



UD1: Arquitecturas en la web e introducción al lenguaje JavaScript (IV)

Parte 4: Depurar JavaScript con DevTools

María Rodríguez Fernández mariarfer@educastur.org

Al terminar la clase de hoy...

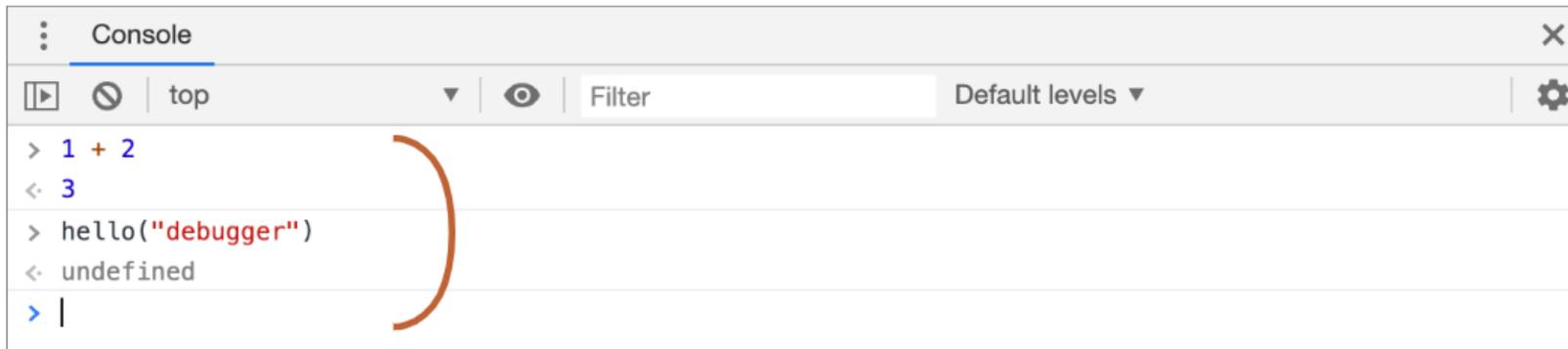
- Aprenderás a usar las principales funcionalidades de la consola de Chrome para depurar JavaScript
- Al menos habrás probado:
 - Varios atajos de teclado
 - A hacer un snippet
 - A poner un breakpoint
 - A añadir un watch

Atajos de teclado

- VSCode:
 - Guardar archivo
 - **Ctrl+S**
 - Mostrar/ocultar barra lateral
 - **Ctrl+B**
 - Indentar el código
 - **Shift+Alt+F**
 - Forzar la finalización de un proceso
 - **Shift + ESC**
- DevTools: **Shift+Ctrl+J** o **F12**

La consola

- En el menú superior “Console” (atajo con **esc**)



```
Console
▶ 🔇 top Filter Default levels ⚙️
> 1 + 2
< 3
> hello("debugger")
< undefined
> |
```

A screenshot of a browser's developer console. The title bar says "Console" with a close button. Below the title bar are controls for "top", "Filter", and "Default levels". The console shows two lines of code and their results: the first line is "> 1 + 2" followed by "< 3"; the second line is "> hello('debugger')" followed by "< undefined". A large orange right curly bracket is drawn on the right side of the console, spanning the two lines of code and their results.

- Muy útil para probar “sobre la marcha” el resultado de una expresión

Opciones útiles de la consola

- Es posible modificar el formato de la salida:

```
console.log('%cHola', 'font-color: yellow; font-weight: bold; background-color: yellow; font-size: 24px');
```

- Mostrará el texto en negrita, tamaño 24px, de color amarillo y fondo negro

- También es útil la orden **console.table** para mostrar el contenido de un array en forma de tabla

```
console.table(["apples", "oranges", "bananas"]);
```

(index)	Values
0	"apples"
1	"oranges"
2	"bananas"

Snippets

- En la consola, sección **Source**, parte izquierda, existe la posibilidad de crear **Snippets** de código
 - Son fragmentos de código con una funcionalidad concreta
- Se pueden ejecutar con **Run** y llamarlos desde la consola para probar su funcionalidad

¿Os acordáis de la tabla de multiplicar?

```
function generarTablaMultiplicar(base, limite) {  
  let resultado="";  
  for (let i = 1; i <= limite; i++)  
    resultado += base + " x " + i + " = " +  
                base* i+"   ---   ";  
  
  return resultado;  
}
```

EJERCICIO PROPUESTO I

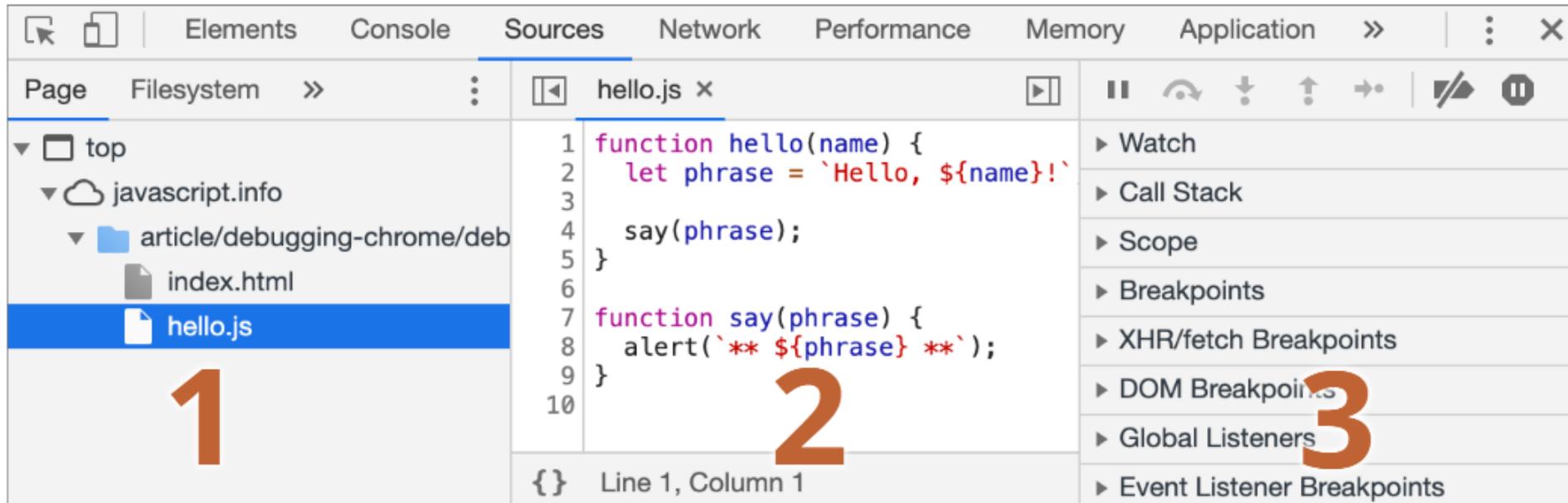
- Crea un **Snippet** “Tabla de multiplicar” con el código para generar la tabla de multiplicar de un número que se pasa como parámetro, parando en el límite que también se le pasa
- Ejecútalo y comprueba que funciona generando la tabla del 4

```
4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20
4 * 6 = 24
4 * 7 = 28
4 * 8 = 32
4 * 9 = 36
4 * 10 = 40
```

```
generartablaMultiplicar(4, 9)
```

Panel “Sources”

- Partes:
 - 1. Zona de recursos
 - 2. Código fuente de los archivos
 - 3. Zona con opciones de debugging

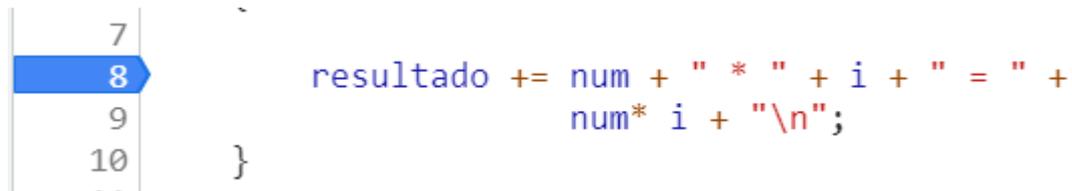


The screenshot displays the Chrome DevTools Sources panel. On the left, the 'Page' tab shows a file system tree with 'hello.js' selected, marked with a large orange '1'. The main area shows the source code of 'hello.js', with the 'say' function definition highlighted, marked with a large orange '2'. On the right, the debugging sidebar is visible, with 'Event Listener Breakpoints' selected, marked with a large orange '3'. The code in the center reads:

```
1 function hello(name) {  
2     let phrase = `Hello, ${name}!`  
3  
4     say(phrase);  
5 }  
6  
7 function say(phrase) {  
8     alert(`** ${phrase} **`);  
9 }  
10
```

Breakpoints

- Son puntos donde queremos que nuestro código se detenga en tiempo de ejecución
- Se insertan haciendo clic sobre el número de línea:



- También se puede usar la instrucción `debugger;` en el código para forzar dicha detención
- Los siguientes botones permiten ir paso a paso decidiendo si entrar o no en el código interno de cada función:



Opciones de ejecución

- Reanudar, continua siguiente instrucción (si no hay más breakpoints, termina la ejecución): F8 
- Siguiente paso (ejecuta la siguiente sentencia): F9 
- Saltar paso (siguiente sentencia, sin entrar en funciones): F10 
- Continuar la ejecución hasta el final de la función actual: Shift + F11 

Si hay un error lo vas a encontrar

function hello(name) {
 name = "John"
 let phrase = `Hello, \${name}!`; phrase = "Hello, John!"
 say(phrase);
}

function say(phrase) {
 alert(`** \${phrase} **`);
}

ver expresiones →

ver los detalles de la llamada externa →

variables actuales →

Paused on breakpoint

1 Watch + ↻
No watch expressions

2 Call Stack

- hello hello.js:4
- (anonymous) index.html:10

3 Scope

- Local
 - name: "John"
 - phrase: "Hello, John!"
 - this: Window
- Global Window

{ } Line 4, Column 3

¿Cómo?

1. La sección **Watch** permite mirar el valor y tipo de datos que van tomando las variables en tiempo de ejecución
 - El valor de las variables también se puede ver posando el ratón sobre la misma en el código
2. La sección **Call Stack** muestra las llamadas anidadas
3. En **Scope** están las variables activas, locales y globales

EJERCICIO PROPUESTO II

- Introduce un error en el código, por ejemplo, cambiando el operador de acumular por el de asignar:

```
resultado += num + " * " + i + " = " + base* i;
```

```
resultado = num + " * " + i + " = " + base* i;
```

- Inserta un *breakpoint* en esa línea y traza el código, comprobando en todo momento que valores van tomando las variables `resultado`, `base` e `i`

EJERCICIO PROPUESTO III

- Pon en práctica todo lo aprendido hoy para depurar los dos códigos ejemplo1 y ejemplo 2 que contienen errores de diversa índole

