INTRODUCCIÓN A JAVASCRIPT

Javascript es un lenguaje que puede ser utilizado por profesionales y para quienes se inician en el desarrollo y diseño de sitios web. No requiere de compilación ya que el lenguaje funciona del lado del cliente, los navegadores son los encargados de interpretar estos códigos.

Muchos confunden el Javascript con el Java pero ambos lenguajes son diferentes y tienes sus características singulares. Javascript tiene la ventaja de ser incorporado en cualquier página web, puede ser ejecutado sin la necesidad de instalar otro programa para ser visualizado.

Java por su parte tiene como principal característica ser un lenguaje independiente de la plataforma. Se puede crear todo tipo de programa que puede ser ejecutado en cualquier ordenador del mercado: Linux, Windows, Apple, etc. Debido a sus características también es muy utilizado para internet.

1.1.- ¿QUÉ ES JAVASCRIPT?

Javascript es un lenguaje con muchas posibilidades, utilizado para crear pequeños programas que luego son insertados en una página web y en programas más grandes, orientados a objetos mucho más complejos. Con Javascript podemos crear diferentes efectos e interactuar con nuestros usuarios.

Este lenguaje posee varias características, entre ellas podemos mencionar que es un lenguaje basado en acciones que posee menos restricciones. Además, es un lenguaje que utiliza Windows y sistemas X-Windows, gran parte de la programación en este lenguaje está centrada en describir eventos, escribir funciones que respondan a movimientos del mouse, aperturas, utilización de teclas, cargas de páginas entre otros.

Es necesario resaltar que hay dos tipos de JavaScript: por un lado está el que se ejecuta en el cliente, este es el Javascript propiamente dicho, aunque técnicamente se denomina **Navigator JavaScript**. Pero también existe un Javascript que se ejecuta en el servidor, es más reciente y se denomina **LiveWire Javascript** (gracias a las alternativas como **Node.js**, Jaxer o RingoJS, entre otras).

1.2.- ¿CÓMO NACE JAVASCRIPT?

Javascript nació con la necesidad de permitir a los autores de sitio web crear páginas que permitan intercambiar con los usuarios, ya que se necesitaba crear webs de mayor complejidad. El HTML solo permitía crear páginas estáticas donde se podía mostrar textos con estilos, pero se necesitaba interactuar con los usuarios.

En los años de 1990, Netscape creo Livescript; las primeras versiones de este lenguaje fueron principalmente dedicadas a pequeños grupos de diseñadores Web que no necesitaban utilizar un compilador, o sin ninguna experiencia en la programación orientada a objetos.

A medida que estuvieron disponibles nuevas versiones de este lenguaje incluían nuevos componentes que dan gran potencial al lenguaje, pero lamentablemente esta versión solo funcionaba en la última versión del Navigator en aquel momento.

En diciembre de 1995, Netscape y Sun Microsystems (el creador del lenguaje Java) luego de unirse objetivo de desarrollar el proyecto en conjunto, reintroducen este lenguaje con el nombre de Javascript. En respuesta a la popularidad de Javascript, Microsoft lanzo su propio lenguaje de programación a base de script, VBScript (una pequeña versión de Visual Basic).

En el año de 1996 Microsoft se interesa por competir con Javascript por lo que lanza su lenguaje llamado Jscript, introducido en los navegadores de Internet Explorer. A pesar de las diferentes críticas que se le hacen al lenguaje Javascript, este es uno de los lenguajes de programación más populares para la

web. Desde que los navegadores incluyen el Javascript, no necesitamos el Java Runtime Environment (JRE), para que se ejecute.

El Javascript es una tecnología que ha sobrevivido por más de 10 años, es fundamentales en la web, junto con la estandarización de la "European Computer Manufacturers Association" (ECMA) (adoptada luego por la ISO) y W3C DOM, Javascript es considerado por muchos desarrolladores web como la fundación para la próxima generación de aplicaciones web dinámicas del lado del cliente.

La estandarización de Javascript comenzó en conjunto con ECMA en Noviembre de 1996. Es adoptado este estándar en Junio de 1997 y luego también por la "Internacional Organization for Standardization" (ISO). El DOM por sus siglas en inglés "Modelo de Objetos del Documento" fue diseñado para evitar incompatibilidades.

1.3.- ¿CÓMO IDENTIFICAR CÓDIGO JAVASCRIPT?

El código javascript podemos encontrarlo dentro de las etiquetas <body></body> de nuestras páginas web. Por lo general se insertan entre: <script></script>. También pueden estar ubicados en ficheros externos usando:

```
<script type="text/javascript" src="micodigo.js"></script>
```

1.4.- ALGUNAS CARACTERÍSTICAS DEL LENGUAJE SON:

Su sintaxis es similar a la usada en Java y C, al ser un lenguaje del lado del cliente este es interpretado por el navegador, no se necesita tener instalado ningún Framework.

- Variables: var = "Hola", n=103
- Condiciones: if(i<10){ … }
- Ciclos: for(i; i<10; i++){ … }
- Arrays: var miArray = new Array("12", "77", "5")
- Funciones: Propias del lenguaje y predefinidas por los usuarios
- Comentarios para una sola línea: // Comentarios
- Comentarios para varias líneas:
  /*
  Comentarios
  */
- Permite la programación orientada a objetos: document.write("Hola");
- Las variables pueden ser definidas como: string, integer, flota, bolean simplemente utilizando "var". Podemos usar "+" para concatenar cadenas y variables.

1.5.- ¿ES COMPATIBLE CON NAVEGADORES?

Javascript es soportado por la mayoría de los navegadores como Internet Explorer, Netscape, Opera, Mozilla Firefox, entre otros.

Con el surgimiento de lenguajes como PHP del lado del servidor y Javascript del lado del cliente, surgió Ajax en acrónimo de (Asynchronous Javascript And XML). El mismo es una técnica para crear aplicaciones web interactivas. Este lenguaje combina varias tecnologías:

- HTML y Hojas de Estilos CSS para generar estilos.
- Implementaciones ECMAScript, uno de ellos es el lenguaje Javascript.
- XMLHttpRequest es una de las funciones más importantes que incluye, que permite intercambiar datos asincrónicamente con el servidor web, puede ser mediante PHP, ASP, entre otros.

Debemos tener en cuenta que aunque Javascript sea soportado en gran cantidad de navegadores nuestros usuarios pueden elegir la opción de Activar/Desactivar el Javascript en los mismos.

1.6.- ¿QUÉ OCURRE SI HAY UN ERROR EN MI CÓDIGO JS?

Pues algo similar: si el navegador encuentra una sentencia sintácticamente incorrecta que produce un error lo más habitual es que detenga de inmediato el motor de JS (de hecho muchos navegadores ni siquiera informan del error). A partir de ese error no se ejecutará más código JS.

SINTAXIS GENERAL DEL JS

- El lenguaje es **case-sensitive**, es decir, diferencia entre mayúsculas y minúsculas.
- Las sentencias se separan con un punto y coma. En realidad este punto y coma no es obligatorio si las sentencias se colocan en líneas distintas. No obstante será una buena práctica utilizarlo siempre, y así lo haremos.
- El lenguaje es **débilmente tipeado**. Existen, al igual que en C++ y Java, tipos primitivos de dºatos. La diferencia es que al declarar una variable no indicamos el tipo de datos y será el intérprete el que lo asigne según el primer valor recibido por la variable.

Ejemplos:

```
numero = 15; // acabamos de crear una variable entera (ojo, es real).
mensaje ="hola"; // acabamos de crear una cadena.
var pi = 3.14; // acabamos de crear una variable de tipo real.
```

Como ves no hemos tenido que especificar el tipo de datos a la hora de declarar variables.

- Dado que existe el punto y coma para separar sentencias todos los espacios en blanco, los tabuladores y los saltos son ignorados por el intérprete. Podríamos escribir un programa completo en una única línea, por razones de legibilidad no se recomienda para el desrrollo.

Actividad 1

Copia el siguiente código en una página web de ejemplo:

```
<script type="text/javascript">
    alert("Saludos");
</script>
```

a) ¿Puedo colocar este código dentro de <body>?
b) ¿Qué hace este código?
c) ¿Qué ocurrirá si mi navegador tiene deshabilitadas las funciones Javascript? (deshabilite la función en su navegador). ¿El usuario puede percatarse de esta situación?
d) Sin habilitar Javascript, introduzca el siguiente código html en el <body>. ¿Qué ocurre con la página al recargarla?

```
<noscript>
    Para utilizar las funcionalidades completas de este sitio es necesario tener
    JavaScript habilitado. Aquí están las <a href="http://www.enable-javascript.com/es/"
    target="_blank"> instrucciones para habilitar JavaScript en tu navegador web</a>.
</noscript>
```

e) Habilite en el navegador las funciones Javascript. Y sustituya la línea de código JavaScript por **`ALERT("Saludos");`** ¿Qué ocurre? ¿Cómo puedo ver los errores de JavaScript?
f) Instala un plugin en tu navegador que te permita gestionar las cachés del navegador y así poder desarrollar sin interferencias de las mismas. Si no conoces alguna utiliza ***Web Developer***.

TIPOS DE DATOS DE JAVASCRIPT

TIPOS DE DATOS EN		NOMBRE	DESCRIPCIÓN aprenderaprogramar.com

JAVASCRIPT	**TIPOS PRIMITIVOS**	String	Cadenas de texto
		Number	Valores numéricos
		Boolean	true ó false
		Null	Tipo especial, contiene null
		Undefined	Tipo especial, contiene undefined
	TIPOS OBJETO	Tipos predefinidos de JavaScript	Date (fechas) RegExp (expresiones regulares) Error (datos de error)
		Tipos definidos por el programador-usuario	Funciones simples Clases
		Arrays	Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos.
		Objetos especiales	Objeto global
			Objeto prototipo
			Otros

Los tipos de datos JavaScript se dividen en dos grupos: tipos primitivos y tipos objeto. Los tipos primitivos incluyen: cadenas de texto (String), variables booleanas cuyo valor puede ser true o false (Boolean) y números (Number). Además hay dos tipos primitivos especiales que son Null y Undefined.
Los tipos objeto son datos interrelacionados, no ordenados, donde existe un nombre de objeto y un conjunto de propiedades que tienen un valor. Un objeto puede ser creado específicamente por el programador. No obstante, se dice que todo aquello que no es un tipo primitivo es un objeto y en este sentido también es un objeto, por ejemplo, una función.

Una variable se declara con la palabra clave ***var*** o ***let***. Por ejemplo var precio; constituye la declaración de una variable denominada precio. En la declaración no figura qué tipo de variable es (por ejemplo si es texto tipo String o si es numérica tipo Number). Entonces, ¿cómo se sabe de qué tipo es una variable? JavaScript decide el tipo de la variable por inferencia. Si detecta que contiene un valor numérico, la considerará tipo Number. Si contiene un valor de tipo texto la considerará String. Si contiene true ó false la considerará booleana.

El nombre de una variable deberá estar formado por letras, números, guiones bajos o símbolos dólar ($), no siendo admitidos otros símbolos. El nombre de la variable no puede empezar por un número: obligatoriamente ha de empezar con una letra, un signo dólar o un guión bajo. Por tanto son nombres de variables válidos precio, $precio, _precio_ , _$dato1, precio_articulo, etc. y no son nombres válidos 12precio ni precio# ó pre!dato1.

Una variable se inicializa cuando se establece su contenido o valor por primera vez. Por ejemplo precio = 22.55; puede ser una forma de inicializar una variable. Una variable se puede declarar e inicializar al mismo tiempo. Por ejemplo podríamos escribir var precio = 22.55; con lo que la variable ha sido declarada e inicializada en una misma línea.

JavaScript no requiere declaración del tipo de las variables, e incluso permite que una variable almacene contenido de distintos tipos en distintos momentos. Por ejemplo podríamos usar precio = 22.55; y en un lugar posterior escribir precio = 'muy caro';. Esto, que en otros lenguajes generaría un error, es aceptado por JavaScript.

JavaScript distingue entre mayúsculas y minúsculas (no sólo para las variables): por tanto no es lo mismo precio = 22.55 que Precio = 22.55. Precio es una variable y precio otra.

JavaScript permite hacer uso de una variable sin que haya sido declarada. En muchos lenguajes de programación es necesario declarar una variable antes de poder hacer uso de ella, pero JavaScript no obliga a ello. Cuando JavaScript se encuentra una variable no declarada, crea automáticamente una variable y permite su uso.

var se usa únicamente para declarar una variable. No puede usarse para otra cosa. Una vez declarada la variable, ya se hará uso de ella sin precederla de la palabra clave var. Si se declara una variable estando ya declarada, JavaScript intentará continuar (y posiblemente lo consiga), pero esto puede considerarse una mala práctica excepto si se sabe muy bien lo que se está haciendo.

3.1.- TIPOS PRIMITIVOS

- El tipo **undefinded**: tiene un único valor que se escribe undefined. Es el tipo de datos de una variable a la que no se le ha asignado ningún tipo de datos ni un valor inicial.
- El tipo **null**: tiene un único valor que se escribe null. Técnicamente un valor null es igual al valor undefined. Sin embargo este tipo de datos se utiliza para aquellos objetos que aún no han sido inicializados, es decir, es utilizado para las variables de tipo objeto (tipos referencia en Java), cuando no están inicializadas.
- El tipo **boolean**: tiene únicamente dos valores, escritos en minúsculas, true y false. Cuando una variable booleana se usa en una expresión numérica los valores true y false pueden sufrir una conversión. False equivale al número cero mientras que cualquier número que no sea el cero se asume que es true, de ahí que en ocasiones la palabra reservada true pueda interpretarse como un uno.
- El tipo **String**: corresponde a las cadenas de caracteres. En JS las cadenas se pueden encerrar indistintamente entre comillas dobles o bien comillas simples. Las cadenas son tratadas como simples arrays de caracteres. Se puede acceder directamente a cada carácter, y el primero está en la posición 0 (cero). El tipo String es un objeto predefinido de JavaScript.
- El tipo **Number**: JS sólo tiene un tipo de datos primitivo para los números, es el tipo Number, que almacena los números en formato IEEE-754 de 64 bits, esto es, en punto flotante. Podríamos asimilarlo al double de Java. Este tipo de datos cuenta con valores especiales:
    - El valor **NaN**, que significa Not A Number puede producirse cuando intentamos guardar en una variable numérica un dato que no es un número. Ejemplo: numero = 10 / “hola”;
    - Los valores **Infinity** y **–Infinity** también están contemplados, tratando de reflejar el infinito, aunque no sirven para operar. Se generan automáticamente cuando sobrepasamos los límites superior e inferior del IEEE-754.

Constante	Descripción	Valor de

		aproximación
Math.E	Constante matemática e. Es el número de Euler, la base de los logaritmos naturales.	2.718
Math.LN2	Logaritmo natural de 2.	0.693
Math.LN10	Logaritmo natural de 10.	2.302
Math.LOG2E	Logaritmo en la base 2 de e.	1.443
Math.LOG10E	Logaritmo en la base 10 de e.	0.434
Math.PI	Pi. Es la relación entre la circunferencia de un círculo y su diámetro.	3.14159
Math.SQRT1_2	Raíz cuadrada de 0,5 o, de forma equivalente, uno dividido entre la raíz cuadrada de 2.	0.707
Math.SQRT2	Raíz cuadrada de 2.	1.414

JavaScript define algunas constantes matemáticas que representan valores numéricos significativos:

Para convertir a número el valor de una variable también podemos utilizar dos funciones de JS: **parseInt**(variable o expresión) y **parseFloat**(variable o expresión). En estas dos funciones se devuelve un número. En el caso de la primera un número entero y en el caso de la segunda un número real, aunque lo devuelto en JS será de un único tipo: Number.
Si el parámetro de estas funciones es una cadena se comienza la interpretación desde el primer carácter y se continúa hasta el final. En el momento en que uno de los caracteres no sea un dígito se aborta el proceso retornando **NaN**. parseFloat además admite el carácter punto como separador decimal.

3.1.1.- ISNAN

isNaN es una función de alto nivel y no está asociada a ningún objeto. Intenta convertir el parámetro pasado a un número. Si el parámetro no se puede convertir, devuelve true; en caso contrario, devuelve false.

Esta función es útil ya que el valor NaN no puede se probado correctamente con operadores de igualdad. x == NaN y x === NaN son siempre false, sin importar lo que sea x, incluso si x es NaN. Por ejemplo, tanto 1 == NaN como NaN == NaN devuelven false.

Ejemplos

```
isNaN(NaN) //devuelve true
isNaN("string") //devuelve true
isNaN("12") //devuelve false
isNaN(12) //devuelve false
```

3.2.- OPERADORES

Operadores aritméticos

OPERADORES BÁSICOS	DESCRIPCIÓN
+	Suma
–	Resta
*	Multiplicación
/	División
%	Resto de una división entre enteros (en otros lenguajes denominado mod)

Las operaciones con operadores siguen un orden de prelación o de precedencia que determinan el orden con el que se ejecutan. Con los operadores matemáticos la multiplicación y división tienen precedencia sobre la suma y la resta. Si existen expresiones con varios operadores del mismo nivel, la operación se

ejecuta de izquierda a derecha. Para evitar resultados no deseados, en casos donde pueda existir duda se recomienda el uso de paréntesis para dejar claro con qué orden deben ejecutarse las operaciones. Por ejemplo, si dudas si la expresión 3 * a / 7 + 2 se ejecutará en el orden que tú deseas, especifica el orden deseado utilizando paréntesis: por ejemplo 3 * ((a / 7) + 2).
En JavaScript el operador + se usa para realizar sumas pero también para concatenar cadenas, es decir, resulta una operación u otra según el tipo de variables a las que se aplique.

Operadores Lógicos y relacionales

OPERADORES LÓGICOS Y RELACIONALES	DESCRIPCIÓN	EJEMPLO
==	Es igual	a == b
===	Es estrictamente igual	a === b
!=	Es distinto	a != b
!==	Es estrictamente distinto	a !== b
<, <=, >, >=	Menor, menor o igual, mayor, mayor o igual	a <=b
&&	Operador and (y)	a && b
||	Operador or (o)	a || b
!	Operador not (no)	!a

Los operadores && y || se llaman **operadores en cortocircuito** porque si no se cumple la condición de un término no se evalúa el resto de la operación. Por ejemplo: (a == b && c != d && h >= k) tiene tres evaluaciones: la primera comprueba si la variable a es igual a b. Si no se cumple esta condición, el resultado de la expresión es falso y no se evalúan las otras dos condiciones posteriores.

Dos cadenas de texto se pueden comparar resultando que se comparan letra a letra por el valor del equivalente numérico de cada letra. Cada letra tiene un número asociado: por ejemplo la a es el número 97, la b el 98, etc. Si comparamos 'avellana' < 'sandia' obtenemos true. Sin embargo, los códigos numéricos pueden generar resultados no previstos. Por ejemplo, ¿qué código numérico es menor, el de la a ó el de la A? Aún más, resulta que todos los códigos numéricos de mayúsculas son menores que los de minúsculas, con lo cual podemos obtener que 'Zulú' < 'avellano' devuelve true (cosa que a priori nos resultará ciertamente extraña). Para comparar cadenas en base a un orden alfabético necesitaremos usar entonces otras técnicas.

OPERADORES	DESCRIPCIÓN	EJEMPLO	EQUIVALE A
=	Asignación	a = 5	
+=	Suma lo indicado	a += b	a = a + b
-=	Resta lo indicado	a -= b	a = a - b
*=	Multiplica por lo indicado	a *= b	a = a * b
%=	Calcula el módulo por lo indicado	a %= b	a = a % b
++ (anterior)	Incremento unitario antes de operar	++a * 2	a = a + 1 a * 2
++ (posterior)	Incremento unitario después de operar	a++ * 2	a * 2 a = a + 1
-- (anterior)	Decremento unitario antes de operar	--a * 2	a = a - 1 a * 2
-- (posterior)	Decremento unitario después de operar	a-- * 2	a * 2 a = a - 1

Los operadores a **nivel de bits**, por ejemplo, trabajan con las representaciones binarias de los operandos. Los operadores de objetos trabajan con variables que son objetos o bien estructuras de datos (como los arrays).

Operador	Uso	Descripción rápida
& (AND)	a & b	Devuelve 1 si ambos operandos son 1
| (OR)	a | b	Devuelve 1 donde uno o ambos operandos son 1
^ (XOR)	a ^ b	Devuelve 1 donde un operando, pero no ambos, es 1
~ (NOT)	~a	Invierte el valor del operando
<< (Desplazamiento a la izquierda)	a << b	Desplaza a la izquierda un número especificado de bits.
>> (Desplazamiento a la derecha)	a >> b	Desplaza a la derecha un número especificado de bits.
>>> (Desplazamiento a la derecha con acarreo)	a >>> b	Desplaza la derecha un número especificado de bits descartando los bits desplazados y sustituyéndolos por ceros.

Actividad 2

a) ¿Qué hace el siguiente código? ¿para que se utiliza **typeof**?

```
<script type="text/javascript">
    alert(typeof(valor));
</script>
```

b) Indica el valor que generan las siguientes expresiones y el tipo de datos de los mismos:

a. 2 > 3 +2
b. 1 + false +3.5
c. 4 * true + 2
d. 5 / 0
e. 10 > 2 > 0
f. 15 < 3 * 10
g. 3 * hola + 2
h. 5 / “hola”
i. $\pi * 3^2$
j. $\sqrt{25}$
k. “Tienes” + 23 + “años”

c) Observa el siguiente código y contesta:

```
<script type="text/javascript">
    function saludar (){
        var nombre = "Pedro";
        alert("Hola "+ nombre);
    }
    alert("Usuario: " + nombre);
    saludar();
</script>
```

a. La variable nombre se define como una variable local o como una variable global.

b. El código tiene un error ¿cómo puedo arreglarlo?

d) [Trate de resolver antes de ejecutar el código, luego puede comprobarlo] ¿Qué mostrará la segunda ventana emergente?

```
<script type="text/javascript">
    var nombre = "Juan";
    function saludar (){
        var nombre = "Pedro";
        alert("Hola "+ nombre);
    }
    saludar();
    alert("Usuario: " + nombre);
</script>
```

e) ¿Qué generan los siguientes códigos?

a.
```
var resultado=true;
resultado.toString();
```

b.
```
var llueve = false;
llueve += " o true?";
```

f) ¿Qué valor producirán las siguientes expresiones?

a. parseInt("15");
b. parseInt("15.5")
c. parseInt(15.5)
d. parseInt("true")
e. parseInt(true)
f. parseInt("Pedro")
g. parseInt(15 + "Pedro")
h. parseFloat("15.5")
i. parseFloat(15.5)

SENTENCIAS.

La escritura de sentencias, sentencias condicionales y bucles en JS sigue la misma sintaxis que ya conoces de Java (en realidad, siguen la sintaxis de C++). Por lo tanto este apartado te resultará completamente familiar.

4.1.- BLOQUES DE CÓDIGO.

Tal y como hemos comentado el separador de sentencias en JS es el punto y coma. Este separador no sería obligatorio si se escribe cada sentencia en una línea diferente, pero es el separador entre sentencias simples.
Si queremos agrupar varias sentencias dentro de una estructura común, esto es, utilizar un bloque de código utilizaremos las llaves ({ … bloque …. }.
Tanto las sentencias condicionales como los bucles pueden ejecutar como su cuerpo una única sentencia simple (en cuyo caso el cuerpo no requerirá de llaves) o bien un único bloque de código (y en ese caso sí utilizaremos las llaves).

SENTENCIAS

5.1.- SENTENCIAS CONDICIONALES

5.1.1.- SENTENCIA IF-ELSE.

La sentencia if permite evaluar una expresión y bifurcar el código dependiendo de si la expresión evaluada es cierta (parte if) o falsa (parte else).

```
if (expresión)
      sentencia_1;
else
      sentencia_2;
```

La expresión va encerrada entre paréntesis, y se trata de una expresión booleana que debe evaluarse produciendo un valor booleano. Recuerda que tanto el else como la sentencia_2 son opcionales. Por otra parte sentencia_1 y sentencia_2 pueden ser sentencias simples o bien un único bloque **delimitado por llaves.**
En el caso que necesitar evaluar varias condiciones binarias (aquellas que son de tipo true/false) y todas están relacionadas entre sí podremos encadenar varias sentencias if-else. A diferencia de lenguajes de Python aquí no existe elif, sino que incluiremos dentro de un else una sentencia if-else.

```
if(expresión) {
  sentencia_1;
}
else if(expresión) {
  sentencia_2;
}
else if(expresión) {
  sentencia_3;
```

```
}
else {
  sentencia_4;
}
```

5.1.2.- SENTENCIA SWITCH.

No siempre las condiciones son booleanas. A veces nos interesa chequear condiciones multievaluadas. Por ejemplo si preguntas a alguien qué día de la semana es su respuesta podrá ser múltiple (lunes, martes, …, sábado, domingo). En estos casos la mejor sentencia es la sentencia switch, que de nuevo sigue la misma estructura sintáctica que conocemos del lenguaje Java.

```
switch(expression) {
    case n:
        sentencia_1
        break;
    case n:
        sentencia_2
        break;
    default:
        sentencia_3
}
```

Por otra parte cada elemento case incluye un valor (que se compara con la expresión) y después dos puntos. Las distintas sentencias que quieras incluir dentro de un elemento case, aunque sean varias no llevarán llaves.
Los elementos case acaban normalmente con un break; aunque esto es opcional.
Al igual que ocurre en C++ o Java si un case no es terminado con un break y la ejecución llega hasta él ocurrirá que se continúa ejecutando el siguiente case sin evaluarse. El break lo que hace es convertir cada case en excluyente. Como sabes la sentencia break finaliza la ejecución de la sentencia switch.
El elemento default es opcional. Si se incluye se ejecutará su contenido sólo en el caso de que la expresión no se cumpla en ninguno de los elementos case.

5.2.- SENTENCIAS REPETITIVAS

5.2.1.- SENTECIA FOR

La sintaxis habitual es:

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

Lo habitual es usar estas tres expresiones con una variable que actúa de contador, y en este caso, las expresiones se usan para lo siguiente:

- `inicialización`: es la inicialización de la variable contadora. Si existen varias inicializaciones habrá que separarlas por comas. Ejemplo: i=10, j=3.
- `condición`: es la condición de iteración del bucle y puede ser simple o compuesta. Ejemplo: i < 20. Léela siempre con la palabra "mientras".
- `paso`: es la sentencia que sirve para actualizar el valor de la variable contadora (incremento o decremento). Ejemplo: i++; Se ejecuta siempre detrás del cuerpo del bucle.

Otra forma menos común, porque suele dar algunos comportamientos inesperados para los programadores noveles, ya que al ejecutase **sobre algunos objetos**, no se puede predecir ni controlar el orden en el que se asignan los campos del objeto a la variable contadora del bucle. Más aún, el orden puede ser distinto en implementaciones diferentes del lenguaje.

```
var palabras = ["Saludos", "Cordiales"];
```

```
for(i in palabras) {
      alert(palabras[i]);
}
```

Actividad 3

Observa el siguiente ejemplo

```
var o = {
 '1': 1,
 'a': 2,
 'b': 3
}

for(let k in o) {
 console.log(k);
}
```

a) ¿Qué valores retorna el bucle?

Observa el siguiente código:

```
var arr=[1,2,3,4];

for(var i=0; i < arr.length; i++) {  //Primer bucle
      console.log(arr[i]);
}

for(var i=0, len=arr.length; i < len; ++i) {  //Segundo bucle
      console.log(arr[i]);
}
```

b) ¿Muestran la misma información el primer y el segundo bucle?
c) ¿Cuál es la diferencia entre los mismos?
d) ¿Existe diferencia entre ++i e i++?
e) Y qué ocurre en el siguiente código

```
var i = 0;
var k = i++;
var i = 0;
var p = ++i;
console.log("El valor de k es "+k+" el valor de p es "+ p);
```

f) ¿Qué muestra el siguiente código?

```
arr.forEach(function(i){console.log(i)});
```

g) Y el siguiente

```
for (i of arr) {
   console.log(i);
}
```

FOREACH

A partir de la versión ES5 se definieron nuevos elementos que permitían iteraciones sobre los arrays de forma mucho más potente y eficiente.

- every: El bucle para la iteración la primera vez se retorne false. (Ver más en https://goo.gl/qjtww6)
- some: El bucle para la iteración la primera vez se retorne true. (Ver más en https://goo.gl/c1jv4A)
- filter: permite definir filtros que incluiría en la iteración a los elementos que coincidan con el filtro e ignorar aquellos que no cumplan con el mismo. (Ver más en https://goo.gl/Tm2S6E)
- map: crea un nuevo array de los valores retornados por las distintas iteraciones. (Ver más en https://goo.gl/VPU8Wr)
- reduce: construye un valor como resultado de cada iteración pasando en anterior valor, algo así como si se tratase de una función recursiva. (Ver más en https://goo.gl/C95Ero)
- reduceRight: igual que el anterior pero en vez de iterar de forma ascendente lo hace de forma descendente (Ver más en http://goo.gl/XV4Gby).

5.2.2.- SENTENCIA WHILE Y DO ... WHILE

Sintaxis del while

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

Sintaxis del do…while

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

Recordemos que la diferencia básica entre while y do…while radica en que el do-while, o mejor dicho, el código que se encuentra dentro de él, se va a ejecutar por lo menos una vez porque la condición la comprueba al final. En el caso del while podría no ejecutarse nunca.

5.3.- CONTROL DE BUCLES Y SALTOS.

Podemos utilizar las conocidas sentencias break, continue y return: modifican el control de flujo de ejecución sin evaluar condición alguna, es decir, provocan un salto dentro de un bucle.

- **Break**: dentro de un bucle o sentencia switch la instrucción break provoca que la ejecución se transfiera a la primera sentencia fuera del bucle (o fuera del switch).
- **Continue**: sólo se usa dentro de bucles. La instrucción continue provoca que la ejecución se transfiera a la evaluación de la condición que dirige el bucle, esto es, en una iteración determinada se deja de ejecutar secuencialmente y se transfiere el control al comienzo de la siguiente iteración.
- **Return**: como ya sabes provoca la salida de una función o método retornando al punto del programa en el que se hizo la llamada. También es empleada para transferir el valor de retorno del método

FUNCIONES

6.1.- DECLARACIÓN.

Las funciones en JS se declaran utilizando la palabra reservada function (en minúsculas). Dado que JS es un lenguaje débilmente tipeado no es necesario indicar el tipo de los parámetros al declarar una función. Por el contrario este tipo será asignado durante la ejecución de una llamada, al copiarse los parámetros reales (aquellos que se usan en la llamada) en sus correspondientes parámetros formales (aquellos que se definen en la declaración). Observa esta función:

```
function mifuncion (a, b) {
      return a + b;
}
```

Llamadas correctas a la misma serían:

```
var resultado = mifuncion(10, 15);
var mensaje = mifuncion("Hola ", "bienvenido");
var otro = mifuncion (25+3, " veces");
var que_soy = mifuncion; // Si no utilizo paréntesis la variable es una función
```

Recuerda que en las llamadas a las funciones si encontramos expresiones en los parámetros reales éstas se evalúan antes de comenzar a ejecutar el cuerpo de la función.

6.2.- PARÁMETROS Y VALOR DE RETORNO.

Los lenguajes de scripting insertan su código dentro de las páginas web. Como sabes el navegador intentará no detenerse si encuentra errores en el código HTML, simplemente ignorará aquello que no comprende.

Con las funciones existen mecanismos para minimizar el impacto de posibles errores. Si una función tiene tres parámetros pero se realiza una llamada con cinco, los dos últimos serán simplemente ignorados y se continuará con la ejecución (en este caso no se paraliza el motor de JS).

Por otra parte si faltan parámetros reales en la llamada, a todos aquellos que no han sido pasados se les asigna automáticamente el valor **undefined** (y tampoco se detiene el motor de JS).

En cuanto al valor de retorno otra característica de JS es que no soporta procedimientos. Todas las funciones se asume que retornan algo. Sin embargo el **return** no es obligatorio, sólo lo usaremos cuando lo necesitemos. Si una función carece de sentencia **return** se asume que devuelve el valor **undefined**.

Qué ocurre cuando se invoca una función y …

- Los parámetros que sobran se descartan.
- Los parámetros que faltan adquieren el valor undefined.
- Una función sin return se asume que devuelve undefined

6.3.- ANIDAMIENTO Y FUNCIONES ANÓNIMAS (O LAMBDA).

Hay más características que diferencian JS de otros lenguajes y en este apartado veremos dos de estas características.

En JS sí que es posible anidar una función dentro de otra. Observa el siguiente código:

```
function calculaHipotenusa(catetoA, catetoB) {
      function cuadrado(x) {
            return x*x;
      }

      function sumaCuadrados(a,b) {
            return cuadrado(a) + cuadrado(b);
      }

      return Math.sqrt(sumaCuadrados(catetoA, catetoB));
}
```

Como ves dentro de la función `calculaHipotenusa()` hay definidas otras dos funciones: `cuadrado()` y `sumaCuadrados()`. Lógicamente las funciones anidadas sólo podrán ser utilizadas en la función que contiene su declaración, mientras que fuera de ella no existirán.

Por otra parte JS soporta las funciones anónimas, esto es, funciones que carecen de identificador. Veamos un ejemplo:

```
var saludo = function(nombre, edad) {
      return "Bienvenido " + nombre + ". Su edad es " + edad;
}
console.log (saludo("Eva", 19));
```

Viendo este código es casi seguro que piensas que esto no tiene utilidad y además introduce una complejidad innecesaria en los programas. Aquí hemos guardado una función anónima (no tiene nombre) dentro de una variable y después hemos utilizado esta misma variable como si fuera una función. En JS habrá variables que en realidad sean funciones. Esto es así porque JS considera las funciones como simples objetos.

6.4.- USO DE ARGUMENTOS.

Ya hemos visto como las funciones en JS pueden invocarse con un número indeterminado de parámetros y en ocasiones sobrarán o faltarán parámetros sin que la ejecución de código se detenga o se vea afectada.
Para una mayor flexibilidad JS cuenta con una variable predefinida llamada **arguments** de la clase **Arguments**. Podemos tratar esta variable como un vector que puede ser utilizado dentro de cualquier función y tendrá almacenados los distintos parámetros reales.
Esto convierte al lenguaje en uno de los más flexibles (y por lo tanto en ocasiones críptico). Imagina que diseñas una función que no recibe ningún parámetro, pero luego puedes llamarla con el número que parámetros reales que quieras. Dentro de su cuerpo sólo tendrás que usar **arguments** para conocer cuántos eran los parámetros reales y sus valores.

```
function imprimirParamentros(a, b) {
      if (arguments.length != 2)
            alert("El uso recomendado incluye sólo dos parámetros");
      for (i=0; i<arguments.length; i++)
            alert(arguments[i]);
}
```

En realidad **arguments** es un objeto de la clase **Arguments**, y por lo tanto tendrá métodos y propiedades específicos. Una propiedad interesante es **callee**: contiene una referencia la función que se está ejecutando.

```
      function saludar(mensaje) {
            alert(arguments.callee);
            alert(mensaje);
      }
```

Ejemplo:

```
var obj = [{edad:20,fuerza:40}, {edad:30,fuerza:50}, {edad:10,fuerza:23}, {edad:21,fuerza:40}];
alert(masGrande(obj, function(a,b){
  if((50-a.edad)*a.fuerza>(50-b.edad)*b.fuerza){
     return true;
  }else{
     return false;
  }
}));
```

VALIDACIÓN Y EXPRESIONES REGULARES

La mayoría de las aplicaciones actuales guardan su información en bases de datos. Esta información debe ser protegida y la consistencia de la base de datos debe ser robusta. Por ese motivo debemos realizar validaciones o chequeos de cualquier dato que el usuario introduce en un formulario observando si se ajusta al formato requerido.

Imagina que le pedimos a un usuario su dirección y en el código postal (que en España es un conjunto de 5 dígitos) nos escribe un texto. No debemos dejar que se produzca el registro de este dato en la base de datos.
Habitualmente esta validación se realiza en el servidor utilizando el lenguaje de servidor empleando (PHP, ASP.net, JSP, CGI, Servlets, etc). Sin embargo debemos intentar que al servidor lleguen los datos lo más seguro posibles para que éste no pierda tiempo rechazando muchas páginas. El servidor es un recurso muy valioso puesto que todos los usuarios de la aplicación (que pueden ser miles) hacen uso de esta máquina.
Con el objeto de descargar algo de trabajo en el servidor se encontró la necesidad de realizar validaciones en el cliente (habitualmente mediante JS). La ventaja de validar los datos en el cliente es que estos chequeos se llevan a cabo en su propia máquina con lo que el usuario no tiene tiempos de espera largos ni realiza conexiones extra a la red.

7.1.- LAS EXPRESIONES REGULARES

Los métodos de validación rudimentarios consiste en la manipulación de cadenas en JS. Sin embargo para determinados datos lo que necesitaremos es que sigan un patrón. Por ejemplo si estoy validando un DNI el patrón es 8 números y a continuación una letra.
Las expresiones regulares permiten describir estos patrones y constituyen un modelo mucho más potente y rápido para realizar validaciones. No sólo se utilizan en JS, muchos lenguajes o incluso las terminales de Linux permiten emplearlas.
Como ventaja tenemos la potencia y facilidad de uso (los códigos serán más cortos y efectivos). Como desventaja tenemos que en sí mismas constituyen casi un lenguaje nuevo, con lo que la curva de aprendizaje es algo más lenta.
Crear una expresión regular es sencillo:

```
var expresion = new RegExp("^sa"); // patrón para las cadenas que empiezan por "sa"
```

Pero hay una manera más rápida de hacerlo.

```
var expresión = /^sa/; // patrón para las cadenas que empiezan por "sa"
```

Observa: en este caso no usamos comillas sino barras / … /

Dentro de los patrones habrá una serie de carácteres que son especiales y aprender a utilizarlos es lo que dará potencia a las expresiones regulares. En el ejemplo vimos que el carácter ^ significa "**comienza por**".
Una vez creado el objeto RegExp (de cualquiera de las dos formas vistas en el ejemplo anterior), podemos utilizar sus métodos para comprobar concordancias del patrón. Comprobemos ahora si algunas cadenas comienzan por "sa":

```
var expresion = /^sa/; // patrón para las cadenas que empiezan por "sa"
var cadena1 = "sapo", cadena2 = "casa";
alert(expresion.test(cadena1)); // mostrará true pues "sapo" SÍ comienza por "sa"
alert(expresion.test(cadena2));  // mostrará false pues "casa" NO comienza por "sa"
```

Así se usan las expresiones regulares. Reciben una cadena y dentro de la misma buscan coincidencias con un patrón. Y utilizan carácteres especiales. Antes de aprenderlos presentemos la API del objeto **RegExp**:

```
var cadena="dsdjgleajieklasdkr233342 fdf 44";
/glea/.test(cadena);
cadena.match(/glea/);
```

Mientras que test retorna true o false, match obtiene un objeto de tipo RegExp

```
[ 'glea', index: 4, input: 'dsdjgleajieklasdkr233342 fdf 44' ]
```

```
cadena.match(/a/g);
¿Qué hace el modificador g?
```

PROPIEDADES

global: Se ha utilizado el modificador g.
ignoreCase: Se ha utilizado el modificador i.
lastIndex: Índice donde comenzará la siguiente búsqueda del patrón (match)
multiline: Se ha utilizado el modificador m.
source: Guarda el texto de la expresión regular.

MÉTODOS

compile(): Compila la expresión regular. Utilizado para el caso de patrones que pueden modificarse durante la ejecución del programa. Si tenemos esta necesidad no crearemos expresiones regulares de la forma /patrón/ (éstas no pueden cambiar), sino que emplearemos el constructor new RegExp("patrón").
exec(**cadena**) Busca la coincidencia en la cadena. Devolverá la primera coincidencia
test(**cadena**): Busca la coincidencia en la cadena, pero devuelve True o False

Puedes probar las expresiones regulares en: http://rubular.com/ y en https://regex101.com/ donde puedes ver el uso de los modificadores (flag)

Más información sobre el objeto RegExp en https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/RegExp

LOS OBJETOS EN JAVASCRIPT

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

https://msdn.microsoft.com/es-es/library/202863ha(v=vs.94).aspx

ÁRBOL DE NODOS

Una de las tareas habituales en la programación de aplicaciones web con JavaScript consiste en la manipulación de las páginas web. De esta forma, es habitual obtener el valor almacenado por algunos elementos (por ejemplo los elementos de un formulario), crear un elemento (párrafos, , etc.) de forma dinámica y añadirlo a la página, aplicar una animación a un elemento (que aparezca/desaparezca, que se desplace, etc.).

Todas estas tareas habituales son muy sencillas de realizar gracias a DOM[1]. Sin embargo, para poder utilizar las utilidades de DOM, es necesario "transformar" la página original. Una página HTML normal no es más que una sucesión de caracteres, por lo que es un formato muy difícil de manipular. Por ello, los navegadores web transforman automáticamente todas las páginas web en una estructura más eficiente de manipular.

[1] DOM o *Document Object Model* es un conjunto de utilidades específicamente diseñadas para manipular documentos XML. Por extensión, DOM también se puede utilizar para manipular documentos XHTML y HTML. Técnicamente, DOM es una API de funciones que se pueden utilizar para manipular las páginas XHTML de forma rápida y eficiente.

Esta transformación la realizan todos los navegadores de forma automática y nos permite utilizar las herramientas de DOM de forma muy sencilla. El motivo por el que se muestra el funcionamiento de esta transformación interna es que condiciona el comportamiento de DOM y por tanto, la forma en la que se manipulan las páginas.

DOM transforma todos los documentos XHTML en un conjunto de elementos llamados nodos, que están interconectados y que representan los contenidos de las páginas web y las relaciones entre ellos. Por su aspecto, la unión de todos los nodos se llama "árbol de nodos".

9.1.- ACCESO DIRECTO A LOS NODOS

Como acceder a un nodo del árbol es equivalente a acceder a "un trozo" de la página, una vez construido el árbol, ya es posible manipular de forma sencilla la página: acceder al valor de un elemento, establecer el valor de un elemento, mover un elemento de la página, crear y añadir nuevos elementos, etc. DOM proporciona dos formas alternativas para acceder a un nodo específico: acceso a través de sus nodos padre (como una especie de ruta) y acceso directo (accedo directamente al nodo).

Las funciones que proporciona DOM para acceder a un nodo a través de sus nodos padre consisten en acceder al nodo raíz de la página y después a sus nodos hijos y a los nodos hijos de esos hijos y así sucesivamente hasta el último nodo de la rama terminada por el nodo buscado. Sin embargo, cuando se quiere acceder a un nodo específico, es mucho más rápido acceder directamente a ese nodo y no llegar hasta él descendiendo a través de todos sus nodos padre.

Por ese motivo, no se van a presentar las funciones necesarias para el acceso jerárquico de nodos y se muestran solamente las que permiten acceder de forma directa a los nodos.

Por último, es importante recordar que el acceso a los nodos, su modificación y su eliminación solamente es posible cuando el árbol DOM ha sido construido completamente, es decir, después de que la página XHTML se cargue por completo. Más adelante se verá cómo asegurar que un código JavaScript solamente se ejecute cuando el navegador **ha cargado entera la página** XHTML.

9.1.1.- GETELEMENTBYID()

La función getElementById() es la más utilizada, devuelve el elemento XHTML cuyo atributo id coincide con el parámetro indicado en la función. Como el atributo id debe ser único para cada elemento de una misma página, la función devuelve únicamente el nodo deseado.

```
var cabecera = document.getElementById("cabecera");
```

```
<div id="cabecera">
  <a href="/" id="logo">...</a>
</div>
```

9.1.2.- GETELEMENTSBYTAGNAME()

La función getElementsByTagName(nombreEtiqueta) obtiene todos los elementos de la página XHTML cuya etiqueta sea igual que el parámetro que se le pasa a la función. El siguiente ejemplo muestra cómo obtener todos los párrafos de una página XHTML:

```
var parrafos = document.getElementsByTagName("p");
```

El valor que se indica delante del nombre de la función (en este caso, document) es el nodo a partir del cual se realiza la búsqueda de los elementos. En este caso, como se quieren obtener todos los párrafos de la página, se utiliza el valor document como punto de partida de la búsqueda.

El valor que devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. El valor devuelto es un array de nodos DOM, no un array de cadenas de texto o un array de objetos normales. Por lo tanto, se debe procesar cada valor del array de la forma que se muestra en las siguientes secciones.

De este modo, se puede obtener el primer párrafo de la página de la siguiente manera:

```
var primerParrafo = parrafos[0];
```

De la misma forma, se podrían recorrer todos los párrafos de la página con el siguiente código:

```
for(var i=0; i < parrafos.length; i++) {
  var parrafo = parrafos[i];
}
```

La función getElementsByTagName() se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función. En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
var parrafos = document.getElementsByTagName("p");
var primerParrafo = parrafos[0];
var enlaces = primerParrafo.getElementsByTagName("a");
```

9.1.3.- GETELEMENTSBYNAME()

La función getElementsByName(), es muy parecida a la anterior, se buscan los elementos cuyo atributo **name** sea igual al parámetro proporcionado. En el siguiente ejemplo, se obtiene directamente el único párrafo con el nombre indicado:

```
var parrafoEspecial = document.getElementsByName("especial");

<p name="prueba">...</p>
<p name="especial">...</p>
<p>...</p>
```

Normalmente el atributo name es único para los elementos HTML que lo definen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML radiobutton, el atributo name es común a todos los radiobutton que están relacionados, por lo que la función devuelve una colección de elementos, es decir un array.

Cualquier elemento HTML identificado podrá ser accesible como un desde JS. Por ejemplo si tenemos una imagen:

```
<form name=”miformulario”>
<img name=”imagen” src=”retrato.jpg”/>
</form>
```

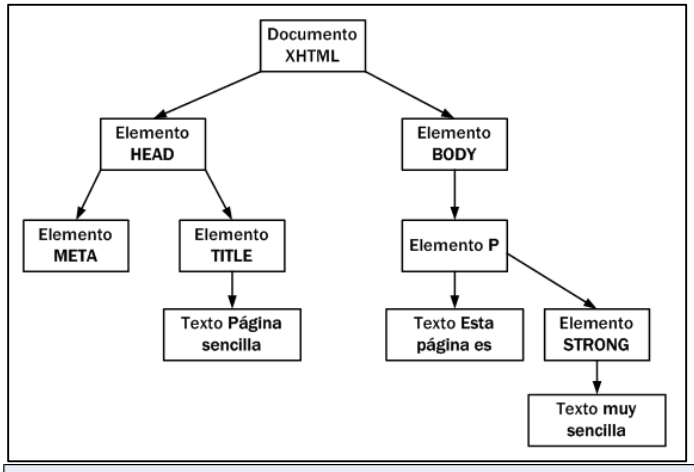
Desde JS podremos acceder a cualquier atributo del elemento como si fuera un atributo del objeto. Continuando con el ejemplo:

```
document.miformulario.imagen.src = “miperro.gif”;
// cambiamos atributo src
```

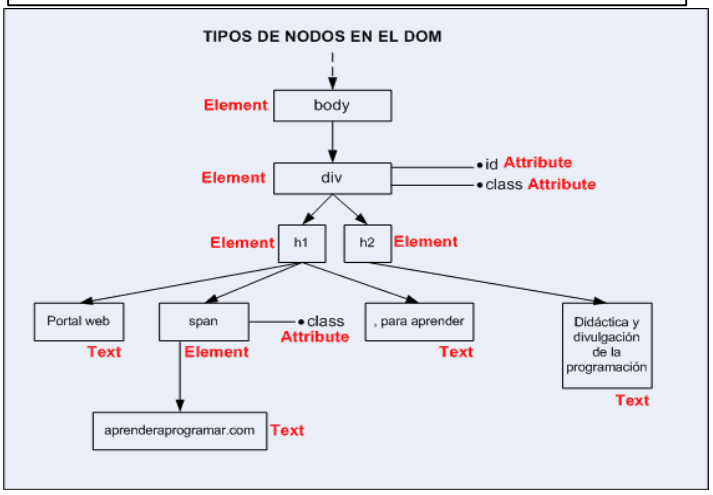
ACTIVIDAD 3

a) Obtenga el código html relacionado con los siguientes árboles de nodos. En el segundo ejemplo puede ver los atributos asociados.

a.



b.



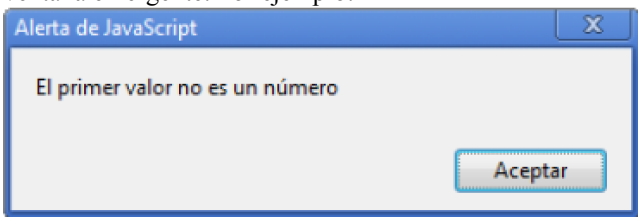
b) ¿Qué valor se obtendrá al ejecutar estas expresiones?

a. 16 << 2
b. 8 > 10 ? v1 = “si” : v1 = “no”
c. var n=7; alert(“n vale “ + (++n));
d. var m=17; alert("m vale " + (m++)); alert(m);
e. 3 ^ 2
f. var y=17; y%=5;
g. var caja = ["percha", "balon"]; alert(caja[1]);
h. var a=3,b=6; var h = a > b ? a : b;

c) Cree el siguiente formulario en HTML.

a. Cuando se pulse el botón = se ejecutará una función con el nombre multiplicar que leerá el primer y segundo campo y guardará en el tercero el resultado. Utilice el atributo `onclick="multiplicar()"`
b. Utilice las tres formas de acceder a los nodos del DOM getElementsByTagName, getElementsByName y getElementById para leer/escribir la información, una distinta para cada campo de texto. Una vez que obtenga el objeto puede acceder a la información utilizando el atributo **.value**.

c. Compruebe que el campo de texto tiene un valor numérico antes de hacer la operación, en el caso de pasar un valor que no sea numérico deberá mostrarse un error en una ventana emergente. Por ejemplo:



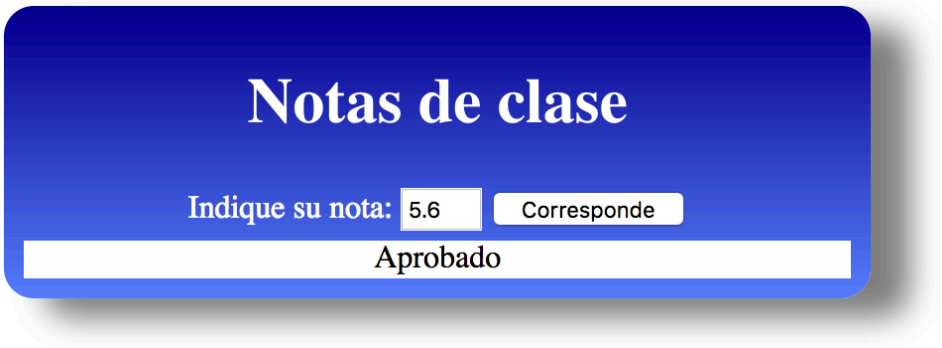
ACTIVIDAD 4

Cree el siguiente formulario en html

a) El campo de texto recogerá numeros reales, por ejemplo 6.5.
b) Cuando se pulse el botón se mostrará una ventana emergente con la siguiente información
   a. Si la nota está entre 0 y 5 el mensaje indicará: **Suspenso**
   b. Si la nota está entre 5 y 7 el mensaje indicará: **Aprobado**
   c. Si la nota está entre 7 y 9 el mensaje indicará: **Notable**
   d. Si la nota está entre 9 y 10 el mensaje indicará: **Sobresaliente**
   e. Si no se introduce un número real: ERROR! – Indique un número del rango 0 a 10
   f. Si es un número que no está comprendido entre 0 y 10: **ERROR! – Indique un número del rango 0 a 10**
c) Al final incluya cinco estructuras y utilice un identificador para cada uno. Cada div contendrá los distintos mensajes que se definen en el anterior ejercicio: Suspenso, Aprobado,…, ERROR! – Indique un número del rango 0 a 10

Utilice css para ocultar los div al iniciar la página.
Cuando pulse el botón, en vez de generar una ventana emergente (comentar este código) mostrará el div correspondiente utilizando el atributo:

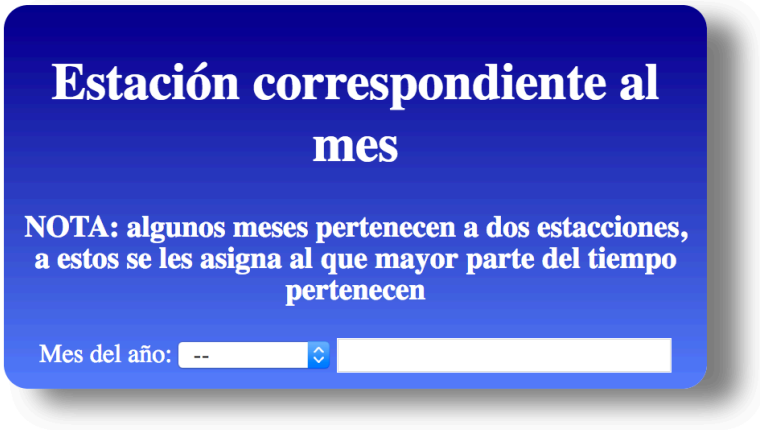
```
XXX.style.display="block";   /* Para mostrar el contenido. XXX corresponde con el id del div que se va a mostrar */
```



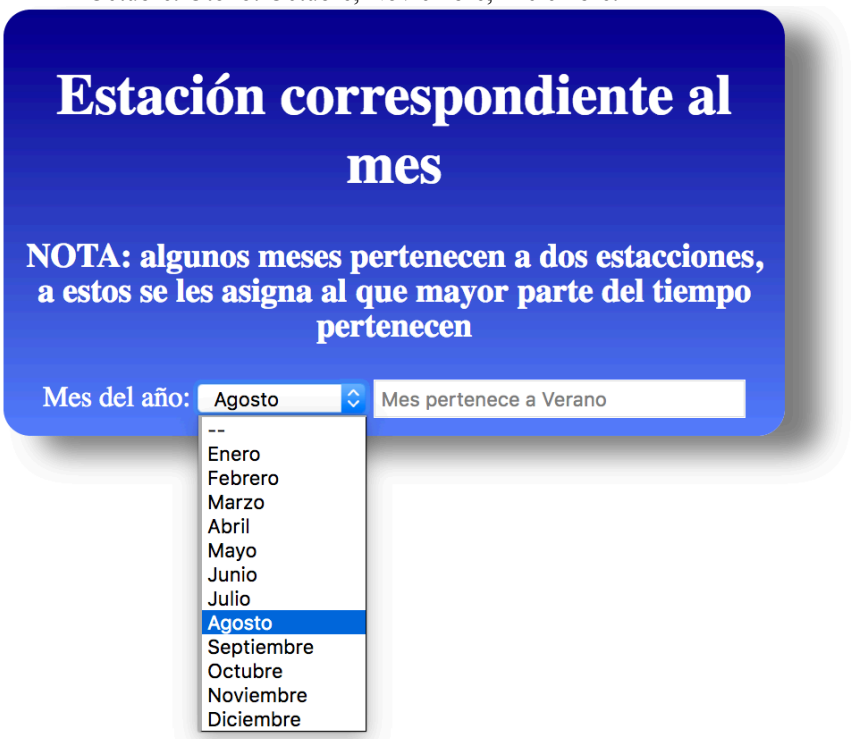
d) Cuando se de un caso de error limpie (borre) la caja de texto.

ACTIVIDAD 5

Cree el siguiente formulario en html



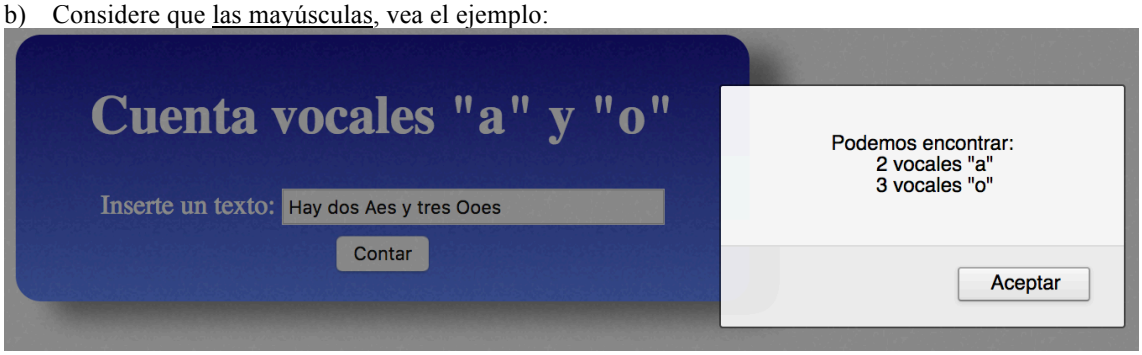
a) Cree un array con las 4 estaciones.
b) Utilice el objeto <select> y el evento evento **onchange** para que al seleccionar uno de los meses en la caja de textos aparecerá un mensaje con la estación correspondiente. Utilice la **cláusula switch**. Invierno: Enero, Febrero, Marzo. Primavera: Abril, Mayo, Junio. Verano: Julio, Agosto, Octubre. Otoño: Octubre, Noviembre, Diciembre.



ACTIVIDAD 6

Cree el siguiente formulario en html

a) Se insertará un texto y al pulsar el botón se mostrará una ventana emergente con las vocales **a** y vocales **o** que contiene el texto. NO utilice expresiones regulares, utilice el método **split()** en JS para separar las letras de la cadena.
b) Considere que las mayúsculas, vea el ejemplo:



FUENTES

http://www.maestrosdelweb.com/que-es-javascript/
http://librosweb.es/libro/javascript/
http://librosweb.es/
http://www.aprenderaprogramar.es/index.php?option=com_content&view=article&id=788:tipos-de-datos-javascript-tipos-primitivos-y-objeto-significado-de-undefined-null-nan-ejemplos-cu01112e-&catid=78:tutorial-basico-programador-web-javascript-desde-&Itemid=206
https://stackoverflow.com/questions/9329446/for-each-over-an-array-in-javascript
https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/every