PROGRAMACIÓN CON FUNCIONES, ARRAYS Y OBJETOS DEFINIDOS POR EL USUARIO

1. FUNCIONES PREDEFINIDAS DEL LENGUAJE

JavaScript cuenta con una serie de funciones predefinidas, es decir, funciones que ya están integradas en el lenguaje. Muchas funciones pueden realizar sus tareas sin necesidad de ninguna información extra. Sin embargo, la mayor parte de las funciones deben acceder a valores de variables para producir sus resultados. Estas variables que reciben las funciones se denominan los argumentos o parámetros de la función.

Algunas funciones predefinidas de JavaScript.

escape():
La función escape () tiene como argumento una cadena de texto y devuelve dicha cadena utilizando la codificación hexadecimal en el conjunto de caracteres latinos IS0. Si por ejemplo quisiéramos saber la codificación del carácter (?), podríamos utilizar el siguiente ejemplo y verificar que obtendríamos la codificación %3F.

```
<script>
        var input = prompt("Introduce una cadena”);
        var inputCodificado = escape(input);
        alert("Cadena codificada: " + inputCodificado);
</script>
```

Unescape() es la función opuesta a escape(), es decir, que esta función decodifica los caracteres que estén codificados.

eval():
Esta función tiene como argumento una expresión y devuelve el valor de la misma para poder ser ejecutada como código JavaScript. El siguiente ejemplo permite ingresar al usuario una operación numérica y a continuación muestra el resultado de dicha operación:

```
<script>
        var input = prompt("Introduce una operación numérica");
        var resultado = eval(input);
        alert ("El resultado de la operación es: " + resultado);
</script>
```

isFinite():
Esta función comprueba si el valor pasado como argumento corresponde o no a un número finito. En JavaScript un valor se define como finito si se encuentra en el rango de $\pm 1.7976931348623157 * 10^{308}$. Si el argumento no se encuentra en este rango o no es un valor numérico, la función devuelve false, en caso contrario devuelve true. Esta función es útil a la hora de realizar comprobaciones en sentencias condicionales y decidir en base al resultado si ejecutar una serie de instrucciones u otras. De este modo podemos comprobar que los resultados de las operaciones matemáticas no

sobrepasen los límites numéricos de JavaScript y evitar la generación de errores de este tipo en nuestra aplicación web.

```
if(isFinite(argumento)){
        //instrucciones si el argumento es un número finito
}else{
        //instrucciones si el argumento no es un número finito
}
```

isNaN():
isNaN() es el acrónimo de is Not a Number (no es un número). Esta función evalúa si el objeto pasado como argumento es de tipo numérico.

```
<script>
        var input = prompt("Introduce un valor numérico: ");
        if (isNaN(input)) {
                alert("El dato ingresado no es numérico.");
        }else{
                alert("El dato ingresado es numérico.");
        }
</script>
```

String():
La función String() convierte el objeto pasado como argumento en una cadena de texto. Por ejemplo, podemos crear un objeto de tipo Date, y convertir dicho objeto en una cadena de texto.

```
<script>
        var fecha = new Date();
        var fechaString = String(fecha);
        alert("La fecha actual es: "+fechaString);
</script>
```

**Number(**):
La función Number () convierte el objeto pasado como argumento en un número. Si la conversión falla, la función devuelve NaN (Not a Number).

Si el parámetro es un objeto de tipo Date, la función Number() devolverá el número de milisegundos transcurridos desde la medianoche del 1 de enero de 1970 hasta la fecha actual.

parseInt():
Esta función intenta convertir una cadena de caracteres pasada como argumento en un número entero con una base especificada. Si no especificamos la base se utiliza automáticamente la base decimal. La base puede ser por ejemplo binaria, octal, decimal o hexadecimal. Si la función encuentra en la cadena a convertir, algún carácter que no sea numérico, devuelve el valor encontrado hasta ese punto. Si el primer valor no es numérico, la función devuelve NaN.

```
<script type="text/javascript">
```

```
        var input = prompt("Introduce un valor: ");
        var inputParsed = parseInt(input);
        alert ("parseInt ("+input+") = "+inputParsed);
</script>
```

parseFloat():
Esta función es muy similar a la anterior. La diferencia es que en lugar de convertir el argumento a un número entero, intenta convertirlo a un número de punto flotante.

```
<script type="text/javascript">
        var input = prompt("Introduce un valor: ");
        var inputParsed = parseFloat(input);
        alert ("parseFloat ("+input+") : " + inputParsed);
</script>
```

2. FUNCIONES DEL USUARIO

Se puede crear nuevas funciones personalizad as con el fin de llevar a cabo las tareas que queramos.

Podemos pensar en una función como un grupo de instrucciones relacionadas para realizar una tarea. A este grupo de instrucciones debemos darle un nombre. Cuando queramos ejecutar este grupo de instrucciones en cualquier otra parte de la aplicación, simplemente debemos utilizar el nombre que le hayamos dado a la función.

Hasta este momento, hemos visto ejemplos en los cuales las instrucciones de JavaScript se encontraban entre las etiquetas <script> y </script>. El navegador se encargaba de ejecutar automáticamente estas instrucciones en el momento de cargar la página. No obstante, todas las instrucciones que se encuentren dentro de una función se ejecutarán solo en el momento en que invoquemos dicha función.

DEFINICIÓN DE FUNCIONES

El mejor lugar para definir las funciones JavaScript es dentro las etiquetas <head> y </head>. El motivo de esto es que el navegador carga siempre todo lo que se encuentra entre estas últimas etiquetas, con lo cual todos los otros guiones de JavaScript que se encuentren en el cuerpo de la página web, ya conocerán la definición de la función.

La sintaxis de la definición de una función es la siguiente:

```
function nombre_función ([argumentos]) {
        grupo_de_instrucciones;
        [return valor;]
}
```

- **Function**. Es la palabra clave que debemos utilizar antes de definir cualquier función.
- **Nombre**. El nombre de la función se sitúa al inicio de la definición y antes del paréntesis que contiene los posibles argumentos. El nombre debe cumplir ciertas reglas con el fin de obtener un correcto funcionamiento:
    - Deben usarse solo letras, números o el carácter de subrayado.

- Debe ser único en el código JavaScript de la página web, ya que dos funciones no pueden tener el mismo nombre.
- No pueden empezar por un número.
- No puede ser una de las palabras reservadas del lenguaje.

Además de estas condiciones, el nombre debería ser representativo de la tarea realizada por el grupo de instrucciones que ejecute la función. Esto se considera una buena práctica de programación.

- **Argumentos**. Los argumentos los definimos dentro del paréntesis que se encuentra a la derecha del nombre de la función. Por ejemplo, si definimos una función que comprueba la identificación y la contraseña de un usuario, debemos pasarle estos dos valores como argumentos en el momento en que llamemos a esta función. Algunas de ellas no necesitan ningún valor para llevar a cabo su tarea, con lo cual el paréntesis se deja vacío en la definición.
- **Grupo de instrucciones**. El grupo de instrucciones es el bloque de código JavaScript que se ejecuta cuando invocamos a la función desde otra parte de la aplicación. Las llaves ({ }) delimitan el inicio y el fin de las instrucciones.
- **Return**. La palabra clave return es opcional en la definición de una función. Esta palabra indica al navegador que devuelva un valor a la sentencia que haya invocado a la función. Si tenemos una función que verifica la identificación y la contraseña de un usuario, es lógico esperar que la función devuelva por ejemplo un valor booleano que puede ser true (verdadero) o false (falso). No todas las funciones deben utilizar la palabra clave return.

A continuación presentamos un ejemplo de la definición de una función que calcula el importe de un producto alimenticio después de haberle aplicado el impuesto sobre el valor añadido (IVA) a los productos de primera necesidad según la legislación española:

```
function aplicar_IVA(valorProducto) {
        var productoConIVA = valorProducto * 1.04; // IVA del 4%
        alert("El precio del producto con IVA es: " + productoConIVA);
}
```

En el grupo de instrucciones de una función podemos declarar nuevas variables. Estas variables se denominan variables locales. Los otros fragmentos de código JavaScript fuera de la definición de la función, desconocen dicha variable y no la podemos utilizar en ninguna otra parte que no sea la misma función. Por el contrario, las variables declaradas fuera de las funciones o fuera de cualquier otro procedimiento se denominan variables globales. Estas últimas podemos utilizarlas en cualquier parte de la aplicación, incluso dentro de una función.

INVOCACIÓN DE FUNCIONES

Una vez que la función personalizada esté definida, necesitaremos llamarla para que el navegador ejecute el grupo de instrucciones que realizan la tarea para la cual hemos definido la función. Una función se invoca usando su nombre seguido de paréntesis. Si la función tiene argumentos, estos deben estar dentro del paréntesis y en el mismo orden en el que los hemos definido en la función.

```
function aplicar IVA(valorProducto, IVA) {
        var productoConIVA = valorProducto * IVA;
        alert("El precio del producto, aplicando el IVA del " + IVA + " es: " + productoConIVA);
}
```

No es lo mismo invocar aplicar_IVA(300, 1.18) que aplicar_IVA(1.18, 300).

Existen diferentes formar de llamar o invocar una función:

- **Invocar una función desde JavaScript**:

```
<!doctype html>
<html lang="es">
	<head>
		<meta charset="utf-8">
		<title>Invocar función desde JavaScript</title>
		<script>
			function mi_funcion([args]) {
				//instrucciones
			}
		</script>
	</head>
	<body>
		<script>
			mi_funcion([args]);
		</script>
	</body>
</html>
```

- **Invocar una función desde HTML**. Es posible invocar funciones también desde código HTML. La definición debe seguir las mismas indicaciones explicadas anteriormente, con la única diferencia que la llamada a la función la establecemos como si fuese un valor de un atributo de una etiqueta HTML. Es muy común invocar funciones de JavaScript desde algunos atributos HTML como por ejemplo onload, onunload u onclick. De este modo ejecutamos las funciones JavaScript en el momento de cargar una página, cerrarla o presionar un botón de la aplicación web.

```
<!doctype html>
<html lang="es">
	<head>
		<meta charset="utf-8">
		<title>Invocar función desde HTML</title>
		<script>
			function mi_funcion([args]) {
				//instrucciones
			}
		</script>
	</head>
	<body onload="mi_funcion([args])"></body>
</html>
```

Hemos visto que la llamada a las funciones las podemos realizar desde cualquier JavaScript o desde el código HTML. Esto significa que las funciones pueden invocar a su vez a otras funciones. Simplemente en las instrucciones de una función debemos utilizar el nombre de otra función. Esto es una práctica muy común de los desarrolladores de aplicaciones web con JavaScript. Por lo general, una aplicación web se divide en varias funciones, cada una de las cuales gestiona una tarea de la aplicación.

La mayoría de las funciones devuelven un valor después de llevar a cabo su grupo de instrucciones. Estas funciones están diseñadas para realizar una tarea y posteriormente devuelven un valor a la sentencia que la haya invocado. Incluso muchos programadores definen funciones que devuelven valores que contienen simplemente un valor que indica si las instrucciones se han ejecutado con éxito.

El siguiente ejemplo es una aplicación que solicita al usuario el resultado de una operación aritmética para verificar que este no sea una máquina que trata de acceder automáticamente a nuestra aplicación. En base al resultado, se muestra el contenido de una página web o un mensaje emergente con un aviso de error. La aplicación define dos funciones JavaScript. La primera función captura el resultado ingresado por el usuario e invoca otra función que verifica si el resultado es correcto. En esta segunda función utilizamos la palabra clave return para devolver un valor booleano que depende de la comprobación del resultado ingresado por el usuario.

```
<!doctype html>
<html lang="es">
	<head>
		<meta charset="utf-8">
		<title> Retorno de valor en una función </title>
		<script>
			function ComprobarHumano() {
				var resultado = prompt("Introduce el resultado de 144/12: ");
				var humano = Verificacion(resultado);
				if(humano == true) {
					document.write("Has ingresado el resultado correcto y podrás
					ver el contenido.");
				}else{
					alert("No has introducido el valor correcto");
				}
			}
			function Verificacion(res) {
				var compruebaResultado;
				if (res == 12) {
					compruebaResultado=true;
				}else{
					compruebaResultado=false;
				}
				return compruebaResultado;
			}
		</script>
```

```
    </head>
    <body>
        <script>
            ComprobarHumano() ;
        </script>
    </body>
</html>
```

Existen herramientas mucho más sofisticadas y eficientes para realizar la comprobación de que el usuario de una aplicación web sea un humano o no. Hemos utilizado este ejemplo con el único fin de mostrar la invocación de una función desde otra función, además de mostrar cómo una función puede devolver un valor.

3. ARRAYS

Un array es un conjunto ordenado de valores relacionados. Cada uno de estos valores se denomina elemento y cada elemento tiene un índice que indica su posición numérica en el array.

En realidad, un array es un objeto más de JavaScript con una serie de funcionalidades adicionales respecto a los demás objetos.

DECLARACIÓN DE ARRAYS

Al igual que ocurre con las variables, es necesario declarar un array antes de poder usarlo. Su sintaxis es la siguiente:

```
var nombre_del_array = new Array();
```

La palabra reservada new se utiliza para crear una instancia de un tipo de objeto.. El constructor Array() tiene la función de construir el objeto en la memoria del ordenador.

Otra forma de declarar un array es especificando el número de elementos que contendrá:

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

Sin embargo, durante la ejecución de la aplicación podemos aumentar este número. Generalmente, se suele omitir el número de elementos que contendrá el array, ya que la mayor parte de las veces no conocemos exactamente el número de elementos que debemos gestionar. No obstante, es una buena práctica de programación especificar este número si se conoce previamente. De este modo obtendremos beneficios en la eficacia de la aplicación, ya que haremos un uso más eficiente de la memoria.

INICIALIZACIÓN DE ARRAYS

Una vez declarado el array, podemos comenzar con el proceso de inicialización del array con los elementos que contendrá. Existen diferentes formas de inicializar un array. La sintaxis general para hacerlo es la siguiente:

```
nombre_del_array[indice] = valor_del_elemento;
```

Podemos declarar un array llamado productos alimenticios e inicializarlo con los siguientes elementos:

```
var productos_alimenticios = new Array();
productos_alimenticios [0]='Pan';
productos_alimenticios [1]= 'Agua'
productos_alimenticios [2]= 'Lentejas'
```

Los arrays se pueden declarar e inicializar simultáneamente mediante la escritura de los elementos dentro del paréntesis del constructor. Debemos tener en cuenta que los elementos deben estar separados por comas:

```
var productos_alimenticios = new Array ('Pan', 'Agua', 'Lentejas' );
```

USO DE LOS ARRAYS MEDIANTE BUCLES

Si mezclamos las características de los bucles junto a las de los arrays, podremos apreciar las ventajas que se consiguen usando arrays cuando debemos trabajar con muchas variables.

Por ejemplo, si quisiéramos crear un array que contenga los códigos de determinados productos, podríamos realizarlo de la siguiente manera:

```
var codigos_productos = new Array();
for (var i=0; i<10;i++) {
        codigos_productos[i]="Codigo_producto " + i;
}
```

La inicialización de un array con un bucle funciona mejor en dos casos. El primero es cuando los valores de los elementos los podemos generar usando una expresión que cambia en cada iteración del bucle. El segundo es cuando necesitamos asignar el mismo valor a todos los elementos del array.

El problema principal cuando manipulamos muchas variables no es la fase de declaración, sino más bien la fase de trabajar con dichas variables. Con los bucles podemos acceder a cada uno de los elementos de un array y hacer lo que queramos con sus valores. Por ejemplo, podemos usar el siguiente código si quisiéramos imprimir en el documento de la página web, todos los códigos que contiene el array del ejemplo anterior.

```
for (var i=0; i<10; i++) {
        document.write (codigos_productos [i] + "<br>");
}
```

PROPIEDADES DE LOS ARRAYS

En el lenguaje JavaScript, un array es en realidad un objeto. Este objeto tiene una serie de propiedades y métodos al igual que los otros objetos que hemos estudiado hasta ahora en este libro.

length

Esta propiedad, devuelve el número de elementos que contiene el array. Esta propiedad es muy útil cuando utilizamos los arrays en los bucles. Su sintaxis es la siguiente:

nombre_del_array.lenght

Por ejemplo, anteriormente hemos utilizado un bucle for para imprimir por pantalla los códigos de determinados productos. En este bucle hemos empleado una expresión condicional para imprimir diez productos, pero es probable que el número de productos aumente con el paso del tiempo. Por este motivo, es mucho más útil definir ese bucle for de la siguiente manera:

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

prototype

Con esta propiedad podemos agregar nuevas propiedades y métodos al objeto Array. La sintaxis de prototype es la siguiente:

```
Array.prototype.nueva_propiedad = valor;
Array.prototype.nuevo_metodo = nombre_de_la_funcion;
```

Esta propiedad permite al usuario extender la definición de los arrays que se utilicen en alguna aplicación web en concreto. Si queremos agregar una nueva propiedad, debemos definir un valor concreto de esa propiedad, aunque luego se pueda cambiar. Si además queremos crear un nuevo método, debemos definir previamente una función JavaScript y escribir el nombre de dicha función en la creación del nuevo método. Si, por ejemplo, queremos crear una propiedad llamada dominio y establecerle el valor .com, debemos escribir el siguiente código:

```
Array.prototype.dominio = ".com";
```

De este modo, si creamos un array, este tendrá ya definida dicha propiedad. Sin embargo, podemos cambiar el valor de esta propiedad en el momento en que creemos nuevos arrays.

```
var paginas_comerciales = new Array();
Array.prototype.dominio = ".com";
var paginas_gubernamentales = new Array();
paginas_gubernamentales.dominio = ".gov";
document.write("Extensión de las páginas comerciales: " + paginas_comerciales.dominio);
document.write("<br>Extensión de las páginas gubernamentales: " +
paginas_gubernamentales.dominio);
```

MÉTODOS DE LOS ARRAYS

El objeto Array posee algunos métodos bastante útiles a la hora de manipular y gestionar todos los elementos presentes en los arrays. Estos métodos permiten unir dos arrays, ordenarlos, convertir sus valores o eliminar fácilmente algunos de sus elementos.

MÉTODOS	
push()	shift()

concat()	pop()
join()	slice()
reverse()	sort()
unshift()	splice()

push():
En el siguiente ejemplo declaramos un nuevo array llamado pizzas y se inicializa con tres elementos. Posteriormente, aplicamos el método push() para añadir dos nuevas pizzas. Este método devuelve el nuevo número de elementos presentes en el array:

```
<script>
	var pizzas = new Array("Carbonara", "Quattro Stagioni", "Diavola");
	var nuevo_numero_de_pizzas = pizzas.push("Margherita", "Boscaiola");
	document.write("Número de pizzas disponibles: " + nuevo_numero_de_pizzas + "<br />");
	document.write(pizzas);
</script>
```

concat():
El método concat() une los elementos de dos o más arrays en uno nuevo.

```
<script>
	var equipos_a = new Array("Real Madrid", "Barcelona", "Valencia");
	var equipos_b = new Array("Hércules", "Elche", "Valladolid");
	var equipos_copa_del_rey = equipos_a.concat(equipos_b);
	document.write("Equipos que juegan la copa: " + equipos_copa_del_rey);
</script>
```

join():
El método join() devuelve una cadena de texto con los elementos del array. Los elementos los podemos separar por una cadena que le pasemos como argumento del método. En el siguiente ejemplo utilizamos un guión como separador entre los elementos del array pizzas:

```
<script>
	var pizzas = new Array("Carbonara", "Quattro Stagioni", "Diavola");
	document.write(pizzas.join(" - "));
</script>
```

reverse():
Este método invierte el orden de los elementos sin crear un nuevo array

```
<script>
	var numeros = new Array(1,2,3,4,5,6,7,8,9,10);
	numeros.reverse();
	document.write(numeros);
</script>
```

unshift():

Con unshift() podemos añadir nuevos elementos y obtener la nueva longitud del array al igual que con el método push(). La diferencia es que push() añade los elementos al final del array, mientras que unshift() los añade al principio:

```
<script>
	var sedes_JJOO 0 new Array("Atenas", "Sydney", "Atlanta");
	var numero_sedes = sedes_JJOO.unshift("Pekin");
	document.write("Últimas " + numero_sedes + " sedes_olimpicas: " + sedes_JJOO);
</script>
```

shift():
El método shift () elimina el primer elemento de un array y devuelve dicho elemento. Si, por ejemplo, tenemos el mismo array de antes con una lista de pizzas y quisiéramos eliminar la primera de ellas, podemos utilizar el método shift() de la siguiente manera:

```
<script>
	var pizzas = new Array("Carbonara", "Quattro_Stagioni", "Diavola");
	var pizza_removida = pizzas.shift();
	document.write("Pizza eliminada de la lista: " + pizza_removida + "<br />");
	document.write("Nueva lista de pizzas: " + pizzas);
</script>
```

pop():
El método pop () tiene la misma funcionalidad que el método shift() con la diferencia de que en vez de eliminar y devolver el primer elemento de un array, elimina y devuelve el último.

```
<script>
	var premios = new Array("Coche", "1000 Euros", "Manual de JavaScript");
	var tercer_premio = premios.pop();
	document.write("El tercer premio es: " + tercer_premio + "<br />");
	document.write("Quedan los siguientes premios: " + premios);
</script>
```

slice():
Este método crea un nuevo array con un subconjunto de elementos pertenecientes a otro array. En el paréntesis especificamos el índice inicial y el numero de elementos del subconjunto que se almacenará en un nuevo array. El índice final es opcional y si no lo especificamos, tomará el subconjunto desde el índice inicial hasta el final del array. Además, el índice inicial puede ser un número negativo, con lo cual, la selección comenzaría desde el final del array.

```
<script>
	var numeros = new Array(1,2,3,4,5,6,7,8,9,10);
	var primeros_cinco = numeros.slice(0,5);
	var últimos_cuatro = numeros.slice(-4);
	document.write(primeros_cinco + "<br>");
	document.write(últimos_cuatro);
</script>
```

sort():
Este método ordena alfabéticamente un array:

```
<script>
        var apellidos = new Array("Pérez", "Guijarro", "Arias", "González");
        apellidos.sort();
        document.write(apellidos);
</script>
```

splice():
El método splice() es el más complejo del objeto Array. Con él es posible añadir o eliminar objetos de un array. Los dos primeros parámetros son obligatorios, mientras que el tercero es opcional. El primer parámetro especifica la posición en la cual añadiremos o eliminaremos los elementos. El segundo parámetro especifica cuántos elementos eliminaremos. El tercer y último parámetro son los nuevos elementos que añadiremos en el array. El siguiente ejemplo presenta un array con una serie de marcas de coches. Con el método splice() añadimos en la posición 2 el elemento llamado "Seat":

```
<script>
        var coches = new Array("Ferrari", "BMW", "Fiat");
        coches.splice(2,0,"Seat");
        document.write(coches);
</script>
```

ARRAYS MULTIDIMENSIONALES

Hemos visto arrays de una dimensión pero es posible crear arrays de más dimensiones. Para obtener esta nueva estructura de datos debemos definir un array que contiene a su vez nuevos arrays en cada una de sus posiciones.

Los arrays bidimensionales son los arrays multidimensionales más comunes en los lenguajes de programación. Podemos pensar en ellos como si fuesen una tabla con filas y columnas, con lo cual, necesitamos conocer dos índices para acceder a cada uno de sus elementos

[Fila, Columna]	0	1	2
0	elemento[0,0]	elemento[0,1]	elemento[0,2]
1	elemento[1,0]	elemento[1,1]	elemento[1,2]
2	elemento[2,0]	elemento[2,1]	elemento[2,2]

En JavaScript no existe un objeto llamado array multidimensional. Para poder utilizar este tipo de estructuras de datos debemos definir un array, en el que en cada una de sus posiciones debemos crear a su vez otro array.

En el siguiente ejemplo definimos un array bidimensional en el que por un lado tenemos el nombre de algunos países (España, Suiza y Portugal) y, por el otro, las cinco palabras más buscadas en Internet de cada país en el año 2011, según los datos ofrecidos por Google.

```
var palabras_espana = new Array('facebook', 'tuenti', 'youtube', 'hotmail', 'marca');
```

```
var palabras_suiza = new Array('facebook', 'youtube', 'hotmail', 'google', 'blick');
var palabras_portugal = new Array('facebook', 'youtube', 'hotmail', 'jogos', 'download');
var palabras_mas_buscadas = new Array (palabras_espana, palabras_suiza, palabras_portugal) ;
```

De este modo obtendríamos una estructura de datos similar a una tabla en la que las filas corresponden a los países y las columnas a las palabras más buscadas en Internet en cada uno de ellos.

[País, Ranking]	0	1	2	3	4
0 (España)	facebook	tuenti	Youtube	hotmail	marca
1 (Suiza)	facebook	youtube	hotmail	google	blick
2 (Portugal)	facebook	youtube	hotmail	Jogos	download

Al igual que con los arrays de una sola dimensión, podemos utilizar bucles para definir e inicializar arrays bidimensionales. Para ello, debemos utilizar lo que se denomina un bucle anidado, es decir, un bucle dentro de otro bucle. En el siguiente ejemplo utilizaremos un bucle for dentro de otro bucle del mismo tipo, con el fin de recorrer y recuperar el valor de todos los elementos del array bidimensional. Además, utilizaremos la técnica de generación de código HTML para construir una tabla en la cual estarán presentes todos los elementos del array.

```
document.write("<table border=\"1\">");
for (i=0; i<palabras_mas_buscadas.length; i++) {
        document.write("<tr>");
        document.write("<td><b>Estado " + i + "</b></td>");
        for (j=0; j<palabras_mas_buscadas[i].length; j++) {
                document.write("<td>" + palabras_mas_buscadas[i][j] + "</td>");
        }
        document.write("</tr>");
}
document.write("</table>");
```

4. OBJETOS DEFINIDOS POR EL USUARIO

Hemos visto los principales objetos predefinidos del lenguaje JavaScript. Sin embargo, es posible crear nuevos objetos definidos por el usuario. Estos objetos pueden tener sus propios métodos y propiedades.

La creación de nuevos objetos puede resultar útil en el desarrollo de aplicaciones avanzadas en las cuales no sean suficientes las características y funcionalidades proporcionadas por los objetos predefinidos de JavaScript.

A continuación mostraremos cómo crear nuevos objetos, además de cómo crear propiedades y métodos característicos de estos nuevos objetos.

DECLARACIÓN E INICIALIZACIÓN DE LOS OBJETOS

Un objeto es una entidad que posee unas propiedades que lo caracterizan y unos métodos que actúan sobre estas propiedades. Aparte de los objetos predefinidos de JavaScript, podemos definir nuevos objetos. Para declararlos debemos seguir la siguiente sintaxis:

```
function mi_objeto (valor_1, valor_2, valor_x) {
        this.propiedad_1=valor_1;
        this.propiedad_2=valor_2
        this.propiedad_x=valor_x
}

<script>
        function Coche (marca_in, modelo_in, anyo_in) {
                this.marca = marca_in;
                this.modelo = modelo_in;
                this.anyo=anyo_in;
        }
</script>
```

Una vez declarado el nuevo tipo de objeto denominado Coche, se pueden crear instancias del mismo a través de la palabra clave new. En el siguiente ejemplo creamos cuatro instancias del objeto Coche. Todas las instancias las guardamos en un array y, posteriormente, con un bucle for, accedemos e imprimimos por pantalla la marca, el modelo y el año de fabricación de cada una de las instancias de este objeto.

```
<script>
        var coches = new Array(4);
        coches[0]= new Coche ("Ferrari", "Scaglietti", "2010");
        coches[1]= new Coche ("BMW", "Z4", "2010");
        coches[2]= new Coche("Seat", "Toledo", "1999");
        coches[3]= new Coche ("Fiat", "500", "1995");
        for(i=0; i<coches.length; i++) {
                document.write("Marca: " + coches[i] .marca + " - Modelo: " + coches[i] .modelo + " -
                Año de fabricación: " + coches[i] .anyo + "<br>");
        }
</script>
```

DEFINICIÓN DE PROPIEDADES Y MÉTODOS

En el apartado anterior hemos visto cómo crear un nuevo objeto personalizado con algunas propiedades. En creación de las propiedades de los objetos utilizamos la palabra clave this seguida del nombre de la propiedad y el valor asignado a dicha propiedad. La palabra clave this hace referencia al objeto en el que utilizamos dicha palabra. Sin embargo, es posible agregar nuevas propiedades a las instancias de los objetos aunque estas no hayan sido declaradas en la definición del mismo.

Hemos visto que el objeto coche presentaba tres propiedades: marca, modelo y año de fabricación. Todas las instancias de este objeto tendrán estas tres propiedades. Sin embargo, es posible añadir otras propiedades a instancia del objeto. Por ejemplo, una vez creado una instancia del objeto Coche, podríamos añadir la siguiente propiedad:

```
function Coche (marca_in, modelo_in, anyo_in) {
```

```
        this.marca = marca_in;
        this.modelo = modelo_in;
        this.anyo = anyo_in;
}
var mi_coche = new coche ("Pegeout", "206cc", "2003");
mi_coche.color = "azul";
```

De este modo, la instancia llamada mi_coche tendrá la nueva propiedad que proporciona el color. Esta nueva propiedad solo afectará a esta instancia del objeto, pero no afectará a las demás.

Otra característica interesante de la definición de las propiedades de los objetos en JavaScript, es la posibilidad de establecer un objeto como propiedad de otro objeto. Por ejemplo, si tenemos un objeto que representa un concesionario de venta de coches, podríamos definir una nueva propiedad del objeto Coche que corresponde a los datos del concesionario donde se ha vendido. El objeto Concesionario podría tener a su vez algunas propiedades, como el código de su oficina, la ciudad donde se encuentra y el nombre del responsable.

```
function Concesionario (cod_oficina_in, ciudad_in, responsable_in) {
        this.cod_oficina = cod_oficina_in;
        this.ciudad = ciudad_in;
        this.responsable = responsable_in;
}

function Coche (marca_in, modelo_in, anyo_in, concesionario_in) {
        this.marca = marca_in;
        this.modelo = modelo_in;
        this.anyo = anyo_in;
        this.concesionario = concesionario_in;
}

var concesionario_atocha = new Concesionario ('281', 'Madrid', 'Pedro Bravo');
var mi_coche = new Coche ('Citroen', 'C4', '2010', concesionario_atocha);
```

Dentro de la definición de los objetos personalizados, podemos incluir funciones que acceden a las propiedades. Estas funciones se denominan los métodos del objeto. Para ello, debemos definir una función que realice las instrucciones que queramos ejecutar y, en la definición del objeto, en la misma sección donde definimos las propiedades, añadimos el nombre de la función a través de la palabra clave this.

Siguiendo el mismo ejemplo descrito anteriormente, podríamos crear una función que escriba en pantalla todos los datos del coche y añadir dicha función en la definición del objeto:

```
function imprimeDatos(){
        document.write("<br>Marca: " + this.marca);
        document.write("<br>Modelo: " + this.modelo);
        document.write("<br>Año: " + this.anyo);
}
```

```
function Coche (nombre_in, modelo_in, anyo_in) {
        this.marca = nombre_in;
        this.modelo = modelo_in;
        this.anyo = anyo_in;
        this.imprimeDatos = imprimeDatos;
}

var mi_coche = new Coche ("Seat", "Toledo", "1999");
mi_coche.imprimeDatos();
```