



A decorative background featuring a grid of thin blue lines. A thick vertical blue line runs down the left side. Two horizontal blue lines are positioned in the center-right area. The top and bottom edges of the slide have a light blue gradient with white wavy lines.

JavaScript

¿Qué es Javascript?

□ ¿Por qué estudiar JavaScript?

JavaScript es uno de las 3 lenguajes que todos los desarrolladores web DEBEN aprender:

- ❖ **HTML** para definir el contenido de las páginas web.
- ❖ **CSS** para especificar el diseño de páginas web.
- ❖ **JavaScript** para programar el comportamiento de las páginas web.

¿Qué es Javascript?

- ❖ Javascript es un lenguaje de programación interpretado utilizado en páginas Web, por lo que no es necesario compilar los programas para ejecutarlos.
- ❖ Fué desarrollado para el navegador Netscape Navigator 2.0
- ❖ Todos los navegadores modernos interpretan el código Javascript integrado dentro de las páginas Web.
- ❖ Es muy importante entender que el lenguaje JS está orientado al lado del cliente, por lo que no puede acceder a bases de datos almacenados en servidor o buscar datos en una tabla.
- ❖ JS ha recuperado esplendor con la aparición de nuevas librerías y frameworks (Ajax, jQuery, Dojo...)

¿Qué es ECMAScript?

Mientras que **JavaScript** es el lenguaje de programación, **ECMAScript** es una especificación de lenguaje de programación.

ECMAScript son las reglas, y dice hacia dónde debe ir el lenguaje. Desde el 2015 cada versión tiene el año en su nombre, desde ese año también decidieron que todos los años saldrían nuevas funcionalidades.

Versiones ECMAScript

- ECMAScript 5.1 (2011). Versión anticuada
- ECMAScript 2015. (ES6). Nuevo Estándar.
- ECMAScript 2016. (ES7).
- ECMAScript 2017. (ES8).
- ECMAScript 2018. (ES9).

¿Qué es Javascript?

Javascript es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas.

Una página web dinámica es aquella que incorpora efectos como:

- ❖ JavaScript puede cambiar elementos HTML
- ❖ JavaScript puede cambiar atributos HTML
- ❖ JavaScript puede cambiar los estilos HTML (CSS)
- ❖ JavaScript puede validar datos

Antes de empezar

El código de un script o programa Javascript consta de los elementos que se dan a continuación:

- Cuerpo principal.
 - ✓ Javascript en el mismo documento html.
 - ✓ Javascript en un archivo externo.
 - ✓ Javascript en respuesta a un evento.
- Manipulador de eventos.
- Funciones.

Cuerpo Principal

○ Javascript en el mismo documento Xhtml.

- El código JavaScript en línea va inscrito entre las etiquetas HTML **<SCRIPT>** y **</SCRIPT>**
- Este código se ejecuta en el momento en que el navegador procesa la parte del documento HTML donde se encuentra el código

JavaScript en línea

```
<SCRIPT>
document.write("Hola a todos")
</SCRIPT>
```

- Inclusión explícita de texto en un documento HTML:
document.write

Cuerpo Principal

- EJEMPLO Javascript en el mismo documento Xhtml.

```
<html>
  <head>
    <title>Fichero 1</title>
    <script LANGUAGE= "Javascript1.2">
      function funcion1() {
        alert ("Se ejecuta funcion1");
      }
    </script>
  </head>
  <body>
    <script>
      funcion1(); // Se llama a la función la primera vez
      funcion1(); // y aquí otra vez
    </script>
  </body>
</html>
```

Cuerpo Principal

○ Javascript en un archivo externo

- Se comporta como el JavaScript en línea en el documento HTML salvo que el código se encuentra en un fichero externo.
- El URL del fichero que contiene el código se indica como valor del atributo **src** de la etiqueta `<SCRIPT>`
- A pesar de no contener código, es necesario incluir la etiqueta final `</SCRIPT>`
- El fichero referenciado no debe contener las etiquetas `<SCRIPT>` ni `</SCRIPT>`

JavaScript desde un fichero

```
<SCRIPT src="saludo.js"></SCRIPT>
```

Cuerpo Principal

- EJEMPLO Javascript en un archivo externo.

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="Problema2.js">
            </script>
            <script>
                function funcionInterna() {
                    document.write('Hola');
                    document.write('<br>');
                    document.write('Esta es la función INTERNA');
                    document.write('<br>');
                    document.write('Adios');
                    document.write('<br>');
                }
            </script>
        </head>
        <body>
            <script>
                funcionInterna();
                funcionExterna();
            </script>
        </body>
    </html>
```

Cuerpo Principal

- Javascript en un archivo externo.

```
//PROBLEMA2.JS
```

```
function funcionExterna() {
    document.write('Hola');
    document.write('<br>');
    document.write('Esta es la función EXTERNA');
    document.write('<br>');
    document.write('Adios');
    document.write('<br>');
}
```

Cuerpo Principal.

○ Javascript en respuesta a un evento.

- Ciertos elementos HTML son sensibles a eventos: movimientos del ratón, pulsaciones sobre botones, ...
- Se puede incluir código JavaScript que actúe en respuesta a dichos eventos
- Este código se ejecuta cuando se genera el evento y no en el momento de cargar la página HTML
- Esta forma de incluir código JavaScript permite modificar un documento HTML en función de la interacción del usuario

JavaScript en respuesta a un evento

```
<BODY onLoad="window.alert('Hola una vez más') ">
```

- Ventana de avisos: **window.alert**

http://www.w3schools.com/js/tryit.asp?filename=tryjs_intro_lightbulb

Etiqueta noscript

Algunos navegadores no disponen de soporte completo de Javascript, otros navegadores permiten bloquearlo parcialmente.

En estos casos, es habitual que si la página web requiere Javascript para su correcto funcionamiento, se incluya un mensaje de aviso al usuario indicándole que debería activar Javascript para disfrutar completamente de la página.

La etiqueta **<noscript>** se debe incluir en el interior de la etiqueta **<body>** (normalmente se incluye al principio de **<body>**).

El mensaje que muestra **<noscript>** puede incluir cualquier elemento o etiqueta Xhtml.

Etiqueta noscript

```
<html>
    <head>
        <script type="text/javascript" src="Problema2.js">
        </script>
        <script>
            ***
        </script>
    </head>
    <body>
        <script>
            funcionInterna();
            funcionExterna();
        </script>
        <noscript>
            document.write('ERROR DE EJECUCION DE script');
        </noscript>
    </body>
</html>
```

Manipuladores de Eventos

- Los eventos son acciones que ocurren respuesta a algo que el usuario realiza, como por ejemplo, hacer clic con el ratón en un botón.
- Los manipuladores de eventos se introducen en el código html como si se tratase de etiquetas, su sintaxis es:

```
<ETIQUETA_html manipuladorEvento = "Código  
Javascript">
```

Manipuladores de Eventos

```
<html>
  <head>
    <title>Fichero 1.3</title>
    <script LANGUAGE= "Javascript1.2" >
      function laTercera() {
        alert ("Activación del evento onFocus");
      }
    </script>
  </head>
  <body>
    <FORM>
      <INPUT TYPE="text" SIZE=60 NAME="texto" VALUE="Evento
      onFocus" OnFocus="laTercera ()">
    </FORM>
  </body>
</html>
```

Funciones

Las funciones son porciones o fragmentos de código Javascript que permanecen a la espera de ser llamados.

Cuando se define una función, se le está indicando al intérprete que disponga de ese código para ser ejecutado desde otra sección de código situado o en el cuerpo principal o en un manipulador de eventos o en otras funciones.

```
<script LANGUAGE= "Javascript">
    function multiplicarPor10 (valor) {
        return 10 * valor;
    }
</script>

<script LANGUAGE="Javascript">
    function escribirpantalla (texto) {
        document.write ("El texto escrito es ...");
        document.write (texto);
    }
</script>
```

Léxico

Comentarios

Javascript admite dos tipos de comentarios:

//Comentario de una línea

/* Comentarios de ocupan
Varias líneas de texto */

// Se utiliza cuando el comentario ocupa una sola línea.

Mientras que cuando el comentario va a ocupar varias líneas se empieza por poner /* y se cierra el comentario con los caracteres */.

Léxico

Punto y coma

- Es característico en los lenguajes de programación restringir las sentencias a una por línea y el uso del ; para delimitar sentencias.
- En JavaScript no hay correspondencia entre sentencias y líneas, una sentencia puede ocupar una o varias líneas.
- No hay ninguna diferenciación en su empleo o no por parte del intérprete de Javascript.

Léxico

Si es **obligatorio** su uso cuando:

- Una sentencia ocupa más de una línea para ver cuando finaliza la misma.

Cital = "Esto es un texto demasiado largo para que ocupe una sola línea por que su finalidad era esa";

- Se sitúan dos sentencias en la misma línea.

Cital = "Esto es un texto corto"; cantidad1=100

NOTA: como su uso no está demás se recomienda poner el ; siempre al finalizar cada instrucción.

Léxico

Declaraciones de variables

- Las *variables* se utilizan para el almacenamiento de valores e información de distinta naturaleza, permitiendo que los programas puedan emplear datos para realizar cálculos y almacenar los resultados.
- Para declarar una variable en Javascript, podemos realizarla de varias formas:

variableEjemplo;

var variableEjemplo;

Let variableEjemplo;

Léxico

Declaraciones de variables

- `variableEjemplo; //Se declara una variable global`
- `var variableEjemplo; //Declara una variable local`
- `Let variableEjemplo; //Declara una variable dentro del bloque al que pertenece.`

```

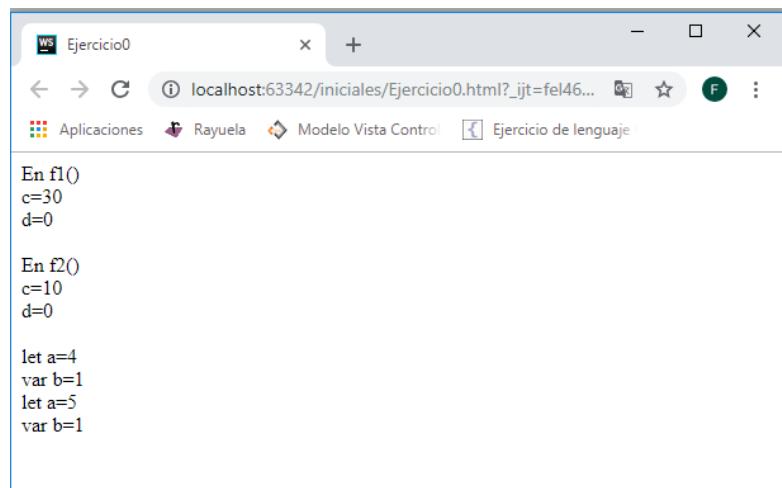
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Ejercicio0</title>
    <script type="text/javascript" src="Ejercicio0.js"></script>
</head>
<body>
    <script>
        var c = 10; // Variable LOCAL.
        d = 20; // Variable GLOBAL

        f1(c,d);

        document.write("<br>");
        f2();

        document.write("<br>");
        varlet();
    </script>
</body>
</html>

```



```

//Diferencia entre variable local y global
function f1(a,b) {
    var c=a+b; // La variable c es LOCAL
    d=0; // La variable d es GLOBAL
    document.write("En f1()<br/>");
    document.write("c="+c+"<br/>");
    document.write("d="+d+"<br/>");
}

function f2() {
    document.write("En f2()<br/>");
    document.write("c="+c+"<br/>");
    document.write("d="+d+"<br/>");
}

//Diferencia entre var y let
function varlet() {
    var a = 5;
    var b = 10;

    if (a === 5) {
        let a = 4; // El alcance es dentro del bloque if
        var b = 1; // El alcance es global

        document.write("let a="+a); // 4
        document.write("<br>var b="+b); // 1
    }

    document.write("<br>let a="+a); // 5
    document.write("<br>var b="+b); // 1
}

```

Léxico

- Otra posibilidad es declarar e inicializar la variable al mismo tiempo:

```
var variableEjemplo = "Contiene un texto";
```

- O incluso sin utilizar la palabra reservada var:

```
variableEjemplo = "Contiene un texto";
```

NOTA: en Javascript no hay un límite claro entre la declaración y el uso de variables, por ello se recomienda la declaración de las mismas al inicio del programa.

Léxico

Nombres de variables

- Javascript es sensible al uso de mayúsculas de manera que no es lo mismo definir una variable llamada “Contador” que otra llamada “contador”.
- Los nombres de las variables deben comenzar siempre por una letra o por el símbolo “_”. A continuación se puede incluir todos los caracteres y números que se consideren necesarios.
- No se utilizará el espacio en blanco.
- Otro aspecto a tener en cuenta a la hora de nombrar variables o funciones son las palabras reservadas.

Léxico

Nombres de variables

Variables en JavaScript

```
<HR>
<SCRIPT>
var Dato_1;
Dato_1 = "quince";
Dato_1 = 15;

var Dato_2 = 25;

Dato_3 = 35;

document.write("Total = " + (Dato_1 + Dato_2 + Dato_3));
</SCRIPT>
<HR>
```

Léxico

Palabras reservadas

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

Léxico

Alcance de variables

Hace referencia a los lugares del documento html o del programa Javascript en los que una variable es visible.

- **Variable local:** aquella que no puede utilizarse o ser manipulada fuera de la función en que se define.
- **Variable global:** aquella que se declara fuera de la función y está disponible en cualquier punto del programa.

Léxico

```
<!DOCTYPE html>
<html>
<head>
<title>Fichero 2.4</title>
<script LANGUAGE="Javascript1.2">
    var varGlobal = 100;
    function ejemploVariableLocal() {
        var varLocal = 10;
        varGlobal = 200; //Después de la función, la vale 200
    }
</script>
</head>
<body>
    <script LANGUAGE="Javascript1.2">
        document.write (varGlobal); // varGlobal vale 100
        ejemploVariableLocal(); // Se llama a la función
        document.write (varLocal); // No tiene valor fuera de la
                                // función al ser variable local
        document.write (varGlobal) // Ahora varGlobal vale 200
    </script>
</body>
</html>
```

Léxico

Alcance de variables

- Si se declara una variable dentro de una función, y esta tiene el mismo nombre que una global, la variable local tiene lo que se denomina **precedencia**.
- Esto sencillamente indica que las referencias a esa variable se asocian a la local y no a la global, y cambios en su contenido no afectan, por lo tanto, a la variable global.

Léxico

```
<html>
    <head>
        <title>Fichero 2.5</title>
        <script LANGUAGE="Javascript">
            var varEjemplo = 100;
            function ejemploAlcanceVariables() {
                var varEjemplo = 10;
                document.write ("La variable vale "+varEjemplo+"<BR/>");
            }
        </script>
    </head>
    <body>
        <script LANGUAGE="Javascript">
            ejemploAlcanceVariables();
            document.write ("La variable vale "+ varEjemplo);
        </script>
    </body>
</html>
```

Léxico

Tipos de Datos

Javascript contiene varios tipos básicos de datos:

- Valores numéricos.
- Valores booleanos.
- Cadenas de caracteres o Strings.

Números

- 1, 12.10, -0.578

booleanos

- true o false.

Cadenas o Strings

- “esto es una cadena”
- ‘Esto es otra cadena’

Léxico

Valores numéricos

Javascript usa dos tipos de valores numéricos:

- **Enteros**: positivos y negativos sin parte decimal. Que se pueden especificar:
 - en base 10 (17, 25...)
 - en base octal (010, 111...)
 - en base 16 (0x12, 0x05...).
- **Coma flotante**: estos número son los que tienen parte decimal y pueden representarse de dos formas:
 - Notación por punto: 5.3458.
 - Notación exponencial: -3.54e25

Léxico

Valores booleanos

Los tipos de datos lógicos o booleanos únicamente puede tener dos valores:

- **true** (verdadero)
- **false** (falso)

Las variables con valores booleanos se utilizan para comprobar que en un determinado punto del programa se cumple una condición.

Léxico

Cadenas de caracteres

Un *String* es un conjunto de cero o más caracteres encerrados entre simples o dobles comillas.

En un *String* es posible utilizar combinaciones de caracteres que definen símbolos especiales que por medio de letras o números no se podrían representar, como un salto de línea o un carácter de tabulación. Estas combinaciones de caracteres reciben el nombre de caracteres o secuencias de escape.

Léxico

\n	Salto de línea
\r	Retorno de carro
\t	Tabulación
\b	Carácter anterior
\\"	El símbolo \
\'	La comilla simple
\"	La comilla doble

Cuando se inserta una secuencia de escape el resultado no aparece por pantalla.

Léxico

Valores null

Un valor null no representa nada, ningún valor. Se suele utilizar para comprobar si una variable tiene asignado algún valor, ya que en caso contrario el contenido de la misma será null.

```
var variableEjemplo // Su valor es null  
alert ("El valor de la variable es: " + variableEjemplo);
```

```
variableEjemplo = 10 // Ahora su valor es 10  
alert ("El valor de la variable es: " + variableEjemplo);
```

Léxico

Conversión de tipos de datos

La conversión tiene lugar cuando Javascript convierte una cadena de caracteres a un número y viceversa, pero en determinadas ocasiones esta conversión automática no funciona.

Cuando un string representa un número, como por ejemplo "1234", la conversión no es un problema, pero cuando está compuesto por un conjunto de letras se produce un error de conversión.

```
// Error de conversión
valorEntero = 3 * "José Sánchez"

// No hay ningún problema
valorEntero = 3 + "1234"
```

Léxico

Para evitar errores de conversión durante la ejecución del programa, una buena práctica de programación es forzar voluntariamente el proceso de conversión en lugar de permitir al intérprete que lo realice. Para lograr esto se encuentran disponibles las funciones:

- `parseInt`
- `parseFloat`

Estas dos funciones se utilizan para extraer valores numéricos de cadenas de caracteres. La sintaxis de ambas es la siguiente:

```
parseInt (cadena [, base]);  
parseFloat (cadena);
```

Léxico

También hay otra forma de realizar las conversiones:

- Para convertir un número x a una cadena de caracteres hay que emplear `" " + x`. Por ejemplo:

`" " + 3` es equivalente a `"3"`

- Para convertir una cadena de caracteres x a un número hay que emplear `x - 0`. Por ejemplo:

`"3" - 0` es equivalente a `3.`

Léxico

Expresiones

Es una colección de variables (operandos) y operadores que devuelven un valor. En Javascript existen varios tipos:

- **Expresiones de asignación:** Permiten asignar un valor a una variable.
- **Expresiones aritméticas:** Permiten calcular nuevos valores a partir de variables ya existentes.
- **Expresiones con cadenas de caracteres:** se utilizan para crear nuevos Strings por concatenación de los ya existentes mediante el empleo del operador '+'.
- **Expresiones condicionales:** Se emplean para ejecutar una u otra sentencia de acuerdo con alguna condición.

Léxico

Operadores de asignación

Cuando se necesita cambiar o asignar un valor a una variable:

=	Permite asignar el valor de la expresión u operando derecho a la variable situada en el lado izquierdo del operador. Ejemplo: <code>a = 2 + 3 // a pasa a valer 5</code>
+=	Suma los operandos izquierdo y derecho y asigna el resultado al operando izquierdo. Ejemplo (a vale 5): <code>a += 3 // a pasa a valer 8</code>
-=	Resta el operando derecho al izquierdo y asigna el resultado al operando izquierdo. Ejemplo (a vale 5): <code>a -= 3 // a pasa a valer 2</code>
*=	Multiplica los operando s izquierdo y derecho y asigna el resultado al operando izquierdo. Ejemplo (a vale 5): <code>a *= 3 // a pasa a valer 15</code>
/=	Divide el operando izquierdo por el derecho y asigna el resultado al operando izquierdo. Ejemplo (a vale 10): <code>a /= 2 // a pasa a valer 5</code>
%=	Divide el operando izquierdo por el derecho y asigna el resto de la división al operando izquierdo (a vale 5): <code>a %= 3 // a pasa a valer 2.</code>

Léxico

Operadores aritméticos

Permiten hacer las operaciones matemáticas básicas:

OPERADOR	DESCRIPCIÓN
+	Se utiliza para suma de números y concatenación de strings. Ejemplos: <code>a = 3 + 2 // a pasa a valer 5</code> <code>cadena = "J." + " " + "Sánchez" // cadena es "J. Sánchez"</code>
-	Se utiliza para resta y negación (convertir un valor positivo en negativo y viceversa). <code>a = 3 - 2 // a pasa a valer 1</code> <code>b = -a // b pasa a valer -1</code>
*	Multiplicación: <code>a = 2 * 3 // a pasa a valer 6</code>
/	División: <code>b = 6 / 2 // b pasa a valer 3</code>
%	Cálculo del resto de una división: <code>a = 10 % 3 // a pasa a valer 1</code>
++	Incremento de una variable en una unidad: <code>a++ // a incrementar su valor en 1</code>
--	Decremento de una variable en una unidad: <code>a-- // a decrementar su valor en 1</code>

Léxico

Operadores lógicos y de relación

Permiten hacer operaciones lógicas:

OPERADOR	DESCRIPCIÓN
<code>==</code>	Expresa igualdad: <code>1 == 1</code> es true; <code>1 == 0</code> es false;
<code>!=</code>	Expresa desigualdad: <code>1 != 1</code> es false; <code>1 != 0</code> es true;
<code><</code>	Menor que: <code>1 < 5</code> es true; <code>5 < 1</code> es false;
<code>></code>	Mayor que: <code>1 > 5</code> es false; <code>5 > 1</code> es true;
<code><=</code>	Menor o igual que: <code>1 <= 5</code> es true; <code>5 <= 5</code> es true;
<code>>=</code>	Mayor o igual que: <code>1 >= 5</code> es false; <code>5 >= 5</code> es true;
<code>!</code>	Negación lógica: <code>!(1 == 1)</code> es false ya que <code>(1 == 1)</code> es true;
<code>&&</code>	AND lógica: <code>false && false</code> es false / <code>false && true</code> es false / <code>true && false</code> es false / <code>true && true</code> es true;
<code> </code>	OR lógica: <code>false false</code> es false / <code>false true</code> es true / <code>true false</code> es true / <code>true true</code> es true

Operador de igualdad

En la versión 1.2 o superior los operadores de igualdad y desigualdad se comportan de manera diferente que en versiones anteriores:

1. Nunca intentan convertir operadores de un tipo a otro.
2. Sólo comparan operandos del mismo tipo, de tal forma, que si los operandos no son del mismo tipo, son diferentes.

Léxico

```
<script>
    document.write ("3" == 3) //true
</script>

<script>
    document.write(("3"- 0) == 3) //true
</script>

<script LANGUAGE=" Javascript" >
    document.write ("3" == 3) // true
</script>

<script LANGUAGE="Javascript1.1">
    document.write ("3" == 3) // true
</script>

<script LANGUAGE="Javascript1.2">
    document.write ("3" == 3) // false
</script>

<script LANGUAGE = "Javascript1.2" >
    document.write (( "3"-0) == 3) // true
</script>
```

Léxico

Operador de igualdad

JavaScript tiene comparaciones estrictas y de conversión de tipos.

Una comparación estricta (por ejemplo, `==`) solo es verdadera si los operandos son del **mismo tipo** y los **contenidos coinciden**.

La comparación abstracta más comúnmente utilizada (por ejemplo, `==`) convierte los operandos al mismo tipo antes de hacer la comparación.

Para las comparaciones abstractas relacionales (p. Ej., `<=`), Los operandos primero se convierten en primitivos, y luego en el mismo tipo, antes de la comparación.

Léxico

Operador de igualdad

Especificaciones

ECMAScript 1st Edition (ECMA-262)	Standard	Definición inicial. Implementado en JavaScript 1.0
ECMAScript 3rd Edition (ECMA-262)	Standard	Agrega === y !== operadores. Implementado en JavaScript 1.3
ECMAScript 5.1 (ECMA-262)	Standard	Definido en varias secciones de la especificación: Operadores Relacionales , Operadores de Igualdad
ECMAScript 2015 (6th Edition, ECMA-262)	Standard	Definido en varias secciones de la especificación: Operadores Relacionales , Operadores de Igualdad
ECMAScript Latest Draft (ECMA-262)	Draft	Definido en varias secciones de la especificación: Operadores Relacionales , Operadores de Igualdad

Sentencias Javascript

Sentencias Javascript

Como ya tenemos una base de programación sólo se van a enumerar:

- Ejecución condicional
 - If anidados
- Sentencias de control de bucles
 - Sentencia for.
 - Sentencia While
 - Sentencia do while
 - Sentencia Break
 - Sentencia Continue
- Sentencias Switch

Funciones

Definiendo una función

Javascript se basa en lo que se conoce como **programación modular**, es una técnica utilizada habitualmente para facilitar la escritura y legibilidad de grandes programas.

Consiste en **subdividir** un programa en pequeñas secciones de código llamadas módulos de tal forma que cada sección realice alguna tarea que tenga entidad propia.

Funciones

Las reglas para la definición de funciones son similares a las de las variables. Otras reglas adicionales a tener en cuenta son:

- El nombre de la función debe ir precedido de la palabra reservada **function**.
- Las funciones deben ser únicas, no puede haber dos funciones con el mismo nombre.
- No se pueden utilizar palabras reservadas como nombres de funciones.

Funciones

La declaración de una función consta de la palabra reservada **function**, el *nombre de la función*, un par de paréntesis() y, a continuación, un conjunto de sentencias agrupadas por un par { }.

```
function nombre_función () {  
    .....  
    conjunto de sentencias  
    .....  
}
```

Funciones

Argumentos de una función

Una característica de las funciones es que se pueden declarar para que cuando se las llame acepten argumentos o parámetros.

```
function nombre_función (param1, param2, ... , parmaN) {  
    .....  
    conjunto de sentencias  
    .....  
}
```

La llamada a la función se realiza poniendo el nombre de la misma y la lista de parámetros entre paréntesis siempre.

Funciones

Devolución de un valor

Una de las características más útiles de las funciones es el de poder devolver un resultado por medio de la instrucción `return`.

```
<html>
  <head>
    <title>Fichero 4.4: Ejemplo de función para sumar números</title>
    <script LANGUAGE= "Javascript">
      function sumarNumeros (numerol, numero2) {
        return numerol + numero2
      }
    </script>
  </head>

  <body>
    <script LANGUAGE="Javascript">
      var resultado
      resultado = sumarNumeros (10, 20)
      alert ("El resultado es " + resultado)
    </script>
  </body>
</html>
```

Definición de función y Expresión con función

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

Esta función está disponible para todo el programa.

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

No es posible referenciarla antes de la expresión, porque realmente no se trata de una función declarada, sino del valor asignado a una variable.

Una diferencia más. En el segundo caso (usar una función en una expresión) es necesario terminar la expresión con un punto y coma (;), igual que cualquier otra expresión.

Definición de función y Expresión con función

```
//function declaration
// No produce error

saludar();

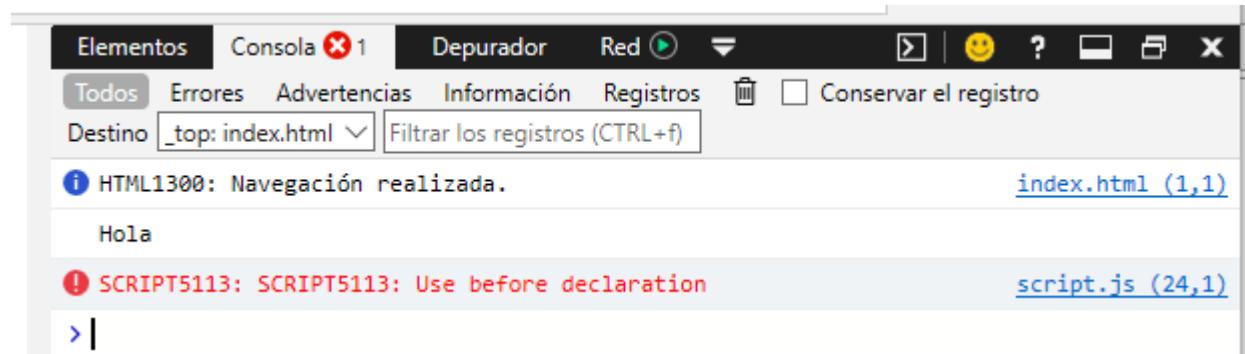
function saludar(){
    console.log('Hola');
}

//function expression
// Produce error si se llama antes de crearla

suma();

const suma= function(){
    console.log('Hola');
}
```

En el primer caso, la función se ejecuta correctamente, mientras en el segundo se produce un error, ya que se ha llamado antes de que la variable suma tenga valor.



Funciones

Sobrecarga

Algo muy común en lenguajes modernos es la sobrecarga de funciones y procedimientos. En javascript **NO** podemos hacerlo directamente.

Si hicieramos esto:

```
function Buscar(nombre){  
    ***  
}  
function Buscar(nombre, apellido){  
    ***  
}
```

El resultado sería que la segunda función sobreescribiría la primera y solo nos quedaríamos con esa. Así que ambas llamadas irían a la función que tiene dos parámetros.

Lo que nos salva en este escenario es el hecho que **todos los parámetros** en las funciones **son opcionales** en JavaScript. Lo malo es que depende de nosotros manejar si el usuario envió o no un parámetro *y manejar a la defensiva*.

Funciones

Sobrecarga

Ejemplo 1

```
function sumar (num1, num2){  
    return num1+num2  
}
```

```
Sumar (1)
```

```
NaN
```

En este caso tenemos la función Sumar que recibe dos parámetros, pero al invocarla solo le estamos pasando uno, lo que resulta que al tratar de sumar la variable este *undefined* y retorne la excepción *NaN* (Not a number).

Funciones

Sobrecarga

Ejemplo 2

```
function sumar (num1, num2){  
    if (num1 && num2)  
        return num1+num2;  
    if (num1)  
        return num1;  
    return 0;  
}
```

Sumar ()

0

Sumar (1)

1

Sumar (1,1)

2

En este segundo ejemplo lo que hacemos es verificar si existen ambos parámetros antes de hacer la suma (obviamente faltaría verificar el tipo de parámetro para manejar excepciones).

Funciones

Arrow Functions en ES6

<https://desarrolloweb.com/articulos/arrow-functions-es6.html>

```
let mifuncion = () => {
  //código de la función
}
```

La invocación de la función se realizaría como ya conoces.

```
mifuncion();
```

```
let saludo = (nombre, tratamiento) => {
  alert('Hola ' + tratamiento + ' ' + nombre)
}

//invocamos
saludo('Miguel', 'sr.');
```

Arrays

Declaración de un array

Un array es una estructura de datos que permite almacenar varios datos en una única variable. En Javascript se declara de la siguiente manera:

```
var nombre_array = new Array ( )
```

```
var nombre_array = new Array (longitud_del_array)
```

En Javascript la longitud es una referencia pudiendo superar esta sin problemas.

Arrays

Declaración por notación literal

En Javascript1.2 existe la posibilidad de crear un array mediante la notación literal. La sintaxis para la creación de un array mediante esta notación es la siguiente:

```
var name_array = [elemento0, elemento1 , . . . , elementoN]
```

```
var coche = ["opel", , "mercedes", "audi"]
```

De esta manera se cumple que:

Coche[0] es “opel”

Coche[1] es “undefined”

Coche[2] es “mercedes”

Coche[3] es “audi”

Arrays

Existen dos tipos diferentes de declaración:

```
var tabla= new Array();           // No utilizar en javaScript  
var tabla= [];                  // Si utilizar en javaScript
```

```
var tabla= new Array(40, 100, 1, 5, 25, 10) // No utilizar en javaScript  
var tabla= [40, 100, 1, 5, 25, 10];          // Si utilizar en javaScript
```

Consultar

http://www.w3schools.com/js/js_arrays.asp

Arrays

Manipulación de un array

Para asignar un valor a un elemento de una variable de tipo array se emplea la siguiente expresión:

```
nombre_array [posición] = valor;
```

y para asignar el valor de uno de los elementos del array a una variable del mismo tipo que los elementos:

```
variable = nombre_array [posición];
```

Es muy importante no olvidar que en Javascript todo elemento de un array que **NO** haya sido **inicializado** con un valor tiene por defecto el valor **null**.

Arrays

Longitud un array

Una de las características de los arrays en Javascript es la existencia de una propiedad o atributo denominado **length** que contiene la longitud que el array va teniendo. El acceso a esta propiedad se realiza de la siguiente forma:

nombre_array.length

Siempre devuelve un valor de tipo entero.

Arrays

Array Asociativo

Este array asocia a cada elemento un texto que concuerda con la posición que ocupa en el mismo:

```
var alineacion = new Array();      // No se utiliza  
var alineacion = []                // mejor
```

```
alineacion["portero"] = "Casillas";  
alineacion["central1"] = "Puyol"  
alineacion["central2"] = "Sergio Ramos";  
...  
alineacion["delantero"] = "Fernando Torres";
```

La complejidad existe a la hora de recorrerlos.

Arrays

Recorrer Array Asociativo

Para recorrer un array asociativo y conservar los índices se puede realizar con la siguiente implementación del for:

```
for (var indice in alineacion) {  
    document.write(indice+':'+alineacion[indice]+'<BR/>');  
}
```

Arrays

Arrays bidimensionales

En Javascript no existe una forma directa de declararlo un array bidimensional. Para ello, hay que partir de la declaración de un array unidimensional y recurrir a un bucle for que vaya insertando en cada elemento del array un nuevo array unidimensional.

El siguiente código muestra cómo crear un array bidimensional de 10 x 10 elementos:

Simplemente hace que cada elemento dentro de la matriz sea una matriz.

```
var x = new Array(10);

for (var i = 0; i < x.length; i++) {
    x[i] = new Array(3);
}
```

Arrays

El acceso a los elementos de un array bidimensional es exactamente igual al método empleado para acceder a los unidimensionales, pero utilizando un índice adicional para manipular la segunda dimensión:

```
variable_array_bidimensional [indice1] [indice2] = valor;
```

y para la asignación:

```
variable = variable_array_bidimensional [indice1] [indice2];
```

Arrays bidimensionales

vamos a crear un array de dos dimensiones donde tendremos por un lado ciudades y por el otro la temperatura media que hace en cada una durante de los meses de invierno.

```
var temperaturas_medias_ciudad0 = new Array(3)
temperaturas_medias_ciudad0[0] = 12
temperaturas_medias_ciudad0[1] = 10
temperaturas_medias_ciudad0[2] = 11

var temperaturas_medias_ciudad1 = new Array (3)
temperaturas_medias_ciudad1[0] = 5
temperaturas_medias_ciudad1[1] = 0
temperaturas_medias_ciudad1[2] = 2

var temperaturas_medias_ciudad2 = new Array (3)
temperaturas_medias_ciudad2[0] = 10
temperaturas_medias_ciudad2[1] = 8
temperaturas_medias_ciudad2[2] = 10
```

Arrays bidimensionales

Con las anteriores líneas hemos creado tres arrays de 1 dimensión y tres elementos, como los que ya conocíamos.

Ahora crearemos un nuevo array de tres elementos e introduciremos dentro de cada una de sus casillas los arrays creados anteriormente, con lo que tendremos un array de arrays, es decir, un array de 2 dimensiones.

```
var temperaturas_cuidades = new Array (3)
temperaturas_cuidades[0] = temperaturas_medias_ciudad0
temperaturas_cuidades[1] = temperaturas_medias_ciudad1
temperaturas_cuidades[2] = temperaturas_medias_ciudad2
```

Arrays bidimensionales

Para recorrer el array bidimensional, utilizaremos un bucle anidado.

```
for (i=0;i<temperaturas_cuidades.length;i++){
    document.write("<tr>")
    document.write("<td><b>Ciudad " + i + "</b></td>")
    for (j=0;j<temperaturas_cuidades[i].length;j++){
        document.write("<td>" + temperaturas_cuidades[i][j] + "</td>")
    }
    document.write("</tr>")
}
```

Eventos

- ▶ Los eventos son señales que se generan como consecuencia de la interacción del usuario: movimientos del ratón, pulsaciones sobre botones, ...
- ▶ Ciertos elementos HTML son sensibles a determinados eventos
 - ▶ Un manejador de evento es un trozo de código JavaScript que actúa en respuesta a un evento generado sobre un elemento HTML
 - ▶ Los manejadores de eventos se ejecutan cuando se genera el evento y no en el momento de cargar la página HTML
 - ▶ Esta forma de incluir código JavaScript permite modificar un documento HTML en función de la interacción del usuario

Eventos

Podemos distinguir entre distintos tipos de eventos:

- ▶ Eventos de carga.
- ▶ Eventos del ratón.
- ▶ Eventos del foco.
- ▶ Eventos de teclado.
- ▶ Eventos de formulario.

Eventos HTML Común

Evento	Descripción
onchange	Un elemento HTML se ha cambiado
onclick	El usuario hace clic en un elemento HTML
onmouseover	El usuario mueve el ratón sobre un elemento HTML
onmouseout	El usuario mueve el ratón fuera de un elemento HTML
onkeydown	El usuario pulsa una tecla del teclado
onload	El navegador ha terminado de cargar la página

Eventos de carga

➤ Evento load

- Ocurre cuando se finaliza la carga de una página o de una imagen
- Manejador: **onLoad**
- Se aplica a las etiquetas **<BODY>** e ****

Evento load

```
<HTML>
<HEAD>
    <TITLE>JavaScript: Evento 'load'</TITLE>
</HEAD>

<BODY onLoad="alert('Evento load en BODY')">

<H1>JavaScript: Evento 'load'</H1>

<IMG src="simpsons.jpg"
      onLoad="alert('Evento load en IMG')">

</BODY>
</HTML>
```

Eventos de carga

➤ Evento unload

- ❑ Ocurre cuando se sale de una página a través de un enlace
- ❑ Manejador: **onUnload**
- ❑ Se aplica a la etiqueta **<BODY>**

Evento unload

```
<HTML>
<HEAD>
  <TITLE>JavaScript: Evento 'unload'</TITLE>
</HEAD>

<BODY onUnload="alert('Evento unload')">

<H1>JavaScript: Evento 'unload'</H1>

<A href="http://www.cs.us.es">CCIA</A>

</BODY>
</HTML>
```

Eventos de carga

➤ Evento abort

- ❑ Ocurre cuando se aborta la carga de una imagen
- ❑ Manejador: **onAbort**
- ❑ Se aplica a la etiqueta <**IMG**>

Evento abort

```
<HTML>
<HEAD>
    <TITLE>JavaScript: Evento 'abort'</TITLE>
</HEAD>

<BODY>

<H1>JavaScript: Evento 'abort'</H1>

<IMG src="simpsons.jpg"
      onAbort="alert('Evento abort')">

</BODY>
</HTML>
```

Eventos de carga

➤ Evento error

- ❑ Ocurre cuando la carga de una imagen causa un error
- ❑ Manejador: **onError**
- ❑ Se aplica a la etiqueta ****

Evento error

```
<HTML>
<HEAD>
    <TITLE>JavaScript: Evento 'error'</TITLE>
</HEAD>

<BODY>

<H1>JavaScript: Evento 'error'</H1>

<IMG src="springfield.jpg"
     onError="alert('Evento error')">

</BODY>
</HTML>
```

Eventos del ratón

➤ Evento mouseover

- ❑ Ocurre cuando el ratón se coloca dentro de un elemento
- ❑ Manejador: **onMouseover**

➤ Evento mouseout

- ❑ Ocurre cuando el ratón se saca fuera de un elemento
- ❑ Manejador: **onMouseout**

➤ Evento mousemove

- ❑ Ocurre cuando el ratón se mueve encima de un elemento
- ❑ Manejador: **onMousemove**

➤ Se aplican a la mayoría de las etiquetas HTML

Eventos del ratón

➤ Evento mousedown

- ❑ Ocurre cuando se presiona un botón del ratón dentro de un elemento
- ❑ Manejador: **onMousedown**

➤ Evento mouseup

- ❑ Ocurre cuando se suelta un botón del ratón dentro de un elemento
- ❑ Manejador: **onMouseup**

➤ Se aplican a la mayoría de las etiquetas HTML

Eventos del ratón

➤ Evento click

- ❑ Ocurre cuando se pulsa un botón del ratón dentro de un elemento
- ❑ Manejador: **onClick**

➤ Evento dblclick

- ❑ Ocurre cuando se pulsa dos veces un botón del ratón dentro de un elemento
- ❑ Manejador: **onDblclick**

➤ Se aplican a la mayoría de las etiquetas HTML

Eventos del ratón

Eventos de ratón

```
<IMG src="simpsons.jpg"
      onMouseover="alert('Evento mouseover')"
      onClick="alert('Evento click')"> </TD>

<IMG src="simpsons.jpg"
      onMouseout="alert('Evento mouseout')"
      onMouseup="alert('Evento mouseup')"> </TD>

<IMG src="simpsons.jpg"
      onMousemove="alert('Evento mousemove')"
      onDblclick="alert('Evento dblclick')"> </TD>

<IMG src="simpsons.jpg"
      onMousedown="alert('Evento mousedown')"> </TD>
```

Eventos de foco

➤ Evento focus

- ❑ Ocurre cuando un elemento gana el “foco”
- ❑ Manejador: **onFocus**

➤ Evento blur

- ❑ Ocurre cuando un elemento pierde el “foco”
- ❑ Manejador: **onBlur**

➤ Se aplican a los controles activos de un formulario y a la etiqueta **<A href ...>**

Eventos de foco

```
<FORM name="ejemplo">
    ¿Cual es tu nombre? <BR>
    <INPUT type="text" name="nombre"
        onFocus="alert('Evento focus')"
        onBlur="alert('Evento blur')"> <BR>
</FORM>

<A href="http://www.cs.us.es">CCIA</A>
```

Eventos de teclado

➤ Evento keydown

- ❑ Ocurre cuando se pulsa una tecla
- ❑ Manejador: **onkeydown**

➤ Evento keypress

- ❑ Ocurre cuando se presiona una tecla
- ❑ Manejador: **onkeypress**

➤ Evento keyup

- ❑ Ocurre cuando se suelta una tecla
- ❑ Manejador: **onkeyup**

➤ Se aplican a los controles activos de un **formulario**

Eventos de teclado

Eventos de teclado

```
<FORM>
  Evento keydown:
  <INPUT type="text"
         onkeydown="alert('Evento keydown')"> <BR>
  Evento keypress:
  <INPUT type="text"
         onkeypress="alert('Evento keypress')"> <BR>
  Evento keyup:
  <INPUT type="text"
         onkeyup="alert('Evento keyup')">
</FORM>
```

Eventos de formulario

➤ Evento reset

- ❑ Ocurre se reinicia un formulario
- ❑ Manejador: **onReset**
- ❑ Se aplica a la etiqueta <FORM>

➤ Evento submit

- ❑ Ocurre se envía un formulario
- ❑ Manejador: **onSubmit**
- ❑ Se aplica a la etiqueta <FORM>

➤ Evento change

- ❑ Ocurre cuando se cambia el valor de un elemento
- ❑ Manejador: **onChange**
- ❑ Se aplica a las etiquetas <INPUT type="text">, <SELECT> y <TEXTAREA>

Eventos de formulario

Eventos de formulario

```
<FORM name="ejemplo"
      onSubmit="alert('Evento submit')"
      onReset="alert('Evento reset')">
  ¿Cual es tu nombre? <BR>
  <INPUT type="text" name="nombre"
         onChange="alert('Evento change')">
  <INPUT type="submit" value="Enviar">
  <INPUT type="reset" value="Reset"> <BR>

  Aficiones: <BR>
  <SELECT align="top" name="aficiones" size="2"
         onChange="alert('Evento change')">
    <OPTION value="v1">Música</OPTION>
    <OPTION value="v2">Cine</OPTION>
    <OPTION value="v3">Lectura</OPTION>
    <OPTION value="v4">Fotografía</OPTION>
  </SELECT> <BR>

  ¿Algún comentario? <BR>
  <TEXTAREA onChange="alert('Evento change')">
  </TEXTAREA>
</FORM>
```

Objetos y Métodos ejemplos

Objetos

Cuando definimos un objeto, en realidad estamos construyendo una serie de datos y métodos, o funciones, que van a regir la “vida” de dicho objeto.

Objetos y Métodos ejemplos

Objetos

Pero a la hora de crear objetos reales, basados en dicha definición, lo que estamos realmente creando y manejando son instancias del objeto.

Cuando se define una instancia de un objeto hay que asignarle un nombre (nombre_objeto) que nos permita diferenciarle del resto de instancias.

Objetos y Métodos ejemplos

Objetos

La sintaxis para acceder a las propiedades y métodos de una instancia de un objeto es:

- Propiedades: nombre_objeto.propiedad
- Métodos: nombre_objeto.método(argumentos)

Objetos y Métodos

Definición de clases

Como bien se sabe en Programación Orientada a Objetos la clase es la definición y el objeto es la instancia.

En Javascript las clases se crean utilizando funciones. De hecho si queremos crear una clases lo único que hay que hacer es crear una función que denominaremos constructor.

```
function rectangulo(lado1, lado2) {  
    this.lado1 = lado1;  
    this.lado2 = lado2;  
}
```

Y para crear la instancia:

```
var rectangulo_1 = new rectangulo(3, 8);
```

Objetos y Métodos

Creación y llamada de métodos

Para definir los métodos se asigna el nombre de una función ya creada al nombre de un método del constructor del tipo de objeto deseado.

```
//función ya creada
function get_area() {
    return this.lado1*this.lado2;
}

function rectangulo(lado1, lado2) {
    this.lado1 = lado1;
    this.lado2 = lado2;
    this.calcular_area = get_area;
}
```

Y para llamar al método:

```
rectangulo_1.calcular_area();
```

Objetos y Métodos

Nuevas propiedades y métodos

Javascript tiene la posibilidad de **añadir** propiedades y métodos a los objetos iniciales ya existentes. Para ello se utiliza la propiedad **prototype**, su sintaxis es:

```
Constructor.prototype.propiedad = valor;
```

```
//añadimos la propiedad al objeto
rectangulo.prototype.perimetro = 0;

function get_perimetro(){
    this.perimetro=2*this.lado1+2*this.lado2;
}
```

Y para incluir el método:

```
rectangulo.prototype.calcular_perimetro = get_perimetro;
```

Objetos y Métodos

Borrar propiedades y métodos

Igual que se tiene la posibilidad de añadir propiedades y métodos a los objetos ya existentes, Javascript permite eliminarlas. Para ello se utiliza la palabra reservada delete, su sintaxis es:

```
delete nombre_objeto.nombre_propiedad;  
delete nombre_objeto.nombre_metodo;
```

```
//eliminamos la propiedad al objeto  
delete rectangulo.perimetro;
```

```
//eliminamos el método al objeto  
delete rectangulo.calcular_perimetro;
```

Herencia

prototype permite herencia entre objetos.

```
function objeto(dato1)
{
    this.dato1 = dato1;          //this hace las variables públicas,
                                //su ausencia las hace privadas
    this.metodo1 = function(){ ... };

}
var variable = new objeto();
function objeto2(dato1, dato2)
{
    objeto.call(this,dato1); //Pasa la variable a constructor de la clase base
    this.dato2 = dato2;
    this.metodo2 = function() { ... };
}
objeto2.prototype = new objeto();
```

Si el constructor de la clase padre tiene parámetros, podemos llamarlo desde la clase hija usando la función **call()** que tienen definidas las clases por defecto.

la palabra reservada **this** hace referencia al objeto indicado en la llamada a **call()**.

Herencia

Hemos visto la importancia del atributo **prototype** en la herencia de javascript. Debemos indicar que podemos acceder al atributo **prototype** en todas clases, incluso en las clases del sistema, como String o Date. El siguiente código añade un nuevo método `capitalize()` a la clase de sistema String:

```
String.prototype.capitalize = function() {  
    return this.charAt(0).toUpperCase() + this.substr(1).toLowerCase();  
    alert("hola".capitalize())  
}
```