:
  - Con recursividad.
  - Sin recursividad.

7. Manejo de errores

En JavaScript, para llevar un control de los errores que se puedan producir en la ejecución, se utilizan sentencias **try catch**:

```
try {
        ...
}
catch(error) {
        ...
}
```

El funcionamiento es similar al de sentencias homónimas de otros lenguajes, donde un error producido dentro del bloque try será tratado dentro del bloque catch. Puede incluirse un tercer bloque, identificado con la palabra **finally**, que se ejecutará en cualquier caso.

Para lanzar un error desde nuestro código lo haremos con la sentencia **throw**:

```
if (causa)
        throw 'Error: causa';
```

También podemos lanzar un objeto de tipo **Error**:

```
if (causa)
        throw new Error('Error: causa');
```

Y por último, existe la posibilidad de crear tipos de errores propios, derivados de Error:

```
    function ExceptionMes(mensaje) {
        this.message = "Error: " + mensaje;
    }
    ExceptionMes.prototype = Object.create(Error.prototype);
    ExceptionMes.prototype.name = " ExceptionMes ";

//función donde se va a usar el tipo de error creado
    function getNombreMes(numMes) {
        numMes--;
        let meses = new Array("Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul",
 "Ago", "Sep", "Oct", "Nov", "Dic");
        if (meses[numMes] != null) {
            return meses[numMes];
        } else {
            throw new ExceptionMes("Numero de Mes No Valido");
        }
    }
```

8. Buenas prácticas

Al programar con Javascript hay que tener cuidado, ya que no es tan estricto como pueda serlo, por ejemplo, Java, y se pueden cometer errores no controlados de los que no va a avisar.

'use strict'

Esta sentencia, incluida al comienzo del documento JavaScript, permite que el intérprete avise, por ejemplo, de variables sin declarar. Con ello, el navegador no va a permitir:

- Usar una variable sin declarar.
- Definir más de 1 vez una propiedad de un objeto.
- Duplicar un parámetro en una función.
- Usar números en octal.
- Modificar una propiedad de sólo lectura.

Variables

En cuanto a las variables, es conveniente seguir las siguientes recomendaciones:

- El nombre debe ser claro, evitando abreviaturas o nombre sin significado.
- Usar para nombrarlas la notación lowerCamelCase.
- Usar let o const cuando corresponda, evitando abusar del uso de var.
- Declarar todas las variables al principio de cada bloque o función.
- Evitar, siempre que sea posible, variables globales.
- Inicializar las variables al declararlas.
- Evitar conversiones de tipo automáticas.
- Por motivos de eficiencia, evitar el uso de objetos Number, String o Boolean, sustituyéndolos por los tipos primitivos, y lo mismo al crear arrays, objetos o expresiones regulares (usar let array = [] en lugar de let array = new Array()).

Errores

Cuando, durante la ejecución de una función se produce un error, no es óptimo comprobar si ésta devuelve un valor correcto o no, sino que debería lanzarse un error en el momento de producirse. La llamada a dicha función se incluirá en un try...catch para capturar el error.

Otras

De forma general, se consideran buenas prácticas de programación las siguientes:

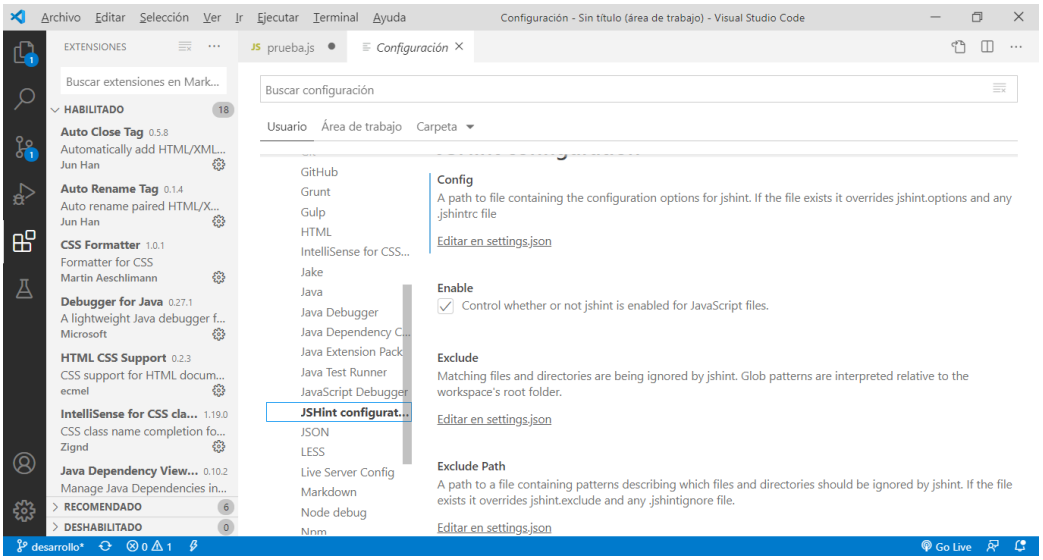
- Programar con coherencia manteniendo un mismo estilo a lo largo de todo le desarrollo. Por ejemplo, si se ponen espacios antes y después del = en una asignación hacerlo siempre igual.
- Separar líneas de más de 80 caracteres para una mejor legibilidad del código.
- Usar === en las comparaciones.

- Asignar valores por defecto a los parámetros de una función en caso de que sea posible que falten en la llamada.
- Programar de forma clara para que se entienda de un vistazo lo que se está haciendo.
- Comentar el código para complementar al punto anterior, en caso de que sea necesario.

Un validador de buenas prácticas de JavaScript es JSHint. Se trata de una herramienta que verifica que el código cumpla con las reglas de codificación: https://jshint.com/

También se puede instalar en VSC una extensión basada en jshint. Para ello:

- Descargar NodeJS e instalarlo en el equipo.
- Desde el símbolo del sistema o poweshell, comprobar si se ha instalado bien con npm –version.
- En VSC descargar extensión JSHint.
- Instalar JSHint con npm install -g jshint.
- Para habilitar ES6, en la configuración de jshint de la "configuración del usuario", editando el fichero de settings, hay que incluir la siguiente instrucción: "jshint.options": { "esversion": 6 }




prueba.js
Configuración
settings.json
C: > Users > Portatil > AppData > Roaming > Code > User > settings.json > ...
{
"window.zoomLevel": 1,
"workbench.colorTheme": "Visual Studio Light",
"liveServer.settings.donotShowInfoMsg": true,
"liveServer.settings.CustomBrowser": "chrome",
"liveServer.settings.donotVerifyTags": true,
"editor.suggestSelection": "first",
"vsintellicode.modify.editor.suggestSelection": "automaticallyOverrodeDef
"[xml]": {
"editor.defaultFormatter": "DotJoshJohnson.xml"
},
"git.autofetch": true,
"jshint.config": "",
"jshint.options": {
"esversion": 6
},
"jshint.lintHTML": true,
"jshint.trace.server": "messages"
}