Desarrollo de apps con tecnologías web: JavaScript

Programación de aplicaciones para dispositivos móviles

Curso 2024-25

JavaScript: Introducción (1/2)

- JavaScript: Lenguaje de programación que permite dotar de lógica a los documentos HTML **(comportamiento).**
- Creado por Brendan Eich, empleado de Netscape, en el año 1995 para poder incorporar funcionalidad que se ejecutara en el mismo navegador.
- Lenguaje dinámico orientado a prototipos, con características funcionales.
- Lenguaje de script que se ajusta a la especificación de ECMAScript (especificación definida en ECMA-262 para crear un lenguaje de scripting de propósito general).

JavaScript: Introducción (2/2)

- Pensado para escribir de manera rápida código que se ejecuta en distintos *contextos* (caracterizados por un conjunto de *objetos* cuyos métodos pueden invocarse).
- JavaScript es un lenguaje interpretado y está tipado dinámicamente (es decir, no es necesario especificar tipos, ya que no se comprueba la corrección de tipos en expresiones y sentencias).
- JavaScript puede asociarse con documentos HTML. Los *scripts* pueden acceder a modelos de objetos que representan dichos documentos (DOM), así como a otros objetos globales que representan el entorno de ejecución (objetos del navegador…).

JavaScript: Ubicación del código en el fichero HTML (1/2)

- Se puede incluir código JavaScript tanto en el `<head>` como en el `<body>`.
- Pero hay que tener en cuenta el flujo de carga de una web: cuando se descarga un fichero JS hay que procesar el código y ejecutarlo.
- Mientras se ejecuta el fichero JS el navegador se queda bloqueado.

JavaScript: Ubicación del código en el fichero HTML (2/2)

- En el `<head>` se iniciará la carga antes que el resto de recursos, pero bloqueará el "renderizado" de la página mientras se ejecuta.
- Al final de `<body>` permitirá que se "renderice" la página, pero tardará más en ejecutarse.
- Cuando todas las imágenes, scripts, CSS, etc. se han cargado, se dispara el evento **load** de window.
- Es muy común incluir el código que inicializa la ejecución del programa en el manejador (*handler*) de ese evento:

```
window.onload = function () {
  // << TODO>>
};
```

JavaScript: Scripts inline

- Incluir el código JavaScript dentro del fichero HTML utilizando la etiqueta `<script>`.

```
<script> console.log("Hello, world!"); </script>
<script>
  function myFunction() {
     document.getElementById("demo").innerHTML =
         "El texto del párrafo ha cambiado.";
  }
  myFunction();
</script>
```

JavaScript: Scripts externos

- Referencia al fichero donde se localiza el código JavaScript dentro del fichero HTML utilizando la etiqueta `<script>`.

```
<script src="js/game.js"></script>
```

JavaScript: Ejecución asíncrona (1/2)

- Mientras JavaScript se está ejecutando se bloquea todo lo demás (incluida la UI).
- La manera de programar JavaScript en la Web es suscribiéndose a eventos y ejecutando código solamente cuando esos eventos se disparan (*callbacks*).
- Si los eventos no se disparan, o los *callbacks* no se llaman, no se ejecuta código.
- Los *callbacks* de los eventos se ejecutan de uno en uno (hilo único).
- Los *callbacks* se ejecutan inmediatamente cuando el evento se dispara (el resto del código espera su finalización).

JavaScript: Ejecución asíncrona (2/2)

- Los *callbacks* de los eventos pueden recibir un argumento con información sobre el evento.
- El argumento es un objeto de tipo **Event**.

```
var button = document.getElementById('start');
button.addEventListener('click', function (evt) {
    console.log(evt);
});
```

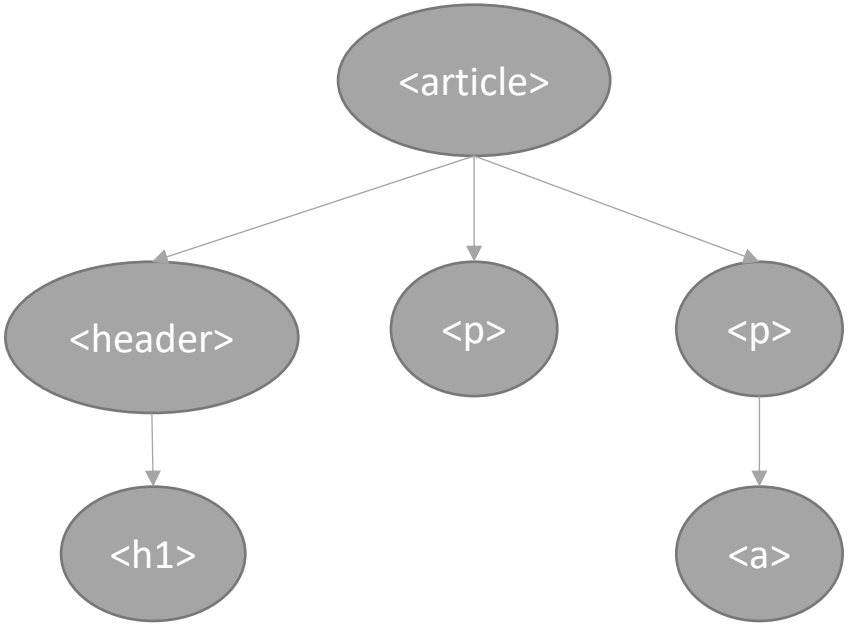
JavaScript: Ejemplo

```
button.onclick = function (evt) {
    console.log("Click");
}

// ..

button.trigger('click');
// El botón no se desactiva hasta que el handler de "click"
// no haya acabado de ejecutarse.
button.disabled = true;
```

Document Object Model: Ejemplo de sección (1/2)

```
<article>
  <header>
    <h1>Nota informativa</h1>
  </header>
  <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Duis dictum tempus mollis. Cras eget augue eu dui
elementum commodo non vel odio.
  </p>
  <p>
    Ut nec dignissim turpis, vel ultrices neque. Cras porta
eros id pulvinar tincidunt. Proin quis sem pharetra,
malesuada justo sit amet, malesuada purus
    <a href="http://wikipedia.org">Acceso wikipedia</a>
  </p>
</article>
```

Document Object Model: Ejemplo de sección (2/2)



Document Object Model: Operaciones

- Leer/escribir las propiedades de los elementos, y usar sus métodos.
- Eliminar elementos del DOM.
- Insertar nuevos elementos en el DOM.

Document Object Model: Acceso por id

- Seleccionar un elemento (las ID’s deben ser únicas).

```
<button id="show-fullscreen">Fullscreen</button>
```

```
var button = document.getElementById('show-fullscreen');
```

Document Object Model: Acceso por selector

- Usar la sintaxis de los selectores CSS para localizar uno (o varios) elementos.

```
// Selecciona el primer párrafo que encuentra
var paragraph = document.querySelector('p');
// Selecciona el primer elemento con clase .warning
// (atributo class).
var label = document.querySelector('.warning');
// Selecciona TODOS los párrafos.
var allPars = document.querySelectorAll('p');
```

Document Object Model: Iterar sobre una lista

- `querySelectorAll` no devuelve un array, sino una colección de nodos: NodeList.
- No podemos utilizar métodos de Array sobre ella. Sin embargo, tiene la propiedad `length` y el operador [], por lo que se puede acceder a través de un bucle:

```
var buttons = document.querySelectorAll('button');
for (var i = 0; i < buttons.length; i++) {
    // Ocultar botones
    buttons[i].style = "display: none";
}
```

Document Object Model: Navegación

- Se accede al **padre** de un elemento con la propiedad `parentNode`.
- Se accede a la lista de **hijos** de un elemento con `childNodes`.
- Se accede al **hermano anterior o siguiente** con `previousSibling` y `nextSibling`.
- De esta manera se puede recorrer el DOM en cualquier dirección.

Document Object Model: Propiedades (1/3)

- `innerHTML`: El interior (o contenido) del elemento.
- Se puede escribir HTML dentro, creando de manera dinámica nuevos elementos del DOM.

```
button.innerHTML = 'Aceptar';
p.innerHTML = 'Párrafo con <b>negrita</b>';
```

Document Object Model: Propiedades (2/3)

- `style`: Permite aplicar estilos CSS inline (tienen la máxima prioridad).
- Muy útil para ocultar/mostrar y habilitar/deshabilitar elementos.
- También permite leer la propiedad CSS.
- **NOTA:** `display:none` es universal, pero para mostrar un elemento se puede elegir entre varios valores, aunque los más comunes son `inline`, `inline-block` y `block`.

```
var button = document.getElementById('btnOK');
button.style="display:none"; // Oculta el botón.
button.style.display = "none"; // Oculta el botón.
button.style="display:inline-block;" // Muestra el botón.
button.style.display="inline-block;" // Muestra el botón.
var previousDisplay = button.style.display;
if (previousDisplay == "none") {
   console.log("No visible");
else {
   console.log("Visible");
}
```

Document Object Model: Propiedades (3/3)

- `classList`: Permite acceder a las clases CSS de un elemento
- El elemento classList resulta útil para cambiar el aspecto de la UI en función de las interacciones.

```
button.classList.add('loading');
button.classList.remove('loading');
button.classList.contains('loading'); // Consulta.
button.classList.toggle('loading'); // No funciona en IE.
```

Document Object Model: Manipulación (1/3)

- Insertar elementos nuevos a través de `createElement`.

```
var button = document.createElement('button');
button.innerHTML = 'Comenzar';
button.setAttribute('type', 'button');
button.setAttribute('id', 'start’);
```

```
<button type="button" id="start">Comenzar</button>
```

Document Object Model: Manipulación (2/3)

- Cuando se crea un elemento con `createElement` está huérfano y no se verá “renderizado” en la página.
- Hay que añadirlo como “familiar” de algún otro elemento a través de `appendChild`, `insertBefore`…

```
document.body.appendChild(button);
```

Document Object Model: Manipulación (3/3)

- Reemplazar elementos a través de `replaceChild`.
- Eliminar elementos a través de `removeChild`.

```
var button = document.getElementById('start');
button.parentNode.removeChild(button);
```

Document Object Model: Eventos (1/8)

- Los elementos del DOM disparan eventos que admiten suscripción. Ejemplos: al hacer clic en un botón, al cambiar el texto de un elemento `<input>` y cuando se selecciona un **checkbox**.
- El elemento **window** también dispara eventos, por ejemplo, **load** y **resize**.

Document Object Model: Eventos (2/8)

- Existen dos maneras de escuchar los eventos disparados por un elemento:
  - Utilizando el método `Event.addEventListener`.
  - Utilizando los manejadores *(handlers) on-event* (por ejemplo, `onclick` y `onfocus`).

Document Object Model: Eventos (3/8)

- Solamente se va a dar un manejador por evento.
- La suscripción o eliminación del manejador se realiza a través de una asignación:

```
// Realizar suscripción.
var button = document.getElementById('start');
button.onclick = function (evt) { /* ... */ };
// Cancelar la suscripción.
button.onclick = null;
```

Document Object Model: Eventos (4/8)

- Se permiten varias suscripciones al mismo evento (*event listeners*).
- Es la manera recomendada y más segura:

```
var sayHi = function () { /* */ };
// Suscripción.
var button = document.getElementById('start');
button.addEventListener('click', sayHi);
// Cancelar la suscripción
button.removeEventListener('click', sayHi);
```

Document Object Model: Eventos (5/8)

- El **bubbling** es la metáfora con la que se explica cómo se comportan los eventos del DOM.
- Cuando un elemento dispara un evento, se propaga hacia arriba en el árbol del DOM.

```
<section>
    <button id="start">Pulsar</button>
</section>

var section = document.querySelector('section')
section.addEventListener('click', function () {
    console.log('Has pulsado el botón...');
});
```

Document Object Model: Eventos (6/8)

- Se puede **interrumpir el bubbling** a través de métodos como `Event.stopPropagation`.

```
var button = document.getElementById('start');
button.addEventListener('click', function (evt) {
    evt.stopPropagation();
});
```

Document Object Model: Eventos (7/8)

- Se puede **cancelar el bubbling** a través de métodos como `Event.preventDefault`.
- Esto **no es interrumpir el bubbling**.
- Es cancelar el evento y evitar acciones por defecto asociadas a él. Es útil para interceptar formularios, o cambiar el comportamiento de un menú. Ejemplo:

```
<a href="file.zip" download>Download zip</a>
```

```
var link = document.querySelector('a');
link.addEventListener('click', function (evt) {
    // El navegador no detectará que se ha pulsado el enlace.
    evt.preventDefault();
});
```

Document Object Model: Eventos (8/8)

- Otro Ejemplo de cancelación:

```
var button = document.getElementById('start');
button.addEventListener('click', function (evt) {
   //Evitamos que haga el submit directamente con esta acción.
    evt.preventDefault();
    var self = this;
    //Antes de hacer submit se cambia su comportamiento.
    //...
    self.form.submit();
});
```

JavaScript: Escritura

- JavaScript no soporta directamente instrucciones de escritura.
- La escritura debe ser habilitada por los objetos de contexto. Por ejemplo, es posible escribir en el documento HTML:

```
document.write("generado desde JavaScript");
```

- Si el navegador soporta una *consola*, a efectos de navegación, también es posible escribir en dicha consola:

```
console.log("generado desde JavaScript");
```

JavaScript: Sintaxis básica (1/2)

- Las sentencias básicas se terminan en ;
- Se pueden definir bloques de sentencias mediante {…}
- Las variables, llamadas **identificadores** en JavaScript, deben comenzar con una letra, _ o $. El resto de los caracteres pueden ser letras, dígitos, _, o $. **No puede comenzar con un dígito**.
- JavaScript introduce también distintas palabras reservadas que no pueden utilizarse como identificadores (por ejemplo, `break`, `case`, `catch`, `class` y `const`).
- JavaScript distingue entre minúsculas y mayúsculas.

JavaScript: Sintaxis básica (2/2)

- El código de caracteres de JavaScript es UNICODE.
- Es posible partir *strings* en varias líneas con \

```
var text = "¡Hola \
Mundo!";
```

- Los comentarios son del tipo de Java:
  - Comentario de línea: `//`
  - Comentario más flexible para una línea o varias líneas: `/* … */`

JavaScript: Variables (1/2)

- Una forma de declarar las variables es con la palabra reservada **`var`** (no se especifica tipo, ya que JavaScript está tipado dinámicamente). Ejemplo: `var carName;`
- La asignación se realiza con **=**. Ejemplo: `carName="Mustang";`
- Las variables pueden inicializarse en el momento de su declaración. Ejemplo: `var carName="Mustang";`
- Las variables declaradas dentro de un bloque son locales a dicho bloque.
- Aunque no es aconsejable, se pueden asignar valores a variables no declaradas (las variables se crearán en dicho momento, pero de forma global).
- En una misma sentencia **var** es posible declarar varias variables. Ejemplo: `var lastName = "Orwell", age = 30, job = "escritor";`
- Las variables no inicializadas tienen valor **undefined**.

JavaScript: Variables (2/2)

- **let**: otra forma de declarar variables a partir de ECMAScript 2015 (conocido de manera más popular como ES6) . Crea las variables en el momento de su definición, pero no antes, como ocurre con **var (hoisting)**.
- **const**: otra forma de declarar variables y que tiene el mismo comportamiento en ámbito que **let**, pero en este caso impide que la referencia cambie.
- Lo declarado con **const** podría cambiar, pero no el objeto al que apunta la variable. Ejemplo:

```
const myObject= {x: 1, y: 2};
myObject.x = 4; // ok
myObject = {z: "trabajando"}; // error
const myText = "¡Hola!"
myText = "¡Adiós!" //error
```

JavaScript: hoisting (1/2)

- **JavaScript:** Variables con o sin hoisting => **var** vs **let**

```
var i; // Variable declarada, pero sin valor definido.
console.log(i); // undefined
i = 1;
console.log(i); // 1

console.log(x); // undefined (permite hoisting)
var x = 1;

console.log(z); // ReferenceError: Cannot access 'z'
let z = 1;      // let crea variables a partir de su definición.
```

JavaScript: hoisting (2/2)

- **var:**

```
function f() {
  for (var i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
      console.log('i: ', i, ' j: ', j);
    }
  }
  console.log(i); // 10
  console.log(j); // 10
}
```

```
function f() {
  var i;
  var j;
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
      console.log('i: ', i, ' j: ', j);
    }
  }
  console.log(i); // 10
  console.log(j); // 10
}
```

- **let:**

```
function f() {
  for (let i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
      console.log('i: ', i, ' j: ', j);
    }
  }
  console.log(i); // ReferenceError: Cannot access 'i'
  console.log(j); // 10
}
```

JavaScript: Ámbito de las variables

- Los parámetros y las variables declaradas en el interior de una función son *locales* a la función.
- El resto de variables (incluso aquellas creadas directamente por asignación) son *globales*.
- La vida de las variables globales comienza cuando se carga una página, y finaliza cuando la página se cierra.

JavaScript: Tipos de datos (1/6)

- Tipos primitivos:
  - booleanos
  - undefined
  - números
  - enteros grandes
  - cadenas de caracteres
  - símbolos
- Como JavaScript está tipado dinámicamente, una misma variable puede contener valores de diferentes tipos en distintos momentos de una ejecución:

```
var x; //undefined
x = true //booleano: true, false
x = 5; // número 1234.5, -Infinity, +Infinity
x = '¡Hola, mundo!'; //cadena "¡Hola, mundo!" "¡Hola, 'Juan'!" 'Hola, "Juan"'
```

JavaScript: Tipos de datos (2/6)

- Objetos: Los objetos en JS no son primitivos. Tienen datos y métodos. Ejemplos:

```
var myObject = {}; // Otros equivalentes:[], null
var myCode = function () { return 42; };
```

- Se pueden reconocer porque responden de manera distinta al operador `typeof`. Ejemplos:

```
typeof true; //boolean
typeof 1234.5; //number
typeof '¡Hola, mundo!'; //string
typeof {}; //object
typeof function () { return 42; }; //function
typeof null; //object
```

JavaScript: Tipos de datos (3/6)

- Objetos literales: lista de cero o más pares de **nombres de propiedad y valores** asociados de un objeto, entre llaves ({}). Los valores podrían ser otros objetos, arrays, números, cadenas o funciones.
  - Literales: `var myShape={ width:5, height: 23, color:"rojo"}`
    `//myShape.width, myShape.height, myShape.color`
- Arrays: colecciones de datos ordenados.
  - Literales: `["mozilla", "firefox", ["ie1", "ie2", "ie3"]]`
  - Sus elementos pueden ser de tipos heterogéneos.
  - Tienen una propiedad `length`. Ejemplo: `myList.length`
  - Se acceden a sus propiedades por índice (número entero).
  - Se indexan desde 0 hasta `length-1`.
- Método `push`: añadir un elemento al final del array. Ejemplo: `myList.push("chrome");`
- Método `pop`: quitar un elemento por el final. Ejemplo: `myList.pop();`

JavaScript: Tipos de datos (4/6)

- Método `splice`: insertar o borrar elementos de cualquier posición. Ejemplos:

```
myList = ["mozilla", "firefox", "chrome"];
myList.splice(2, 0, "edge"); // añade edge antes de chrome
myList.splice(2, 2, 'safari'); // reemplaza edge y chrome con safari
myList.splice(0, 0, 'opera'); // añade opera al principio de la lista
```

- Se puede cambiar cualquier valor a través del operador de asignación. Ejemplo:

```
myList[0] = "ie";
```

JavaScript: Tipos de datos (5/6)

- Permite acceder a un valor que no existe, que se puede recuperar o asignar posteriormente:

```
var myList = ["mozilla", "firefox", "chrome"];
console.log(myList);
console.log(myList.length);
var item = myList[10];
console.log(typeof item); // undefined
myList[10] = 'opera'; //Se extiende el array hasta ese índice.
console.log(myList);
console.log(myList.length);
```

```
[ 'mozilla', 'firefox', 'chrome' ]
3
undefined
[ 'mozilla', 'firefox', 'chrome', <7 empty items>, 'opera' ]
11
```

JavaScript: Tipos de datos (6/6)

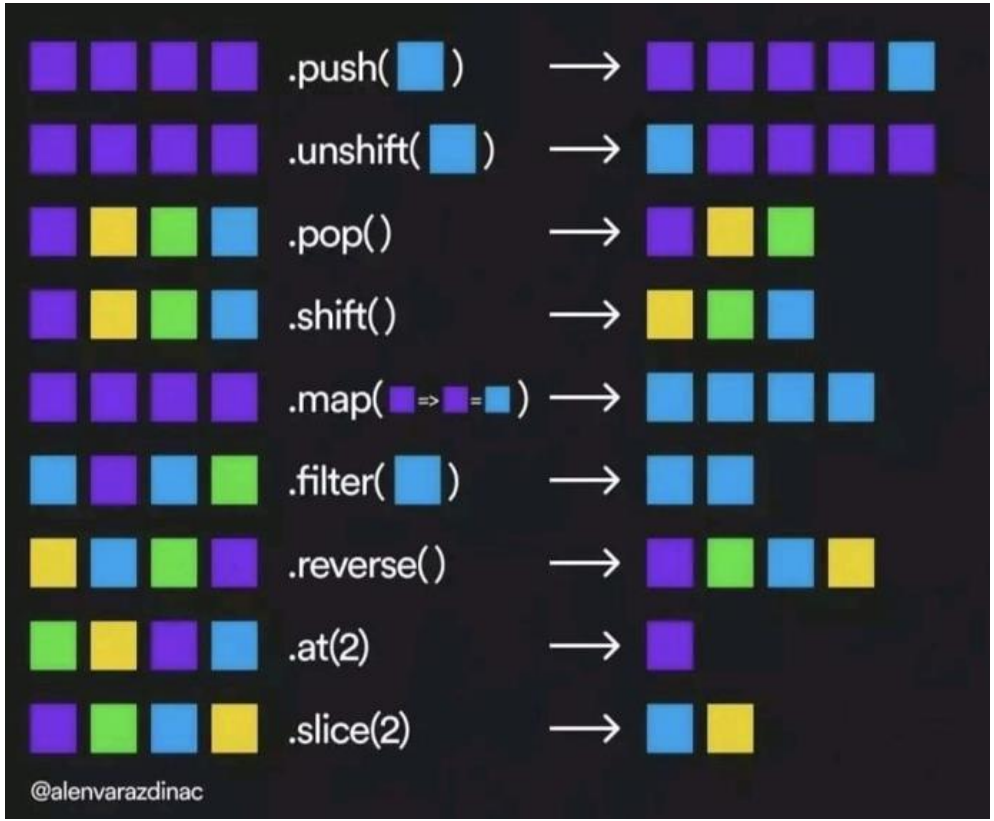
- Arrays y objetos son del tipo `object` por lo que se tiene que utilizar el método `Array.isArray()` para distinguirlos.

```
var obj = {}; // un objeto vacío
var arr = []; // una lista sin elementos
typeof obj; // object
typeof arr; // object
Array.isArray(obj); // false
Array.isArray(arr); // true
```

- **NOTA:** El valor `null` tiene su propio tipo (`null` es de tipo `null`), pero la máquina virtual dirá que es de tipo `object` por motivos históricos.

```
typeof null; // object
```

JavaScript: Resumen métodos Arrays



JavaScript: Herencia

- La herencia en JavaScript se realiza a partir de **prototipos**: cada objeto tiene un prototipo y para crear nuevos objetos se utilizará dicho prototipo.
- Método `create()`. Ejemplo:

```
<script>
  var myShape = {
    width:5,
    height: 23,
    color:"rojo",
    showColor: function(){
       alert(this.color)
    }
  };
  var shape1 = Object.create(myShape);
  console.log(shape1.color);
  shape1.showColor= function(){ ... };
  shape1.showColor();
  myShape.showColor();
</script>
```

JavaScript: Funciones

- Las funciones se representan por su propio tipo (`function`), aunque en realidad son también de tipo `object`.
- Declaración: `function` *nombre* `(`p_1`, …, `p_k`) { `*cuerpo*` }`
- Ejemplo:

```
function prod(p1, p2) {
    return p1 * p2;
 }
```

- Invocación: *nombre*`(`v_1`, .., `v_k`)`
- Ejemplo: `prod(5*6,7)`
- JavaScript no comprueba que el número de parámetros reales coincida con el número de parámetros formales (si se pasan menos parámetros reales, los restantes formales estarán indefinidos; si se pasan más, los que sobran se ignoran).

JavaScript: Eventos

- La conexión de JavaScript con HTML se realiza a través de **eventos**.
- Distintos elementos pueden soportar distintos tipos de eventos.
- Con los atributos se puede especificar qué código JavaScript debe ejecutarse cuando se dispara un evento. Ejemplo:

```
<button onclick="displayDate()">¿Qué hora es?</button>
```

- Algunos eventos comunes en HTML:
  - `onchange`: El elemento ha cambiado.
  - `onclick`: El usuario ha hecho *clic* sobre el elemento.
  - `onmouseover`: El usuario está moviendo el ratón sobre el elemento HTML.
  - `onmouseout`: El usuario ha movido el ratón fuera del elemento HTML.
  - `onkeydown`: El usuario ha pulsado una tecla.
  - `onload`: El navegador ha terminado de cargar la página.

JavaScript: Operadores

- Aritméticos: +, -, *, /, % (módulo en división entera), ++ (incremento), -- (decremento).
- Asignación: =, +=, -=, *=, /=, %=
- +, aplicado a *strings*, realiza la concatenación de dichos *strings* (también si un argumento es un string y otro es un número: el número se convierte a *string*).
- Comparación: ==, === (exactamente igual, compara dos objetos y decide si son iguales), != (distinto a ==), !== (distinto a ===, compara dos objetos y decide si no son iguales), >, <, >=, <=
- Lógicos: &&, ||, !
- Condicional: $c?v_0:v_1$

JavaScript: Instrucciones de control (1/4)

- Las instrucciones de control también son similares a las de Java.
- Instrucciones `if-then` y `if-then-else`. *Ejemplos:*

```
if (time<20) {
  x="¡Buenas tardes!";
}

if (time<10) {
  x="¡Buenos días!";
}
else if (time<20) {
  x="¡Buenas tardes!";
}
else {
  x="¡Buenas noches!";
}
```

JavaScript: Instrucciones de control (2/4)

- Instrucción `switch`. Ejemplo*:*

```
var day=new Date().getDay();
switch (day) {
case 6:
  x="Hoy es sábado";
  break;
case 0:
  x="Hoy es domingo";
  break;
default:
  x="Esperando el fin de semana";
}
```

- Instrucción `for`. Ejemplo:

```
for (var i=0;i<cars.length;i++) {
  document.write(cars[i] + "<br>");
}
```

JavaScript: Instrucciones de control (3/4)

- Instrucción `for-in`. Ejemplo:

```
for (car in cars) {
   document.write(car);
}
```

- Instrucción `while`. Ejemplo:

```
while (i<5) {
  x=x + "El número es: " + i + "<br>";
  i++;
}
```

- Instrucción `do-while`. Ejemplo:

```
do {
  x=x + "El número es " + i + "<br>";
  i++;
} while (i<5);
```

JavaScript: Instrucciones de control (4/4)

- Instrucción `break`. Ejemplo:

```
for (i=0;i<10;i++) {
  if (i==3)break;
  x=x + "El número es " + i + "<br>";
}
```

- Instrucción `continue`. Ejemplo:

```
for (i=0;i<=10;i++) {
 if (i==3)continue;
  x=x + "El número es " + i + "<br>";
}
```

- Etiquetas: En JavaScript es posible etiquetar bloques de código con `label`, y utilizar `break` o `continue` con dichas etiquetas.

JavaScript: Errores

- JavaScript introduce un mecanismo de manejo de errores/excepciones análogo al de Java.
- Mediante `throw` es posible lanzar una excepción (el argumento de `throw` puede ser cualquier valor). Ejemplo: `if(isNaN(x)) throw "no es un número";`
- Mediante `try-catch` es posible capturar las excepciones. Ejemplo:

```
try {
  alert("¡Hola!");
}catch(err) {
  txt="Se ha producido un error en la página.\n\n";
  txt+="Descripción del error: " + err.message + "\n\n";
  txt+="Pulsar OK para continuar.\n\n";
  alert(txt);
}
```

JavaScript: Declaraciones y modo estricto

- En JavaScript las declaraciones de un bloque se procesan antes de procesar el código (es decir, las variables y las funciones pueden declararse en cualquier orden).
- Sin embargo, las inicializaciones de las variables en las declaraciones se llevan a cabo en el orden de aparición (es decir, primero se crean las variables, y posteriormente, durante la ejecución normal, se inicializan).
- Es posible (y recomendable) ejecutar JavaScript en **modo 'estricto'**. En este modo, se hacen más comprobaciones que en modo normal (por ejemplo, las variables que se usan deben haber sido declaradas previamente, lo que impide crear accidentalmente nuevas variables globales).
- Para forzar modo estricto debe iniciarse el correspondiente bloque como `"use strict"`.

JavaScript: Módulos (1/5)

- Los módulos permiten concentrarse en su uso y lógica sin “importar” el resto del sistema.
- Los módulos mantienen un conjunto de nombres privado (espacio de nombres) por lo que no va a haber colisión entre nombres de variables y funciones de distintos módulos.
- JavaScript no estaba pensado para grandes sistemas, por lo que inicialmente no contemplaba el uso de módulos.
- Existen muchos sistemas de módulos en JavaScript, pero se utilizará el sistema de módulos del ES6.

JavaScript: Módulos (2/5)

```
// archivo module.js
function myFunction() {
  console.log("this is my function");
}
module.exports = myFunction;

// archivo user_module.js
const importedFunction = require('./module')
importedFunction();
```

JavaScript: Módulos (3/5)

```
// pi.js
export const pi = 3.141592;
export function addPi(x) { return pi + x; }

// math.js
import {pi, addPi} from './pi.js'
console.log(addPi(5));

// math2.js (importamos aquí todo lo exportado del módulo pi.js
import * as p from './pi.js'
console.log(p.pi);
```

JavaScript: Módulos (4/5)

- `export default`: se puede elegir uno de los valores exportados y hacer que sea el valor por omisión. Ejemplo:

```
// pi.js
const PI  = 3.141592;
export default PI;
// Otra manera equivalente:
export { PI as default };

// pi_default.js
import PI_Import from './pi.js';
// Otra manera equivalente:
import { default as PI_Import } from './pi.js'
```

JavaScript: Módulos (5/5)

- Los módulos se diferencian de un “script” normal con las siguientes características:
  - Las variables son locales al módulo.
  - Usan el modo “estricto” por defecto.
  - Se cargan de forma asíncrona ➡ más velocidad.
- Hay que tener en cuenta que JavaScript también permite importar librerías sin necesidad de módulos: no hay que hacer `import` pues ya formará parte del espacio de nombres. Ejemplo en el `<head>` de `index.html`:

```
<!-- index.html -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/phaser/3.18.1/pha
ser.min.js"></script>
```

JavaScript: Experimentando con JavaScript (1/3)

- La mejor forma de experimentar con JavaScript es utilizar la consola de node.

```
$node --use_strict
> 40 + 2
42
> var point = { x: 1, y: 1 };
undefined
> point
{ x: 1, y: 1 }
> point.x
1
```

- Para limpiar la pantalla pulsar ctrl+l.
- Para salir de node pulsar ctrl+c dos veces seguidas.

JavaScript: Experimentando con JavaScript (2/3)

- Otra opción es crear un programa .js y utilizar console.log () para mostrar las expresiones por pantalla. Ejemplo con **test.js**:

```
// test.js
console.log(40 + 2);
var point = { x: 1, y: 1 };
console.log(point);
console.log('Coordenada X:', point.x);
```

- Y ahora ejecutar el programa con node:

```
$node --use_strict test.js
```

JavaScript: Experimentando con JavaScript (3/3)

- Para que los módulos funcionen en node hay que guardar el archivo con extensión `.mjs` y ejecutar node con el siguiente parámetro:

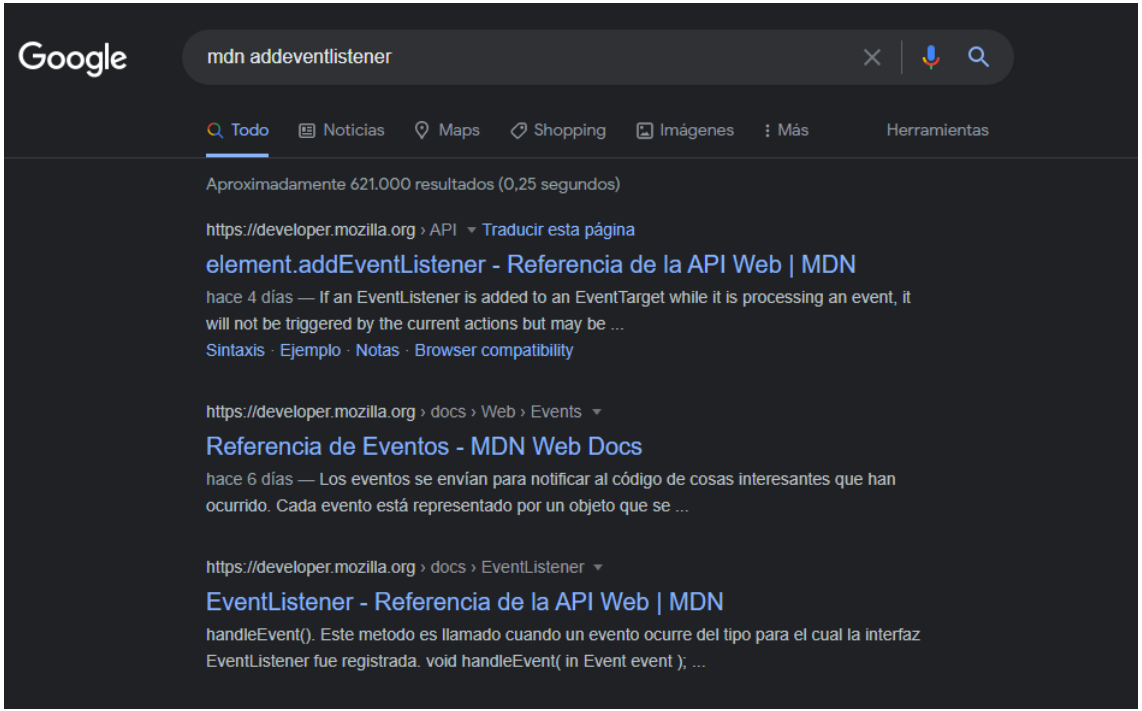
  `--experimental-modules` (va cambiando según la versión de node).

- Para que los módulos ES6 funcionen en HTML hay que añadir el atributo `type="module"` a la etiqueta `<script>`. Ejemplo:

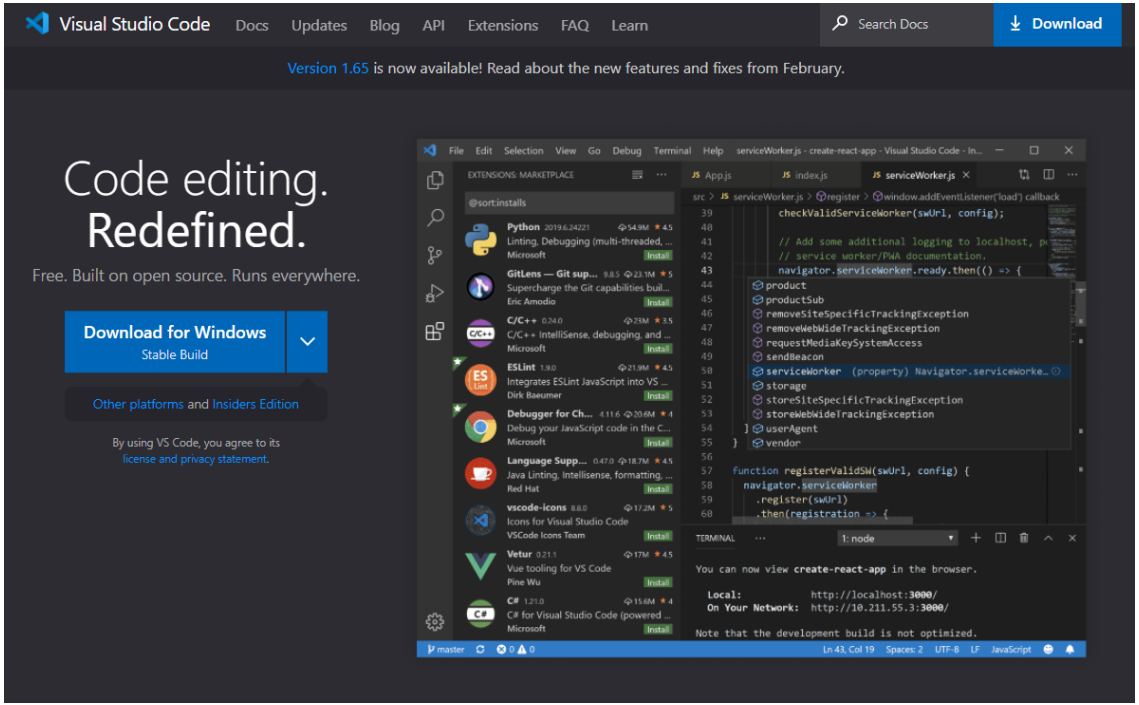
```
<script type="module">
  import PI from './pi.js';
  document.pi = PI;
</script>
```

Documentación: MDN

- http://developer.mozilla.org
- Truco: Añadir **mdn** a cualquier búsqueda.



Visual Studio Code



https://code.visualstudio.com/

Enlaces de interés

- https://developer.chrome.com/docs/devtools/javascript/
- https://developer.mozilla.org/es/docs/Web/JavaScript