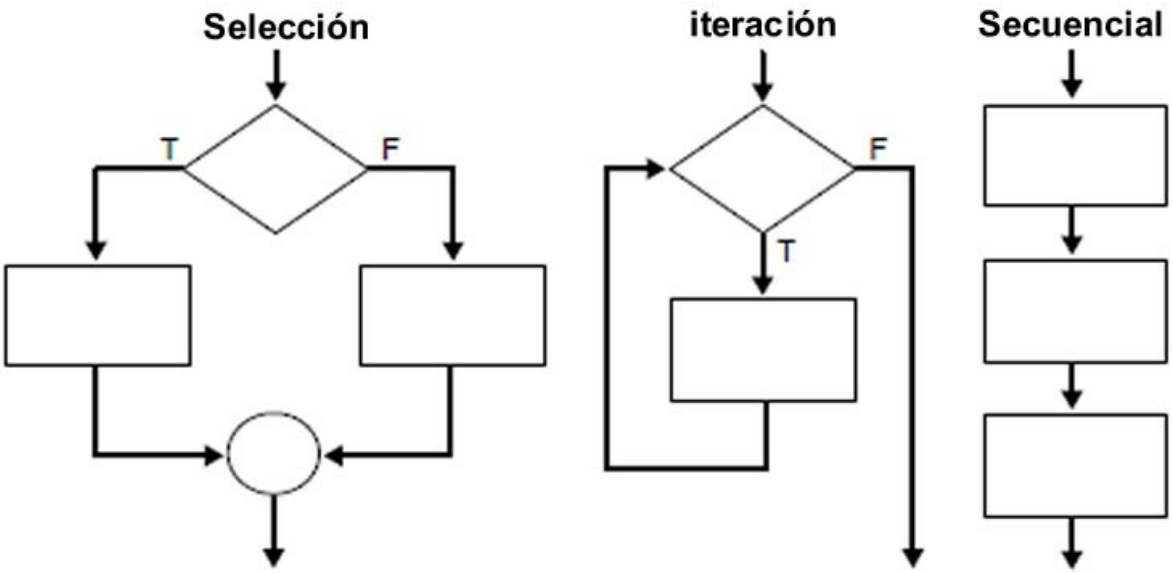
DESARROLLO EN ENTORNO CLIENTE/SERVIDOR

UD3: PORTFOLIO - ESTRUCTURAS DE CONTROL

JavaScript



UD3 - Estructuras de control - JavaScript

1. EVALUACIÓN

El presente documento, junto con sus actividades, cubren los siguientes criterios de evaluación:

RESULTADOS DE APRENDIZAJE	CRITERIOS DE EVALUACIÓN
RA2. Escribe sentencias simples, aplicando la sintaxis del lenguaje y verificando su ejecución sobre navegadores Web.	e) Se han utilizado mecanismos de decisión en la creación de bloques de sentencias. f) Se han utilizado bucles y se ha verificado su funcionamiento. g) Se han añadido comentarios al código. h) Se han utilizado herramientas y entornos para facilitar la programación, prueba y depuración del código.

2. INTRODUCCIÓN. UBICACIÓN LLAMADA FICHEROS .JS

Como vimos en el tema anterior, tenemos dos formas para integrar código JavaScprit:

1. A través de la etiqueta `<SCRIPT>código_en_JavaScript</SCRIPT>` .
2. O insertando un fichero externo de la forma `<SCRIPT src=”fichero.js”></SCRIPT>`.

Siempre va a ser preferible la segunda forma, porque así separamos código de ejecución del HTML de presentación y ambos ficheros quedan más ordenados y limpios.

Pero entonces surge otra cuestión ¿Dónde ubicamos la llamada al fichero .js? Se nos presentan varias opciones, quizás en el <HEAD> o al principio del <BODY>, sin embargo, la mejor opción es al final, justo antes del </BODY>, como en el siguiente ejemplo:

```
<!DOCTYPE>
<html>
	<head>
		<title>Ejemplo JS</title>
	</head>
	<body>
		<p>¡Hola Mundo!</p>
		<script src="prueba.js"></script>
	</body>
</html>
```

El motivo es que, en ese punto, ya se ha cargado la página HTML al completo, por lo que la manipulación de cualquier elemento desde el JavaScript ya es posible. Si el fichero .js se ejecutase antes de una parte del HTML que ha de ser manipulado por JavaScript, obtendríamos un error.

3. COMENTARIOS EN EL CÓDIGO

Como en otros lenguajes de programación, los comentarios de JavaScript se pueden utilizar para explicar el código JavaScript y hacerlo más legible. También se pueden utilizar para evitar la ejecución al probar código alternativo.

Comentarios de una sola línea

Los comentarios de una sola línea comienzan con //. JavaScript ignorará cualquier texto entre // y el final de la línea.

Este ejemplo utiliza un comentario de una sola línea antes de cada línea de código:

```
//La siguiente línea asigna valor a una variable
let a = 5;
//Y a continuación, la mostramos por la consola
console.log(a)
```

También se pueden incluir los comentarios al final de una línea

```
let a = 5; //Inicialización de las variables
```

Comentarios multilínea

Los comentarios multilínea comienzan con /* y finalizan */, cualquier texto entre ambas marcas será ignorado por JavaScript:

```
/*
El código siguiente define
```

```
una función que devuelve el
mayor de dos números
*/
const mayor = (a,b) => a>b?a:b
```

Los comentarios de una línea son los más usuales, y los comentarios multilínea se usan frecuentemente para documentar el código de una manera más formal.

4. SENTENCIAS

Un programa en javaScript es una lista de sentencias, que el navegador ejecutará en orden secuencial. Por ejemplo:

```
let a, b, c;    // Sentencia 1
a = 9;          // Sentencia 2
b = 1;          // Sentencia 3
c = a + b;      // Sentencia 4
```

No es necesario acabar las sentencias por ; pero sí es recomendable hacerlo. JavaScript finalizará cualquier sentencia si la encuentra completa, lo que podría dar lugar a errores, los ; nos ayudan a detectarlos.

JavaScript permite partir sentencias, por ejemplo, esta sentencia:

```
let x = 9; //Se asigna 9 a la variable x
```

y esta otra serían equivalentes:

```
let x =
9; //Se asigna 9 a la variable x
```

Pero si introducimos un salto de línea y lo que queda en la primera línea se puede interpretar como una sentencia completa, JavaScript no mira la siguiente línea:

```
function miFuncion(a) {
  let
  potencia = 10;
```

```
  return a * potencia; //Devuelve a multiplicado por 10
}
```

Pero en el siguiente caso:

```
function miFuncion(a) {
  let potencia = 10;
  return        //Como return es una sentencia válida por sí mismo, no
  a * potencia; //devuelve a*10, si no que devuelve 'undefined'
}
```

Si queremos incluir dos sentencias en una única línea, en ese caso sí es necesario. En ese caso se permiten múltiples sentencias en una línea, pero no es recomendable porque hace un código más difícil de leer por los programadores.

Un bloque de código es un conjunto de sentencias dentro de unas llaves {..}. Indican unas sentencias que deben ser ejecutadas en conjunto, como por ejemplo dentro de una función o en un bucle.

JavaScript es case sensitive. A la hora de crear identificadores podemos usar tres tipos de convenciones a la hora de crearlos: `convenciónCamelCase`, `convención_snake_case`, `ConvencionPascalCase`.

5. MECANISMOS DE DECISIÓN

5.1. if

La sentencia if es similar a la de otros lenguajes de programación. Se evalúa una condición y, si se cumple, se ejecuta el contenido:

```
//Ejecución simple
if(condición) instrucción;
//Ejecución de un bloque de código
if(condición)
```

```
{
      bloque de instrucciones
}
```

La condición se evaluará como un valor booleano. Recuerda que JavaScript se evalúa como falsos los siguientes valores: `false, 0, “”, null, undefined y NaN`. El resto de valores se evalúan como true.

5.2. if..else

Similar a la opción anterior, pero añadiendo la estructura else que se ejecuta cuando no se cumple la condición.

```
//Ejecución de un if..else
if(condición)
{
      instrucción1;//Se ejecuta si la condición se evalúa a true
}
else{
      instrucción2;//Se ejecuta si la condición se evalúa a false
}
```

Ampliación de información:

https://www.w3schools.com/js/js_if_else.asp

5.3. if..else if

Permite anidar múltiples if, que se irán evaluando hasta que se ejecute uno.

```
//Ejecución de un if..else
if(condición1)
{
      instrucción1;//Se ejecuta si la condición1 se evalúa a true
}
```

```
else if(condición 2){

      //Se ejecuta si la condición1 es false y condición2 es true

      instrucción2;

}

else{

      //Se ejecuta si ambas condiciones son false
      instrucción3;

}
```

5.4. switch

En ocasiones, es necesario generar un árbol de condiciones que con sentencias if..else if queda muy farragoso, en su lugar, tenemos la sentencia switch.

La sentencia switch se usa para llevar a cabo diferentes acciones en base a diferentes condiciones. Le pasamos una variable al switch y, cuando ocurre una coincidencia con el mismo valor en un case, el programa ejecuta las declaraciones asociadas correspondientes (usando comparación estricta, ===). Si coinciden múltiples entradas, la primera que se encuentre será la seleccionada.

Si no se encuentra un case que coincida, se busca la cláusula default, que es opcional, y si se existe, se ejecutan las sentencias asociadas. Si no existe cláusula default se continúa la ejecución en la siguiente instrucción posterior al switch. La cláusula default suele ponerse la última después de todos los case, pero es una convención, no porque sea obligatorio por sintaxis.

Al final de cada bloque de una cláusula, se puede poner una sentencia break, que hace que se salga del switch. Si se omite el break el programa continúa la ejecución con la siguiente cláusula que encuentre. Ejemplo

```
//Ejecución simple

switch (sorteo) {

  case 'portátil':

    console.log('Le ha tocado un portátil');
```

```
    break;
  case 'móvil':
    console.log('Le ha tocado un móvil');
    break;
  case 'televisor':
    console.log('Le ha tocado un televisor');
    break;
  case 'raton':
  case 'alfombrilla':
    console.log('Le ha tocado un ');
    break;
  default:
    console.log('Lo lamentamos, por el momento no disponemos de ' + expr
+ '.');
}
//Ejecución de un bloque de código
if(condición)
{
     bloque de instrucciones
}
```

La condición se evaluará como un valor booleano. Recuerda que JavaScript se evalúa como falsos los siguientes valores: `false, 0, “”, null, undefined y NaN`. El resto de valores se evalúan como true.

A diferencia de otros lenguajes, podemos hacer que en las cláusulas case se evalúen condiciones, no sólo valores, esto se consigue con la siguiente sintaxis:

```
let edad = 25;
switch(true) {
```

```
case edad < 18:
console.log("Eres muy joven para entrar");
break;
case edad < 65:
console.log("Puedes entrar");
break;
default:
console.log("Eres muy mayor para entrar");
}
```

Ampliación de información:

https://www.w3schools.com/js/js_switch.asp

6. BUCLES

Como ya sabes, un bucle es una conjunto de instrucciones que se ejecuta de forma repetida. Esta repetición finaliza cuando se cumple (o se deja de cumplir, según se haya programado) una condición que controla dicho bucle. En JavaScript tenemos los bucles while, do-while y for.

6.1. while

Antes de entrar al bucle se evalúa una expresión y si es cierta se ejecuta el código asociado al bucle. Al finalizar la ejecución del código, se vuelve a comprobar la condición, que si sigue cumpliendo, vuelve a ejecutarse el código asociado, y si no, finaliza el bucle:

```
let valor = 1;
while (valor <= 5) { // Imprime 1 2 3 4 5
      console.log(value++);
}
```

Puede ocurrir que la condición no se cumpla ni siquiera la primera vez y nunca se entre en el bucle.

Ampliación de información:

https://www.w3schools.com/js/js_loop_while.asp

6.2. do-while

Es casi igual al bucle while, con una diferencia, el contenido siempre se va a ejecutar al menos una vez:

```
let valor = 1;
while (valor <= 5) { // Imprime 1 2 3 4 5
      console.log(value++);
}
```

6.3. for

El bucle for funciona con tres secciones dentro de los paréntesis:

- Inicialización de parámetros.

  Se ejecuta una única vez, al principio del bucle

- Evaluación de condición.

  Se evalúa una condición y, si es cierta, se ejecuta el bloque de código asociado

- Paso.

  Acciones que se realizan siempre una vez que se termina la ejecución del bloque de código, usualmente, actualizar las variables que controlan la evaluación de condición.

Por ejemplo:

```
let limite = 5;
      for (let i = 1; i <= limite; i++) { // Imprime 1 2 3 4 5
      console.log(i);
}
```

Se puede inicializar más de una variable y puede hacerse más de una acción en el paso:

```
let limit = 5;
for (let i = 1, j = limit; i <= limit && j > 0; i++, j--) {
      console.log(i + " - " + j);
}
```

Ampliación de información:

https://www.w3schools.com/js/js_loop_for.asp

6.4. break y continue

Vamos a tratar el tema de estas dos instrucciones dentro de un bucle mediante una actividad.

Referencia de JavaScript para obtener información:

https://www.w3schools.com/js/js_break.asp

6.5. for of / for in

JavaScript tiene dos bucles adicionales `for of` y `for in` que se usan para recorrer objetos o estructuras de datos iterables. Ya se verán más adelante

7. DEPURACIÓN DE CÓDIGO

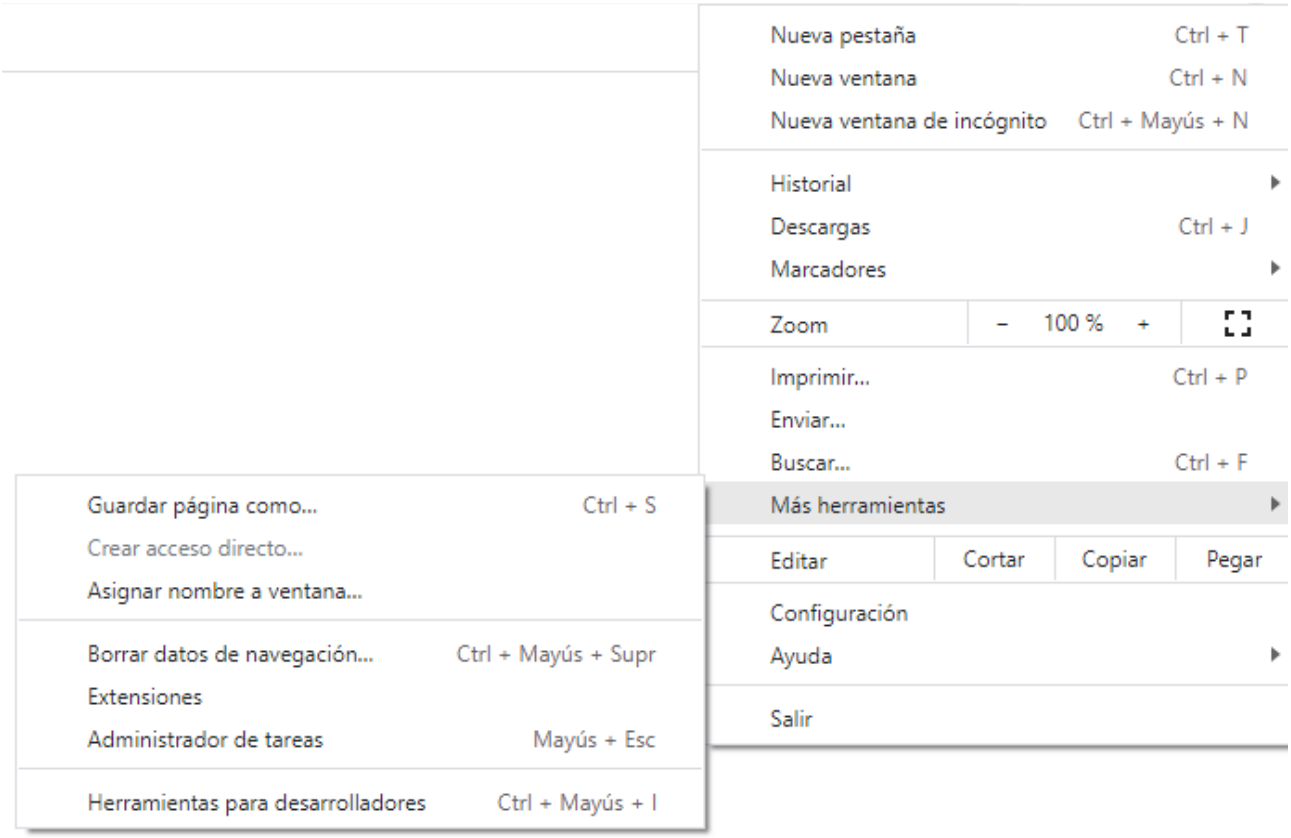
Los navegadores modernos nos permiten realizar una depuración de código muy eficaz, ahorrando tiempo de corrección de errores. Vamos a ver la herramienta de depuración de Chrome, aunque es similar a la de Mozilla, Edge, etc.

Para ver la edición, copiar el siguiente código en un fichero .js

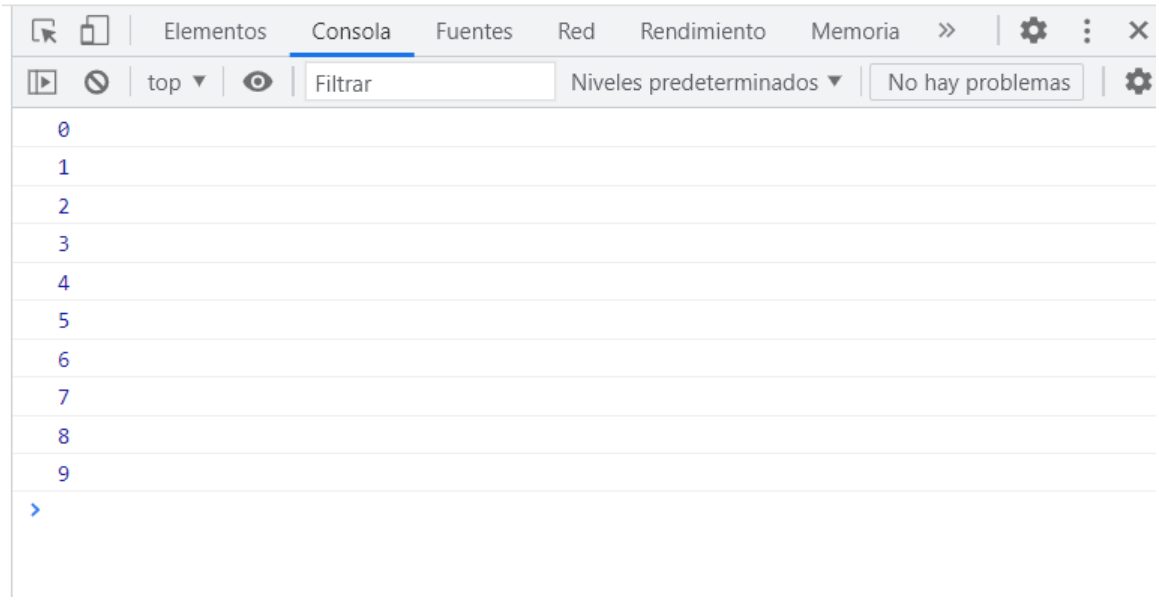
```
'use strict'
let a="Hola";
for (let a=0;a<10;a++){
    console.log(a);
}
```

7.1. Accediendo al depurador

Abrimos en el navegador el fichero HTML que a su vez llamará al fichero .js y abrimos la opción de depuración:

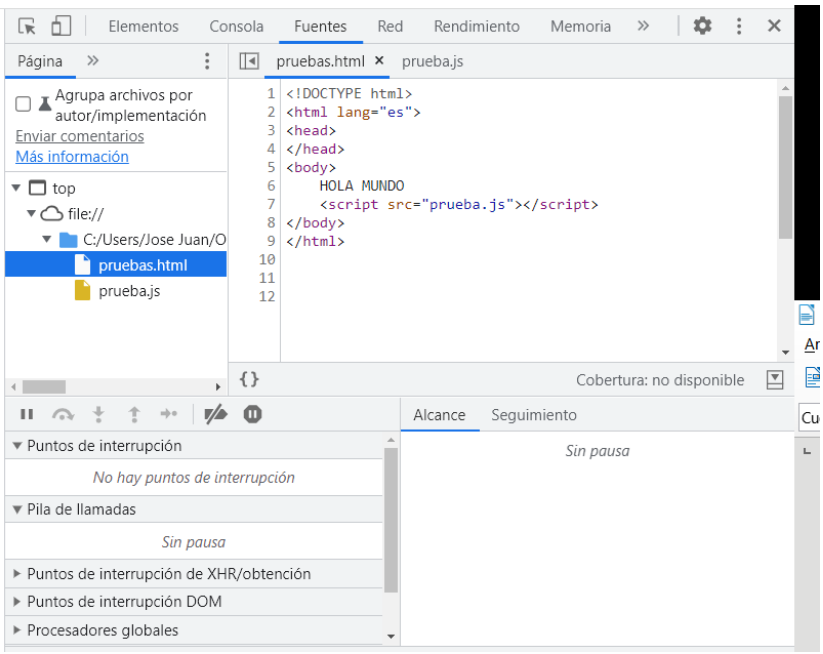


Y nos aparecerá algo parecido a esto:

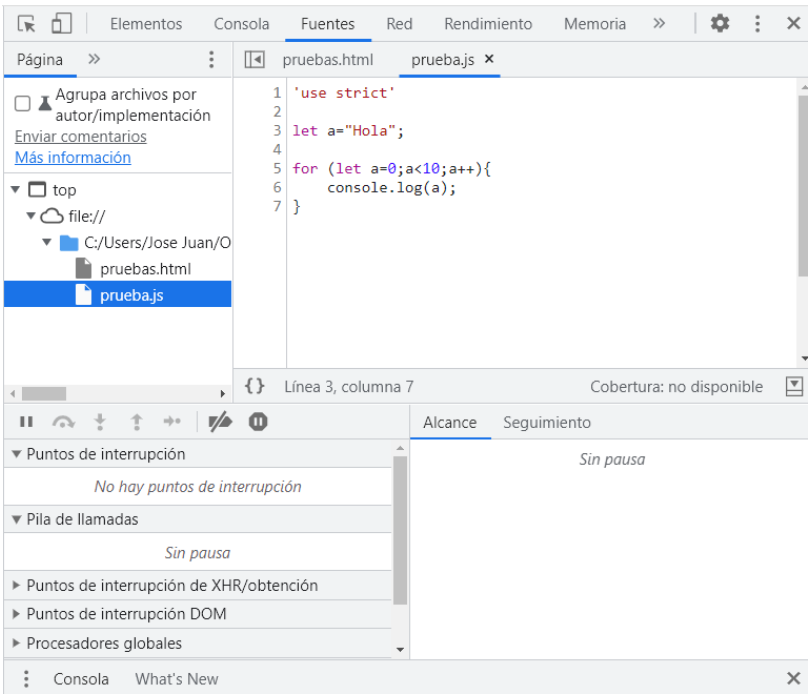


Nos ha mostrado la ventana de depuración, en la que aparecen varias pestañas, siendo las más usadas la de Consola, donde podemos observar la salida de información de JavaScript. Los errores se mostrarán por esta pestaña.

La segunda pestaña importante es Fuentes, en la que podemos ver tanto el fichero HTML:



Como el fichero JavaScript:



Vamos a ver las posibilidades de inspección de código que nos presenta:

- Puntos de ruptura
- Ejecución paso a paso (F9)
- Siguiente función (F10)

- Pasar a la siguiente llamada de función (F11)
- Salir de la siguiente llamada de función (Shift+F11)
- Seguimento