Tema 4

Operadores del lenguaje y bucles

MEDAC
Instituto Oficial de Formación Profesional

El lenguaje de programación de clientes

Aprendiendo Javascript

- Interactividad
- Operadores
- Condicionales
- Bucles
- Funciones

MEDAC

La consola

- La consola nos va a servir para recibir avisos y errores que se pueden producir al ejecutar nuestro código Javascript. Si al intentar ejecutar nuestro código Javascript en el navegador no sucede nada, lo primero que hay que hacer es mirar si en la consola tenemos algún aviso de fallo o error.

- Javascript no mira primero todo el código en busca de errores y los muestra antes de ejecutarlo (hace como JAVA o C++). Por tanto, el navegador va a ir ejecutando el código hasta que encuentre un error (o acabe el programa). Al encontrar un error, lo indica y se para.

MEDAC

Mostrar valores por consola (*)

Para mostrar valores por consola podemos usar el objeto console. Concretamente las siguientes herramientas:

- console.log("mensaje"): muestra un mensaje para guiarnos mientras desarrollamos.
- console.info("mensaje"): muestra un mensaje para dar información a compañeros que estén desarrollando con nosotros.
- console.warn("mensaje"): se usa para mostrar mensajes de aviso de la propia aplicación.
- console.error("mensaje"): se usa para mostrar mensajes de error de la propia aplicación.

MEDAC

Limpieza y orden en la consola (*)

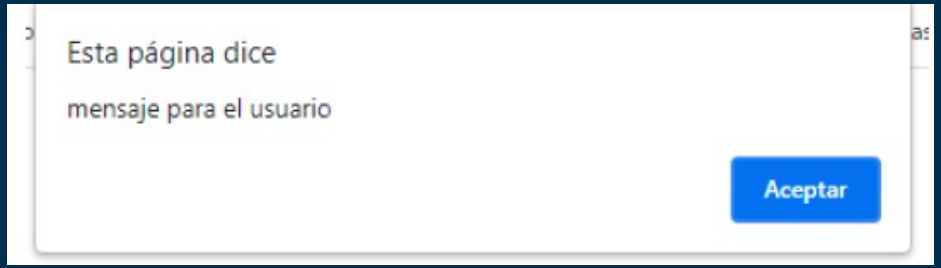
A poco que nuestra aplicación se vuelva medianamente compleja la consola puede llenarse de avisos, errores y mensajes que pongamos. Para mantener un mínimo de orden, podemos usar las consiguientes herramientas:

- console.clear(): elimina todos los mensajes que hay en la consola hasta ese momento. Tras usarse, se muestra un mensaje indicando que la consola se ha limpiado(en español o en inglés dependiendo del navegador).
- console.groupCollapsed("nombre") grupo de mensajes console.groupEnd():Este bloque de herramientas sirve para definir grupos de mensajes en la consola. Estos grupos aparecerán colapsados y con el nombre indicado en groupCollapsed. Debe ser el usuario el que los expanda para ver los mensajes.

Ventanas modales

Javascript tiene predefinidas algunas herramientas que van a permitir interactuar con el usuario a través de ventanas modales
Aunque son herramientas cada vez más en desuso, nos van a ser muy útiles en este momento de aprendizaje para interaccionar con nuestros programas

Ventana de alerta



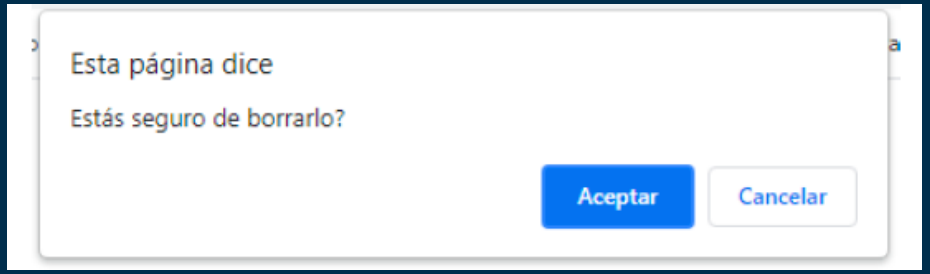
alert (“mensaje”);

Crea una ventana modal de alerta mostrando el mensaje indicado. El programa solo puede continuar cuando el usuario cierre esa ventana de aviso.

MEDAC

Ventanas modales (*)

Ventana de confirmación



confirm (“pregunta para el usuario”);

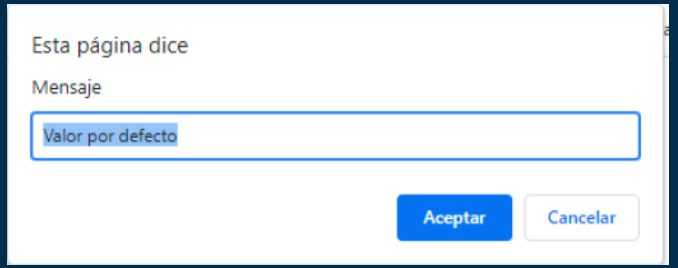
Muestra un aviso de confirmación en el que el usuario debe indicar si está de acuerdo o no pulsando el botón correspondiente (Aceptar o Cancelar). Devuelve un valor booleano (true o false) dependiendo del botón pulsado por el usuario: Aceptar = true, Cancelar = false

MEDAC

Pedir valores al usuario

Javascript permite que se le soliciten valores al usuario a través de una ventana modal usando la herramienta prompt.

promt (“Mensaje”, “Valor por defecto”);



Al usar esa herramienta, se mostrará por pantalla una ventana modal con un mensaje y con un campo de texto en el que el usuario podrá escribir. El valor por defecto es opcional

MEDAC

Pedir valores al usuario (*)

prompt siempre devuelve una cadena de caracteres. Cuando necesitemos pasar a número lo que venga del prompt, vamos a tener que usar las funciones parseInt (transforma a número entero) ó parseFloat (transforma a número decimal). Estas funciones transforman a número la cadena que se le ponga en los paréntesis

```
//pedimos la edad y la pasamos a numero antes de guardarla
var edad = parseInt(prompt("Introduce tu edad:",""));
//Operamos al mostrar
console.log("Dentro de 10 años tendrás: "+(edad+10));
```

MEDAC

document.write () (*)

Otra instrucción que podemos usar para mostrar información por pantalla es document.write

En este caso, la información se mostraría en la misma página, y no en un pop-up. Vemos como debemos escribirlo:

```
document.write("Hello World!");
```

Importante: Si usamos document.write () después de que se cargue un documento, borrará todo el contenido existente de HTML. Además este método no se puede usar en XHTML ni en XML

MEDAC

Operadores

- Para realizar operaciones entre los diferentes datos almacenados en nuestras variables

Operadores Aritméticos

Operador	Descripción
+	Operador suma
-	Comprador resta
*	Operador de producto
**	Operador de potencia
/	Operador división
%	Operador módulo.
++	Operador de incremento
--	Operador de decremento

MEDAC

Operadores de asignación especiales

- Para acortar operaciones matemáticas sencillas y volcar el resultado en una variable

Operador	Equivalencia
variable **+=** valor	variable = variable + valor
variable **-=** valor	variable = variable - valor
variable ***=** valor	variable = variable * valor
variable **/=** valor	variable = variable / valor
variable **%=** valor	variable = variable % valor

Operador typeof

No todos los operadores que poseen los lenguajes de programación tienen por qué ser símbolos. Algunos se escriben como palabras. Un ejemplo es el operador typeof. Este operador es un operador unario, es decir, actúa sobre un solo valor. Y lo que devuelve es el tipo de ese valor:

```
typeof 7.5 //dará como resultado la palabra "number"
typeof "J" //dará como resultado la palabra "string"
typeof Infinity //dará como resultado "number"
```

MEDAC

Ejercicios básicos (*)

MEDAC

Métodos útiles para usar con números (*)

Método	Descripción	Uso

Number.isInteger()	Verifica si un número es entero	console.log(Number.isInteger(10)); // true
		console.log(Number.isInteger(10.5)); // false

parseFloat()	Convierte una cadena a un número con decimales (flotante)	console.log(parseFloat("10.5")); // 10.5
parseInt()	Convierte una cadena a un número entero.	console.log(parseInt("10.5")); // 10
toFixed(decimales)	Redondea un número a un número fijo de decimales y lo convierte en un string.	let num = 3.14159; console.log(num.toFixed(2)); // "3.14"
toPrecision(dígitos)	Formatea el número a una longitud específica (total de dígitos) y devuelve una cadena.	let num = 1.23456; console.log(num.toPrecision(3)); // "1.23"
toString (base)	Convierte el número en una cadena en la base especificada.	let num = 255; console.log(num.toString(16)); // "ff" (hexadecimal)

Se puede usar también Math con sus métodos correspondientes

MEDAC

Operadores de comparación

Permiten realizar comparaciones entre diferentes expresiones o variables. Se aplican a expresiones numéricas y se obtienen valores booleanos

Operador	Descripción
==	Operador de igualdad
===	Igualdad de valor y de tipo
!=	Operador distinto
!==	Operador distinto valor o tipo
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Tabla 3. Operadores de comparación en JavaScript

Nota: NaN es el único número que no es igual a sí mismo. Usar isNaN(n)
null y undefined son iguales

MEDAC

Operadores lógicos

Permiten operar sobre valores booleanos y dan como resultado otro valor booleano

Operador	Descripción
&&	Y lógico
||	O lógico
!	Negación lógica

Tabla 5. Operadores lógicos en JavaScript

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

NOT	
TRUE	FALSE
FALSE	TRUE

Tabla 5. Tablas de verdad de los operadores AND, OR y NOT

MEDAC

Operadores a nivel de bits

Permiten realizar cualquier operación con valores que puedan ser representados con 32 bits

Operador	Descripción	Ejemplo
&	'Y' lógico a nivel de bit	2 & 0 equivalente a 10 & 00 = 00
|	'O' lógico a nivel de bit	2 | 0 equivalente a 10 | 00 = 10
~	Negación	~ 2 equivalente a ~10 = 01

Tabla 6. Operadores lógicos a nivel de bits en JavaScript

~ ALT + 126 (Windows) --- Option + ñ (MAC)

MEDAC

Condicionales - Estructura IF

IF Simple

Se evalúa la condición indicada en la cabecera de la estructura. Si la condición se cumple, se ejecuta la parte del código indicada, en caso contrario, esa parte del código es ignorada y no se ejecuta.

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

MEDAC

Condicionales - Estructura IF

IF.....ELSE

En esta forma definimos dos caminos. El primero se tomará si se cumple la condición (y se ignorará el segundo) y el segundo se tomará si la condición no se cumple (y se ignorará el primero).

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

Condicionales - Estructura IF

IF.....ELSEIF.....ELSE

Con esta forma vamos a crear tantos caminos como nosotros queramos y cada uno de esos caminos va a tener su propia condición para entrar en él. Si no se cumple la condición de los caminos definidos, se ejecutará el código que hay en la parte ELSE.

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is
true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is
false
}
```

Condicionales - Estructura switch (*)

Esta estructura es una manera rápida y elegante de crear un bloque para tomar decisiones en función del valor que tome una variable. Es aconsejable utilizar la sentencia switch cuando el nivel de anidamiento en sentencias condicionales simples es muy elevado.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

MEDAC

Bucles

Los bucles permiten ejecutar un bloque de sentencias en un número determinado de repeticiones. En todo lenguaje de programación esto es una tarea común. Dependiendo del lenguaje de programación, existen diferentes tipos de bucles

Bucle FOR

El bucle FOR se utiliza para repetir instrucciones un determinado número de veces. De entre todos los bucles, el FOR se suele utilizar cuando sabemos seguro el número de veces que queremos que se ejecute la sentencia. La sintaxis del bucle se muestra a continuación:

```
for (i=valor inicial;condición;actualización) {
   sentencias a ejecutar en cada vuelta
}
```

```
var i;
for (i=0;i<=10;i++) {
  console.log("Ya entiendo los bucles");
}
```

```
for (i=343;i>=10;i--){
 console.log(i);
}
```

MEDAC

Bucle WHILE

Este bucle se utiliza cuando queremos repetir la ejecución de unas sentencias un número indefinido de veces (no sabemos las vueltas que tenemos que dar). El bucle siempre va a estar dando vueltas mientras que se cumpla una condición. Hay que tener mucho cuidado con este bucle: Siempre tenemos que estar seguro de que la condición del bucle va a dejar de cumplirse en algún momento, si no, lo que metamos dentro del bucle while se estará repitiendo infinitas veces y así conseguiremos dejar colgado el navegador.

```
while (condición){
    sentencias a ejecutar
}
```

```
while (i < 10) {
  text += "El número es " + i;
  i++;
}
```

MEDAC

Bucle DO-WHILE

Es exactamente igual que el bucle WHILE con la diferencia de que sabemos seguro que el bucle por lo menos se ejecutará una vez (recordamos que en el WHILE, si no se cumple la condición al principio, no entramos en el bucle).

```
do {
    sentencias del bucle
} while (condición)
```

```
do {
  text += "El número es " + i;
  i++;
}
while (i < 10);
```

MEDAC

La sentencia BREAK

La sentencian break se puede usar dentro de muchas de las estructuras y bucles que hemos visto en este tema. Sin embargo, si usamos esta sentencia dentro de un bucle, break lo que se consigue es parar forzosamente las vueltas de ese bucle. Esto significa que se sale de él y deja todo como está en esa vuelta (valores de i, condiciones...) queden las vueltas que queden por dar.
Como es lógico, Al salir del bucle, el flujo de ejecución continua después del bucle.

```
for (i=1;i<=1000;i++){
    console.log(i);
    if (i==25){
       break;
    }//if
}
```

MEDAC

La sentencia BREAK

Importante: El uso de la sentencia break no es una buena práctica de programación debido al abuso que se realiza de ella. En la mayoría de los casos, se puede evitar su uso rompiendo la condición del bucle, por lo que, si se usa para salir de un bucle, es señal de que el programador que lo hace no tiene unas buenas bases del funcionamiento de los bucles en programación.

MEDAC

Continue

La sentencia continue se usa en bucles y sirve para volver al principio del bucle en cualquier momento, sin ejecutar las líneas que haya por debajo de la palabra continue (es decir, se usa para 'saltarse' código en las vueltas de un bucle).

```
var i, inc;

while (i<7){
  inc = prompt("sumo uno a la variable? (si/no)","");
   if (inc == "no"){
      continue; //me salto todo desde desde este punto
                //hasta la llave de cierre del bucle
   }
   i++;
   console.log("La variable se ha incrementado");
}
```

MEDAC

Bucles anidados (*)

La anidación de bucles es algo que se usa normalmente en programación y que permite hacer determinadas acciones más complejas

```
for (i=0;i<=3;i++){
    for (j=0;j<=9;j++) {
        console.log(i + "-" + j)
    }
}
```

MEDAC

Ejercicios básicos
Condicionales y bucles (*)

MEDAC

Funciones

- Cuando se desarrolla una aplicación, ya sea en un entorno web o no, casi siempre es una buena práctica estructurar el código por módulos. De esta manera, garantizamos que se pueda reutilizar el código a lo largo de nuestra solución, evitando tener código duplicado. Por la forma de programar que tenemos los seres humanos, estamos acostumbrados a repetir una y otra vez las mismas instrucciones para solucionar un mismo problema. En este sentido, es fácil observar que, si las instrucciones se repiten una y otra vez, crearemos redundancia y complicaremos, en gran medida, el código.
- El uso de las funciones nos permite estructurar el código de una manera sencilla y evitar, así, duplicar instrucciones. JavaScript, al igual que otros lenguajes de programación, permite la declaración de funciones a lo largo del código.

Funciones. Sintaxis

- Una función en JavaScript requiere de la utilización de la palabra reservada function.
- A continuación, se incluye un identificador con el nombre de la función. Nótese que las reglas para crear este identificador son exactamente iguales que las reglas para definir los identificadores de las variables.
- Tras el nombre de la función, se incluye la lista de argumentos entre paréntesis. Una función puede tener 'n' argumentos, siendo n mayor o igual que cero. Los argumentos de una función consisten en una lista de valores necesarios para que esta pueda ejecutar el bloque de código correspondiente.
- Posteriormente tenemos el bloque de la función definido entre llaves {} donde se realizarán las instrucciones correspondientes.
- Finalmente, la función puede devolver opcionalmente algún valor, esto lo hace con la palabra reservada return

Funciones. Sintaxis (*)

```
function nombreFuncion(parámetro1, parámetro2) {
    // Bloque de código que se ejecuta cuando se llama a la función
    return resultado; // opcional
}
```

```
function exampleFunction(a, b) {
 return a * b;  //La función devuelve el producto de los parámetros
}
```

- Se dice que el ámbito de visibilidad de una variable dentro de una función es el cuerpo exacto de la función. Es decir, fuera de una función esa variable no va a existir. Por ello es posible utilizar el mismo nombre de variables en otras funciones o en el código principal de nuestra solución.

MEDAC

Ejercicios básicos
Funciones (*)

MEDAC

MEDAC

Instituto Oficial de Formación Profesional