PROGRAMACIÓN WEB – SEMINARIO 5

Javascript esencial para la Web

JavaScript

Dr. José Raúl Romero Salguero
jrromero@uco.es

S5-1.
Aspectos básicos y tipos

Javascript: el lenguaje de la Web

- Javascript es el único lenguaje que **se ejecuta de forma nativa** en el navegador
- Diferentes **usos**:
  - *Frontend*: interfaces dinámicas, SPAs, PWAs
  - *Backend*: APIs, microservicios con node.js
  - *Full-stack*: desarrollo extremo a extremo con un único lenguaje
- Se ha convertido en **lenguaje de propósito general**: servidores, dispositivos IoT, CLI, desktop apps, etc.

Javascript: entorno de ejecución

El mismo lenguaje puede crear interfaces, servir datos, automatizar tareas o ejecutar código en la nube

Frontend:

- Ejecución directa en el navegador
- Manipulación del DOM, interacción con el usuario (p.ej. eventos), comunicación con APIs

Backend:

- Con node.js, JS se ejecuta en el servidor
- Uso común: APIs REST, sockets en tiempo real, SSR

Otros:

- CLI, automatización (`npm`, `eslint`), IoT y Edge computing (`Cloudfare workers`)
- *Full-stack*, aplicaciones de escritorio (`electron`), móviles (`react native`)

Comentarios

```
// comentario en una línea simple
/* comentario en varias líneas */
```

Idéntico a la sintaxis de comentarios de Java
Recordemos: sintaxis de comentarios depende del lenguaje:

HTML: `<!-- comment -->`

CSS/JS: `/* comment */`

Java/JS: `//comment`

```
/**
* Represents a book.
* @constructor
* @param {string} title - The title of the book.
* @param {string} author - The author of the book.
 */
function Book(title, author) { }
```

Es recomendable utilizar formato JSDoc para la generación automática de documentación a partir de comentarios: https://jsdoc.app/

Tipos de datos

Variables y tipos

```
var nombre = expression;                                    JS
```

```
var edad = 32;
var peso = 127.4;
var nombreCliente = "Pepe Pérez";                           JS
```

- Las variables se declaran con la palabra clave ***`var`*** (*case sensitive*)
  - Las variables no tienen tipo (conversion automática)
  - Identificadores pueden contener **letras**, **dígitos**, **$**, **_** (no pueden empezar por un dígito)
- Los tipos no están especificados, pero JS tiene tipos ("tipados libremente")
  - **Number, Boolean, String, Array, Object, Function, null, undefined**
  - Puede averiguar el tipo de una variable llamando a ***`typeof operando`***

`var` puede ser una ***variable global*** – se utilizan otras formas de declarar variables y constantes en JS

Valores especiales: `null` y `undefined`

```
var ned = null;
var benson = 9;
var caroline;

//   ned --> null
//   benson --> 9
//   caroline --> undefined
```
JS

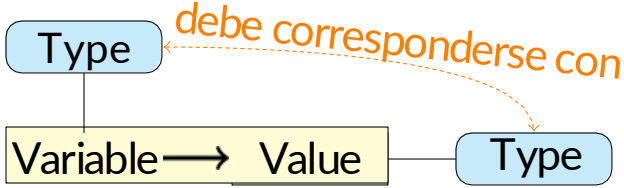
- `undefined`: valor que indica la ausencia de valor inicial
- `null`: existe, pero se le asignó específicamente un valor vacío o nulo

```
console.log(typeof null); // → object
```

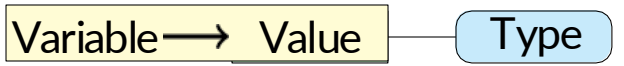
NOTA: Una variable sin declarar produce un `ReferenceError` al acceder.

Javascript: Sistema de tipos

- JS es un lenguaje tipado de forma dinámica y flexible
- Lenguaje de programación estáticamente tipado:
  - Cada **variable** está vinculada a un **tipo** particular
  - Cada **variable** solo puede almacenar un **valor de tipo coincidente**



- Lenguaje de programación tipado dinámicamente:
  - Cada **variable** puede almacenar un **valor de tipo arbitrario**
  - Cada **variable** puede almacenar valores de diferentes tipos en diferentes momentos



Javascript: Sistema de tipos

- Lenguaje de programación fuertemente tipado:
  - En la invocación de una operación, cada **valor de argumento** debe ser de **tipo coincidente**
  - Los valores deben **convertirse explícitamente** al tipo coincidente (a menos que los tipos estén relacionados)

```
2.1 + 5 + Integer.parseInt("7")          // Java
```

- Lenguaje de programación libremente tipado:
  - En la invocación de una operación, cada **valor de argumento** se **convertirá implícitamente** al **tipo coincidente**

```
2.1 + 5 + "7"                  // Javascript
```

Javascript: Sistema de tipos

- Cada **valor** es de un **tipo** particular (o ninguno)

  519 1.9e3 son de tipo número (y solo de ese tipo)
  ’519’ "1.9e3" son de tipo cadena (y solo de ese tipo)

- Pero el **tipo de una variable** no necesita ser declarado.

```
var x; // declara x
```

- El **tipo de una variable** depende del valor que almacena actualmente y el tipo puede cambiar si se le asigna un valor de un tipo diferente.

```
x = 519; // x de tipo number
x = ’519 ’; // x de tipo string
```

Javascript: Sistema de tipos

- Las **declaraciones de funciones** no especifican el tipo de sus parámetros

```
function add(x, y) { return x + y; }
```

- En las **invocaciones a una función**, los tipos de argumentos se ajustarán automáticamente (si es posible)

```
add(519,1.9e3)      // number 2419
add('519',"1.9e3")  // string '5191.9e3'
add(519, '1.9e3')   // string '5191.9e3' '
add(true,1.9e3)     // number 1901
```

✅ Programación más flexible

❌ Potencialmente se producen más errores

Coerción de tipos

- Javascript convierte automáticamente un valor al tipo apropiado según lo requiera la operación invocada (**coerción de tipos**)

```
5  * "3"  //  15
5  + "3"  //  "53"
5 && "3"  //  "3"
```

- El valor **`undefined`** se convierte de la siguiente manera:

Tipo	Default	Tipo	Default	Tipo	Default
bool	false	string	'undefined'	number	NaN

```
undefined || true //          true
undefined +  "-!" //   " undefined -!"
undefined +   1   //           NaN
```

Tipo Number

```
var coste = 99;
var medianGrade = 2.8;
var creditos = 5 + 4 + (2 * 3);
```

- Los enteros y los números reales son del mismo tipo (**int** vs. **double**)
- Mismos operadores: + - * / % ++ -- = += -= *= /= %= **=
- Precedencia de operadores similar a Java

Tipo número: `NaN` e `Infinity`

- El tipo número de JS incluye constantes:
  - **`NaN`** (distingue entre mayúsculas y minúsculas) "no un número"
  - **`Infinity`** (distingue entre mayúsculas y minúsculas) "infinito"

- Las constantes **`NaN`** e **`Infinity`** se utilizan como valores de retorno para aplicaciones de funciones matemáticas que no devuelven un número

  - `Math.log(0)` devuelve `-Infinity`
  - `Math.sqrt(-1)` devuelve `NaN`
  - `1/0` devuelve `Infinity`
  - `0/0` devuelve `NaN`

El operador `NaN` ("*Not a Number*") representa el resultado de una operación numérica inválida o indefinida, como `0/0` o `parseInt("abc")`

Tipo número: `NaN` e `Infinity`

- Los operadores de igualdad y comparación se amplían para abarcar `NaN` e `Infinity`:

```
NaN == NaN                ~ false
Infinity == Infinity      ~ true
NaN == 1                  ~ false
NaN < NaN                 ~ false
1 < Infinity              ~ true
Infinity < 1              ~ false
NaN < Infinity            ~ false
```

```
NaN === NaN               ~ false
Infinity === Infinity     ~ true
Infinity == 1             ~ false
Infinity < Infinity       ~ false
1 < NaN                   ~ false
NaN < 1                   ~ false
Infinity < NaN            ~ false
```

Tipo número: `NaN` e `Infinity`

- Devuelve `False` *si* el valor es `Infinity`

  `bool isFinite(value)`
- No hay ninguna función `isInfinite`
- En JavaScript existen dos formas principales de comprobar si un valor es `NaN`:
  - `isNaN(value)` — función global (heredada del estándar inicial – ES1) – Intenta la coerción
  - `Number.isNaN(value)` — método moderno (introducido en ECMAScript 2015) – Solo `true` si ya es `NaN`
- Ambas se parecen, pero no hacen lo mismo.
- En conversión a un valor booleano:
  - ❑ `NaN` convierte a `false`
  - ❑ `Infinity` convierte a `true`
- En conversión a una cadena:
  - ❑ `NaN` convierte a `'NaN'`
  - ❑ `Infinity` convierte a `'Infinity'`

Tipo `String`

```
var s = "Pepe Juan";
var fNombre = s.substring(0, s.indexOf(" "));  // "Pepe Juan"
var len = s.length;                            // 9
var s2 = 'Melvin Merchant';                    // can use "" or ' '
```

- **Métodos**: charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase
  - **`charAt`** devuelve un `String` de una letra (no hay ningún tipo `char`)
- `length` es una propiedad (no es un método, como en Java)
- Concatenación con **`+`** : `1 + 1` es `2`

  `"1" + 1` es `"11"`

Tipo `String`

```
var count = 10;
var s1 = "" + count;                          // "10"
var s2 = count + " bananas!";                 // "10 bananas!"
var n1 = parseInt("42 es la respuesta"); // 42
var n2 = parseFloat(“una cadena");        // NaN
```

- Las **secuencias de escape** se comportan como en Java : `\' \" \& \n \t \\`
- Para acceder a los caracteres de una cadena, use [*index*] or `charAt`:

```
var firstLetter = s[0];
var firstLetter = s.charAt(0);
var lastLetter = s.charAt(s.length - 1);
```

Tipo `String` – `split`, `join`

```
var s = "the quick brown fox";
var a = s.split(" ");          // ["the", "quick", "brown", "fox"]
a.reverse();                   // ["fox", "brown", "quick", "the"]
s = a.join("!");               // "fox!brown!quick!the"
```

- `split` rompe una cadena en un array usando un delimitador
  - También se puede usar con expresiones regulares rodeadas por `/`:

```
var a = s.split(/[ \t]+/);
```

- `join` combina un array en una sola cadena, colocando un delimitador entre los elementos

Tipo `Boolean`

```JS
var iLikeJS = true;
var ieMola = "IE6" > 0;   // false
if ("PW es genial!") {  /* true */ }
if (0) {  /* false */ }
```

- Las constantes `true` y `false` (*case sensitive*)
- Cualquier valor puede usarse como Boolean:
  - Valores falsos: `0, 0.0, NaN, "", null,` y `undefined`
  - Valores verdaderos: cualquier otra cosa
- Convertir un valor en un Boolean explícitamente:
  - `var boolValue = Boolean(otherValue);`
  - `var boolValue = !!(otherValue);`

Coerción de tipos - `Boolean`

- Al convertir a `Boolean`, los siguientes valores se consideran `false`:
  - El mismo boolean `false`
  - El número `0` (cero)
  - El `string` vacío
  - `undefined`
  - `null`
  - `NaN`
- Cualquier otro valor se convierte en `true`, incluidos
  - Funciones
  - Objetos, en particular, matrices con elementos cero

Arrays

```
var frutas = ['Manzana', 'Banana'];
console.log(frutas.length);
```

```
var stooges = [];
stooges[0] = "Larry";
stooges[1] = "Moe";
stooges[4] = "Curly";
stooges[4] = "Shemp";
```

`Array` en Javascript es un objeto global (objeto tipo lista de alto nivel)

Arrays

- Es posible asignar un valor a `arr.length`
  - Si el valor es mayor que el anterior de `arr.length`, la matriz se 'extiende' con elementos `undefined`
  - Si el valor es menor que el anterior de `arr.length`, se eliminarán los elementos del array con un índice igual o mayor
- Asignar un array a una nueva variable **crea una referencia** al mismo:
  - ~ los cambios en la nueva variable afectan al array original
  - Los arrays también se pasan a funciones por referencia
- La función `slice` se puede utilizar para crear una copia del array:

```
object arr.slice(start, end)
```

devuelve una copia de los elementos del array con índices entre `start` y `end`

Arrays - Funciones

- Javascript no tiene estructuras de datos “pila” o “cola”, pero tiene funciones de pila y cola para arrays:
  - `number arr.push(value1, value2,...)` – agrega uno o más elementos al final; devuelve el número de elementos en el array resultante
  - `mixed arr.pop()` extrae y devuelve el último elemento
  - `mixed arr.shift()` extrae y devuelve el primer elemento
  - `number arr.unshift(value1, value2,...)` inserta uno o más elementos al comienzo del array; devuelve el número de elementos del array resultante

S5-2.
Flujo del lenguaje

Operadores lógicos

- Relacionales: `> < >= <=`
- Lógicos: `&& || !`
- Igualdad: `== != === !==`
  - La mayoría de los operadores lógicos convierten automáticamente los tipos

```
5 < "7"                 //true
42 == 42.0              //true
"5.0" == 5              //true
```

  - Los `===` y `!==` son pruebas estrictas de igualdad; comprueba tanto el tipo como el valor:

```
"5.0" === 5             //false
```

Declaración `if/else`

```
if (condición) {
  sentencias;
} else if (condición) {
  sentencias;
} else {
  sentencias;
}
```
JS

- ¡**Javascript** permite casi cualquier cosa como condición!

Bucle `for`

```
for (inicialización; condición; incremento) {
  sentencias;
}
```

```
var sum = 0;
for (var i = 0; i < 100; i++) {
  sum = sum + i;
}
```

```
var s1 = "hello";
var s2 = "";
for (var i = 0; i < s1.length; i++) {
  s2 += s1[i] + s1[i];
}
// s2 stores "hheellllloo"
```

Bucle `while`

```
while (condición) {
  sentencia;
}
```

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

También existe `break` y `continue`, si bien su uso debe hacerse solo cuando sea estrictamente necesario

Bucles – `break` y `continue`

- `break` detiene la ejecución del bucle y puede usarse también con `while-`, `do while-`, y `for`

```
while (v < 100) {
  if (v == 0) break ;
  v ++
}
```

- `continue` detiene la ejecución de la iteración actual y mueve la ejecución a la siguiente iteración

```
for (x = -2;x <= 2;x++) {
    if (x == 0) continue;
    document.writeln ("10/" + x + "= " +
}   (10/x));
```

```
10 / -2 = -5
10 / -1 = -10
10 / 1 = 10
10 / 2 = 5
```

Arrays – función `forEach`

- La forma recomendada de iterar sobre todos los elementos de un array es con **`for`**

```
for (index = 0;index < narray.length;index ++) {
    ... narray[index ] ...
}
```

- Una alternativa interesante es el uso de la función **`forEach`**:

```
var callback = function (elem,index,arrayArg) {
    statements
}
narray.forEach(callback);
```

- **`forEach`** toma una función como argumento e itera sobre todos los índices
- Pasa como parámetros el elemento actual (*`elem`*), el índice actual (*`index`*) y un puntero al array (*`arrayArg`*)
- Los valores de retorno de esa función se ignoran
- La función **puede tener efectos secundarios**

Arrays – función forEach

```
var rewriteNames =function (elem , index , arr) {
  arr[index] =elem.replace(/(\w+)\s(\w+)/, "$2,$1");
}

var myArray =['Dave Jackson','Ullrich Hustadt'];

myArray.forEach(rewriteNames);


for (i=0; i < myArray.length ; i++) {
  document.write('['+i+']='+myArray[i] + ' ');
}
document.writeln("<br/>");
```

```
[0] = Jackson , Dave [1] = Hustadt , Ullrich < br >
```

Iteradores avanzados

FOR/OF

- Funciona en objetos **iterables** como `Array`, `String`, `Set`, `Map`, etc.
- Es ideal para recorrer colecciones con orden definido.

```
let datos = [1,2,3,4,5,6,7,8,9], sum = 0;
for (let elem of datos) {
  sum += elem;
}
console.log(sum) // ➔ 45
```

```
let obj = {x:1,y:2,z:3};
for (let prop in obj) {
  console.log(prop+": "+obj[prop]);
}
➔x:1
➔y:2
➔z:3
```

FOR/IN

- Funciona en elementos **enumerables** como cualquier objeto.
- Se diferencia en que recorre los nombres de sus propiedades, incluyendo las heredadas, no sus valores.

S5-3.
Variables

Declaración de variables ES2015__ `var`, `let`, `const`

- Con la llegada de ES6 (ECMAScript 2015) se añaden **tres tipos de variables**:
  - `var` – se utiliza para declarar una variable, permitiendo su inicialización:
    1. Las variables con `var` son procesadas **ANTES** de la ejecución del código
    2. Su ámbito es su contexto de ejecución (dentro de funciones, privadas)
    3. Fuera de la función (en ámbito global de un *script* en el navegador), `var` declara una variable global (y se asocia a `window`)
    4. Realiza ***hoisting*** con declaración a `undefined`

NOTA: Cuando se asigna al contexto del objeto `window`, JS puede llegar a reasignar su valor sin pedir "permiso" ni notificarlo al programador (p.ej. porque otra librería o componente define otra función/variable de igual nombre) – errores difíciles de localizar sin que JS notifique problema alguno

Declaración de variables ES6__ `var`, `let`, `const`

- `let` –Para resolver el problema con `var`, ES6 introduce `let` → declara una **variable con alcance local**
  1. El alcance de la variable es el bloque en que se define
  2. La variable no puede ser redeclarada en el mismo bloque (¡**el valor sí**!)
  3. `let` realiza ***hoisting***, como `var`, pero la variable queda en **TDZ** (temporal dead zone) hasta su declaración

```
function f() {
  for (let x = 0; x < 2; x++) {
    console.log(x);
    for (let x = 10; x < 12; x++) {
      console.log(x);
  } }
  // console.log(x); -> Provoca un error, x no esta definida
}
```

Fuente: oddbytes.net

```
function f() {
console.log(x) // salida: undefined
var x = 10; }
```

```
function f() {
console.log(x) // salida: ReferenceError: x is not defined
let x = 10; }
```

Declaración de variables ES6__ `var`, `let`, `const`

- `const` – crea una constante a nivel de bloque
  1. Su valor es de sólo lectura
  2. Debe estar inicializado desde el momento de la declaración
  3. El valor asignado **no es inmutable** pero sí no es reasignable

```
const x = 1;
x = 2; // Error – No se puede reasignar
```

```
const x = [1,2,3];
x.push(4); // CORRECTO. Modifica el contenido del array
x=[5,6]; // Error – No es posible reasignar
```

Según recomendación de ECMAScript 2015:

- Usar **`const`** siempre que sea posible
- Usar **`let`** cuando el valor pueda cambiar
- Evitar **`var`** (*hoisting* y ámbito confuso)

Buenas prácticas de legibilidad y consistencia

- Un **código legible**
  - Reduce errores y tiempo de depuración
  - Facilita el trabajo en equipo y la revisión
  - Mejora la mantenibilidad del proyecto
- La **claridad** prevalece sobre la concisión
- Si se utilizan **comentarios**, deben describir **el por qué**, no el qué

- Un código se escribe una vez y se lee cientos (y por distintas personas)

```
let x=a=>{if(a>10)return!0;else return!1};
```

```
function isValidAge(age) {
  return age > 10;
}
```

Buenas prácticas de legibilidad y consistencia

- Los nombres deben **reflejar la intención**
  - Funciones: verbos → `getUser()`, `calculateTotal()`, `fetchData()`.
  - Variables: sustantivos → `userList`, `price`, `totalScore`.
  - Booleans: prefijos como *is*, *has*, *can* → `isActive`, `hasPermission`.

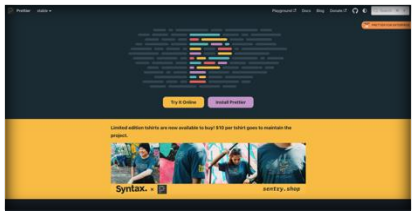
- El estilo siempre debe ser **consistente**
  - Usar camelCase para variables y funciones.
  - PascalCase para clases y componentes.
  - Evitar abreviaturas poco claras (usr, cfg, tmp).

```
let x = true;
```

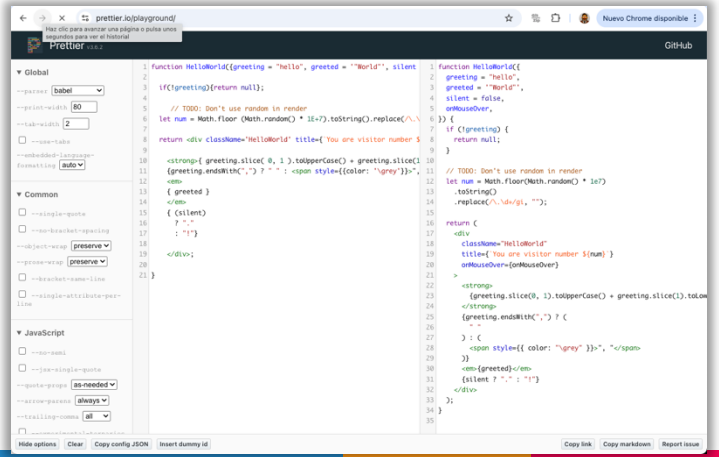
```
let isValidAge= true;
```

Buenas prácticas de legibilidad y consistencia

- La consistencia visual mejora la **comprensión del código**
  - Utiliza (2) espacios de indentación
  - Una única instrucción por línea
  - Líneas menores de 100 caracteres
  - Comillas simples o dobles, pero siempre igual en el proyecto
  - Usar siempre `;`
- Se pueden utilizar **formateadores automáticos** para evitar despistes, diferentes criterios, etc.
  - Un ejemplo conocido es ***Prettier***



https://prettier.io/



Buenas prácticas de legibilidad y consistencia

- Utilizar siempre un **patrón predecible**:
  - Declaraciones en la parte superior
  - Funciones relacionadas agrupadas
  - Evitar anidar funciones sin razón o intención conocida
  - Mantener estructura de código con un orden lógico: declaración→ procesamiento → salida
- Evitar hacer uso de coerción implícita
- Evitar abusar de los operadores cortos
- La **claridad** prevalece sobre el ingenio

```
return !!(user && user.permissions &&
user.permissions.admin);
```

```
if (!user) return false;
return user.permissions?.admin === true;
```

S5-4.
Funciones

Declaración de funciones

```
function fn_name(param1, param2, …) {
  sentencias;
  [return valor;]
}
```

```
function myFunction() {
  alert("Hola!");
  alert("Como estas?");
}
```

- Las declaraciones de las funciones se pueden evaluar en respuesta a los eventos del usuario
- El nombre de la función distingue entre mayúsculas y minúsculas
- `fn_name.length` se puede usar para determinar el número de parámetros formales (en la declaración de la función)

Tipos de funciones

1. Funciones nombradas
2. Expresión de función (anónima o nombrada)
3. Funciones con argumentos por defecto
4. Funciones con parámetros REST (ES6)
5. Función flecha (ES6)

Funciones anónimas

```
function(parameters) {
  statements;
}
```

- Javascript permite declarar funciones anónimas (sin nombre)
- Puede almacenarse como una variable, adjuntarse como un controlador de eventos, etc.
- Javascript posee un objeto de lenguaje (*built-in object*) llamado **arguments**
  - Permite invocar a **`length`** para obtener el número de argumentos pasados a la llamada (invocación)

Funciones anónimas - Ejemplo

```
window.onload = function() {
  var ok = document.getElementById("ok");
  ok.onclick = okayClick;
};

function okayClick() {
  alert("siiiuuu");
}
```

- Lo siguiente también es legal (aunque más difícil de leer y con peor estilo):

```
window.onload = function() {
  document.getElementById("ok").onclick = function() {
    alert("siiiuuu");
  };
};
```

Ámbito global de variables

```
var count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}

incr(4);
incr(2);
console.log(count);
```

count, **incr**, y **reset** son globales

- Se debe evitar el uso de variables globales
- Otros archivos JS pueden verlas y modificarlas

Ámbito global de variables

```
function everything() {
  var count = 0;
  function incr(n) {
    count += n;
  }
  function reset() {
    count = 0;
  }

  incr(4);
  incr(2);
  console.log(count);
}

everything();
```

1 símbolo global: **everything**

- El ejemplo anterior mueve todo el código a una función
- Las variables y funciones declaradas dentro de otra función son locales, no globales

Funciones anidadas

- Las declaraciones de funciones se pueden anidar en Javascript
- Las funciones internas tienen acceso a las variables de las funciones externas
- Por defecto, las funciones internas no se pueden invocar desde fuera de la función en la que se definen

```
function bubble_sort( array ) {
   function swap(i, j) {
      var tmp = array [i];
      array [i] = array [j];
      array [j] = tmp;
   }
   if (!(array && array.constructor == Array))
      throw ("El argumento NO es un Array")
   for ( var i=0; i< array.length ; i++) {
      for ( var j=0; j< array.length - i; j++) {
         if ( array [j+1] < array [j]) swap(j, j+1)
   } }
   return array
}
```

S5-5.
Carga del *script*

Importación del *script*

Importación tradicional

Cada vez que un navegador encuentra un elemento `script`, de forma predeterminada, deja de analizar el HTML restante hasta que el elemento `script` se haya descargado y procesado por completo

~ Puede ocasionar mala experiencia de usuario (esperas) y errores

- "Solución segura": colocar los elementos del `script` al final, antes del elemento `body`
- "Buena solución": utilizar el atributo `async` o `defer` de `<script>` para cambiar el comportamiento predeterminado de descarga y procesado
- "Solución moderna": Uso de **módulos ES6**

Importación del *script*

Importación async/defer

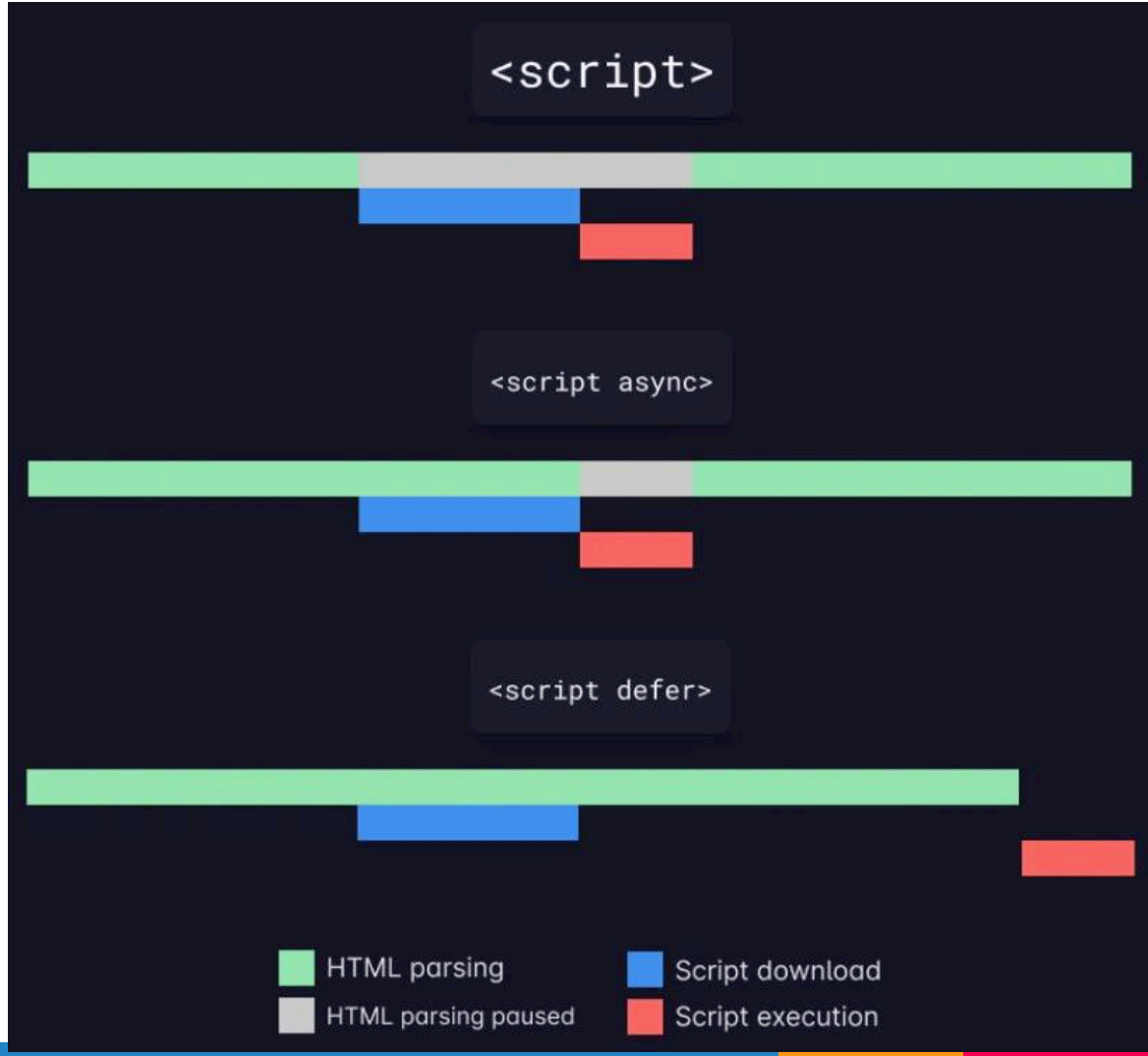
```
<script src="jsLib1.js" async></script>
<script src="jsLib2.js" async></script>
```

- *Async* descarga de forma asíncrona el *script*, sin detener el análisis de HTML. Una vez descargado, detiene el renderizado del HTML y ejecuta el *script*. No se garantiza la ejecución de los *scripts* asíncronos en el orden de aparición en el documento

```
<script src="jsLib1.js" defer></script>
<script src="jsLib2.js" defer></script>
```

- *Defer* descarga de forma asíncrona el *script*, sin detener el análisis de HTML. La ejecución del *script* también es diferida, manteniendo el orden de aparición en el documento. No hay bloqueo en el renderizado

Importación del *script*



Source: 30secondsofcode.org

Importación del *script*

Importación de módulos

```
<script type="module"
src="app.js" ></script>
```

En la actualidad, los navegadores modernos soportan la carga de *scripts* como **módulos ES6**, que se ejecutan de forma diferida (similar a `defer`) y segura (modo estricto, `'use strict'`)

¡Veremos los módulos más adelante!

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Script Loading Example</title>
</head>
<body>
  <h1>Script loading demo</h1>

  <!-- Script clásico -->
  <script src="old.js"></script>

  <!-- Script moderno (modo módulo) -->
  <script type="module">
    console.log("Running in module
mode");
  </script>
</body>
</html>
```

Recursos y lecturas

- Javascript Reference – *API Completa*

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference
- Métodos de String (MDN)

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
- Métodos de Array (MDN)

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
- Métodos de Number (MDN)

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
- Métodos de Math

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
- JS Playground

  https://playcode.io/

Recursos y lecturas

- “You don’t know JS yet” (K. Simpson)

  https://github.com/getify/You-Dont-Know-JS
- “The Modern Javascript Tutorial” (Javascript.info)

  https://github.com/getify/You-Dont-Know-JS
- Coerción de tipos (MDN)

  https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion
- Funciones (MDN)

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions
- Carga de scripts (MDN)

  https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/script