

4

Programación con funciones, arrays y objetos definidos por el usuario

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las principales funciones predefinidas del lenguaje JavaScript.
- ✓ Poder crear funciones personalizadas para realizar tareas específicas que las funciones predefinidas no logran hacer.
- ✓ Comprender el objeto Array de JavaScript y familiarizarse con sus propiedades y métodos.
- ✓ Crear objetos personalizados diferentes a los objetos predefinidos del lenguaje.
- ✓ Definir propiedades y métodos de los objetos personalizados.

Los programas escritos en cualquier lenguaje de programación suelen recurrir constantemente a partes de código que realizan una serie de tareas. Por ejemplo, comprobar el tipo de datos que ingresa un usuario por pantalla es una tarea bastante común. Generalmente, debemos comprobar si estos datos son numéricos, cadenas de texto o valores booleanos. Para realizar estas comprobaciones debemos definir una serie de instrucciones que en algunos casos ocupan bastantes líneas de código. Cuando esta serie de instrucciones se repiten una y otra vez, el código fuente de la aplicación se complica principalmente por dos motivos:

- ✓ El número de líneas de código aumenta considerablemente, ya que muchas de éstas estarán repetidas.
- ✓ El mantenimiento se dificulta debido a que si queremos modificar una instrucción repetida, deberemos hacerlo tantas veces como aparezca en la aplicación, lo que se convierte en un trabajo tedioso y propenso a errores.

Por suerte, en JavaScript, al igual que en todos los lenguajes de programación, no necesitamos escribir una y otra vez todas las instrucciones que realicen una determinada tarea. Para ello, podemos definir estas instrucciones en una parte de código **delimitado** e **invocarlo** cuando nos sea útil. Esta serie de instrucciones englobadas en un mismo proceso se denominan **funciones**. En este capítulo presentamos las principales funciones predefinidas por JavaScript, además de cómo el usuario puede crear y utilizar sus propias funciones.

Otro problema bastante recurrente es la gestión eficaz de los datos. Al inicio del desarrollo de una aplicación web, solemos utilizar un número de variables que puede ir creciendo con el paso del tiempo y que cada vez dificulta más el mantenimiento y la eficacia de la aplicación. JavaScript cuenta con los objetos llamados **arrays**, los cuales permiten una gestión más eficaz para las aplicaciones que presentan una gran cantidad de datos a manipular. Los **arrays** proporcionan un conjunto de propiedades y métodos que permiten realizar diferentes tareas sobre una gran cantidad de información.

La última sección de este capítulo presenta la creación de nuevos objetos definidos por el usuario. JavaScript proporciona una serie de objetos predefinidos. Cada uno de estos objetos tiene una serie de propiedades y métodos que realizan diferentes tareas. Sin embargo, en algunas aplicaciones avanzadas, el usuario se encontrará con la necesidad de crear sus propios objetos y que dichos objetos tengan sus propios métodos y propiedades.

4.1 FUNCIONES PREDEFINIDAS DEL LENGUAJE

JavaScript cuenta con una serie de funciones predefinidas, es decir, funciones que ya están integradas en el lenguaje. Podemos utilizar estas funciones en cualquier momento sin tener que declararlas previamente y sin tener que conocer todas las instrucciones que realiza. Simplemente debemos conocer el nombre de la función y el resultado final que obtenemos al utilizarla.

En la Tabla 4.1 vemos algunas de las funciones predefinidas más utilizadas por los programadores de JavaScript. 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.

Tabla 4.1 Funciones predefinidas de JavaScript

Función predefinida	Descripción
escape()	Recibe como argumento una cadena de caracteres y devuelve esa misma cadena sustituida con su codificación en ASCII.
eval()	Convierte una cadena que pasamos como argumento en código JavaScript ejecutable.
isFinite()	Verifica si el número que pasamos como argumento es o no un número finito.
isNaN()	Comprueba si el valor que pasamos como argumento es un de tipo numérico.
Number()	Convierte el objeto pasado como argumento en un número que represente el valor de dicho objeto.
String()	Convierte el objeto pasado como argumento en una cadena que represente el valor de dicho objeto.
parseInt()	Convierte la cadena que pasamos como argumento en un valor numérico de tipo entero.
parseFloat()	Convierte la cadena que pasamos como argumento en un valor numérico de tipo flotante.

A continuación presentamos una descripción y una serie de ejemplos de cada una de estas 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 ISO³. Si por ejemplo quisieramos saber la codificación del carácter (?), podríamos utilizar el siguiente ejemplo y verificar que obtendríamos la codificación %3F.

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

Un `unescape()` es la función opuesta a `escape()`. Es decir, que esta función decodifica los caracteres que estén codificados.

³ http://en.wikipedia.org/wiki/ISO/IEC_8859-1

- **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 type="text/javascript">
    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 $\pm 1.7976931348623157 \times 10^{308}$. 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. El siguiente ejemplo evalúa si el dato ingresado por el usuario es de tipo numérico y en base a eso, utilizando una sentencia condicional, ejecutamos una instrucción u otra.

```
<script type="text/javascript">
    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. La Figura 4.1 muestra el resultado de este ejemplo, además podemos ver el formato predefinido que usa JavaScript para indicar una fecha.

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

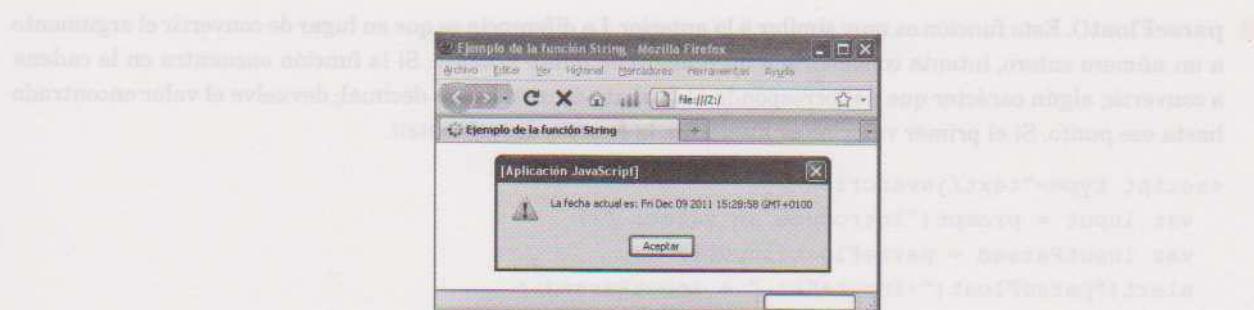


Figura 4.1. Ejemplo de la función *String()*

- **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 (10). La base puede ser por ejemplo binaria (2), octal(8), decimal (10) o hexadecimal (16). 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*. En la Figura 4.2 vemos un ejemplo en el cual el usuario introduce la cadena 28.5º y la aplicación abre una ventana emergente con el resultado después de haber aplicado la función *parseInt()*.

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



Figura 4.2. Ejemplo de la función *parseInt()*

- **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. Si la función encuentra en la cadena a convertir, algún carácter que no corresponda al formato de un número decimal, 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 = parseFloat(input);
    alert("parseFloat(\"+input+\") : " + inputParsed);
</script>
```

ACTIVIDADES 4.1

- Cree un *script* que utilice el método `document.write()` para mostrar por pantalla la codificación de todas las vocales con tilde, tal y como vemos en la Figura 4.3. Recuerde que para obtener dicha codificación debe usar la función `escape()`.

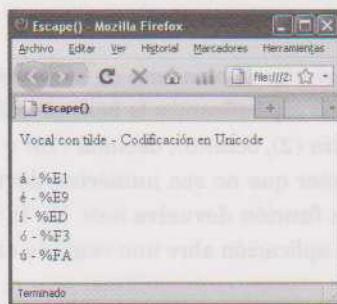


Figura 4.3. Codificación de las vocales con tilde

4.2 FUNCIONES DEL USUARIO

En el apartado anterior hemos visto que JavaScript contiene una serie de funciones integradas en el lenguaje. Estas funciones podemos utilizarlas en cualquier momento para realizar diferentes tareas específicas. Sin embargo, podemos crear nuevas funciones personalizadas 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 al 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.

4.2.1 DEFINICIÓN DE FUNCIONES

El primer paso en el uso de las funciones del usuario es su definición. Hasta ahora, el código JavaScript lo escribíamos dentro de las etiquetas HTML `<body>` y `</body>`, pero 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 definición de una función consta de cinco partes principales: la palabra clave `function`, el nombre de la función, los argumentos utilizados, el grupo de instrucciones y la palabra clave `return`. Los argumentos y la palabra clave `return` son dos partes opcionales. 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 clave del lenguaje.
- 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. No todas las funciones requieren argumentos. 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. Al igual que en el ejemplo utilizado en la explicación de los argumentos, 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`. En algunos casos realizamos tareas que no la requieren.

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 la función anterior hemos utilizado todas las partes descritas anteriormente con excepción de la palabra clave `return`, ya que la función consiste en mostrar un aviso en el navegador con el precio actualizado del producto. Inmediatamente después de la palabra clave `function`, hemos usado el nombre `aplicar_IVA`, con lo cual no podremos utilizar dicho nombre en ninguna otra función de la aplicación que estemos desarrollando. Entre el paréntesis hemos escrito un argumento llamado `valorProducto`, lo que significa que la función necesita un valor para poder ejecutar el grupo de instrucciones definidas en el cuerpo de la función. La primera instrucción define la variable llamada `productoConIVA` y la inicializa al valor pasado como argumento multiplicado por 1,04, es decir, con el valor del argumento más el 4% del mismo valor. La segunda instrucción muestra por pantalla el resultado del cálculo de la primera instrucción.

En el grupo de instrucciones de una función podemos declarar nuevas variables, tal y como hemos visto en el ejemplo anterior. 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 como, por ejemplo, las sentencias condicionales, se denominan variables globales. Estas últimas podemos utilizarlas en cualquier parte de la aplicación, incluso dentro de una función.

4.2.2 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. Si no mantenemos ese orden obtendremos resultados inesperados. Por ejemplo, podemos tener una función que aplica el IVA a una serie de productos y como argumentos presenta el valor del producto y el porcentaje del IVA:

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

En este caso, si invitamos la función, es importante tener en cuenta el orden de los argumentos. Si por ejemplo, queremos utilizar la función para aplicar el IVA del 18% a un producto que cuesta 300 euros, no es lo mismo invocar `aplicar_IVA(300, 1.18)` que `aplicar_IVA(1.18, 300)`. El resultado no sería el esperado, ya que la función espera que el primer argumento sea el valor del producto y el segundo argumento sea el IVA que se le debe aplicar. En la Figura 4.4 vemos el ejemplo anterior, en el cual existe una función que recibe dos argumentos proporcionados por el usuario y al final, realiza un cálculo y lo muestra en una ventana emergente:

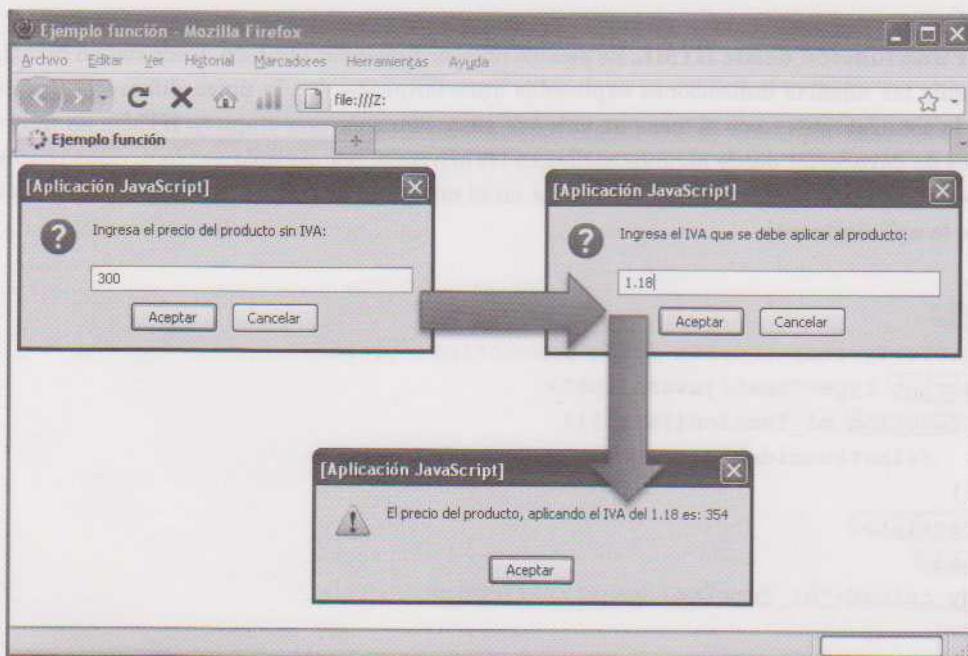


Figura 4.4. Ejemplo de función que recibe dos parámetros del usuario

Existen diferentes formas de llamar o invocar una función:

- **Invocar una función desde JavaScript.** La definición de la función debe estar dentro de las etiquetas HTML `<head>` y `</head>`. La llamada a la función estará dentro de las etiquetas `<script>` y `</script>` que a su vez estarán contenidas en las etiquetas HTML `<body>` y `</body>`. Entre estas etiquetas de la aplicación debemos utilizar el nombre de la función seguido del paréntesis que contiene o no los argumentos necesarios para ejecutar la función.

```
<html>
  <head>
    <title>Invocar función desde JavaScript</title>
    <script type="text/javascript">
      function mi_funcion([args]){
        //instrucciones
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
      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.

```
<html>
  <head>
    <title>Invocar función desde JavaScript</title>
    <script type="text/javascript">
      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. En la Figura 4.5 vemos el resultado de la ventana emergente que muestra un aviso al usuario especificando que no ha introducido un valor correcto. Con lo cual no puede ver el contenido de la página web. 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.

```
<html>
<head>
<title>Retorno de valor en una función</title>
<script type="text/javascript">
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 type="text/javascript">
    ComprobarHumano();
</script>
</body>
</html>
```

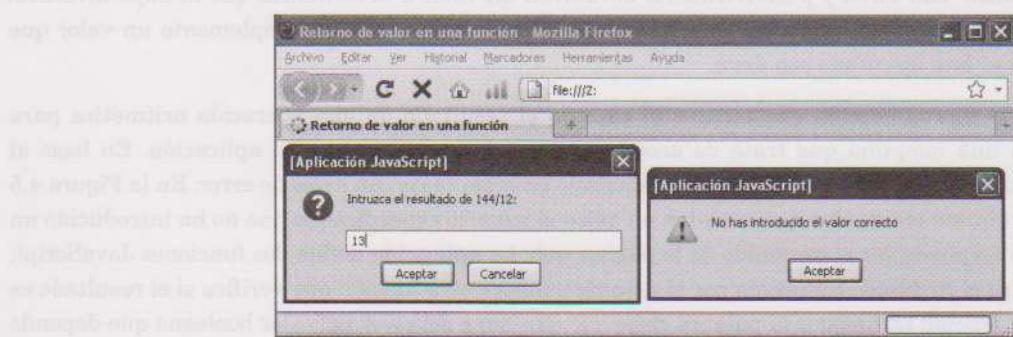


Figura 4.5. Función que emplea el retorno de un valor

“

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.

ACTIVIDADES 4.2



- Escriba una función que recibe como parámetro un número entero y devuelve como resultado una cadena que indica si el número es par o impar. Muestre el resultado por pantalla a través del método `alert()`. Recuerde que puede ser útil el uso del operador módulo `%`.

4.3 ARRAYS

La mayor parte de las aplicaciones web están diseñadas para gestionar un número elevado de datos. Si tenemos por ejemplo una aplicación que gestiona una tienda de productos alimenticios, es normal que manipulemos decenas de variables para identificar a cada producto. Si quisieramos definir los nombres de cada uno de ellos, suponiendo que tenemos 180 productos, podríamos utilizar un código similar al siguiente:

```
var producto1 = "Pan";
var producto2 = "Agua";
var producto3 = "Lentejas";
var producto4 = "Naranjas";
var producto5 = "Cereales";
...
var producto180 = "Salsa agridulce";
```

Si posteriormente quisieramos mostrar los nombres de estos productos en una página web, podríamos utilizar el siguiente código:

```
document.write(producto1);
document.write(producto2);
document.write(producto3);
document.write(producto4);
document.write(producto5);
...
document.write(producto180);
```

La aplicación web escrita de este modo funcionaría correctamente, pero obviamente sería una tarea compleja, repetitiva y propensa a errores. No estaríamos llevando a cabo una programación eficaz, con lo cual la ejecución de la aplicación sería más lenta y difícil de gestionar.

Para gestionar este tipo de escenarios se utilizan los *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*. Además de los valores y los índices de estos valores, un *array* debe tener un nombre.



En realidad, un *array* es un objeto más de JavaScript con una serie de funcionalidades adicionales respecto a los demás objetos. Estas funcionalidades las estudiaremos a lo largo de este apartado del libro.

Los productos alimenticios del ejemplo anterior podrían estar almacenados en un *array*, que podemos verlo como una tabla que presenta una serie de celdas que indican el índice y el nombre de cada producto. Si definimos el nombre de esta tabla como `Productos_Alimenticios`, obtendríamos lo siguiente:

Tabla 4.2 Productos_Alimenticios

Índice	Contenido
0	Pan
1	Agua
2	Lentejas
3	Naranja
4	Cereales
...	...
179	Salsa agridulce

De este modo se podría acceder al nombre de cada producto especificando solamente el nombre de la tabla y el índice. Debemos notar que el primer índice de los *arrays* es siempre cero y no uno, tal y como la mayor parte de programadores novatos suele pensar.

A continuación mostramos cómo declarar e inicializar los *arrays*, además de enseñar las principales operaciones que podemos realizar con estos.

4.3.1 DECLARACIÓN DE ARRAYS

Al igual que ocurre con las variables, es necesario declarar un *array* antes de poder usarlo. La declaración de un *array* consta de seis partes: la palabra clave `var`, el nombre del *array*, el operador de asignación, la palabra clave para la creación de objetos (`new`), el constructor `Array` y el paréntesis final. Su sintaxis es la siguiente:

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

En el capítulo anterior hemos visto que la palabra clave `new` se utiliza para crear una instancia de un tipo de objeto. En este caso, estamos creando una instancia del objeto `Array`. 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. Este número debemos introducirlo entre el paréntesis del constructor. Si por ejemplo, conocemos a priori que es necesario un *array* con diez elementos, podríamos utilizar el siguiente código:

```
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.

4.3.2 INICIALIZACIÓN DE ARRAYS

Una vez declarado el *array*, podemos comenzar con el proceso de inicialización o popularización 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;
```

Según los datos del ejemplo de los productos alimenticios, 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');
```

Todos los elementos deben ser del mismo tipo de datos. En este caso tenemos un conjunto de cadenas, pero podemos utilizar números, valores booleanos, objetos o incluso otros *arrays*. JavaScript no permite mezclar diferentes tipos de datos en un mismo *array*.

4.3.3 USO DE LOS ARRAYS MEDIANTE BUCLES

Hasta ahora hemos utilizado ejemplos que probablemente no dejen claras algunas de las principales ventajas de los *arrays* respecto al uso de las variables convencionales. Si mezclamos las características de los bucles junto a las de los *arrays*, podremos apreciar las ventajas que se consiguen cuando debemos trabajar con muchas variables.

Por ejemplo, si quisieramos 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 sentencia que se encuentra dentro del bucle `for` emplea el valor de la variable `i` para acceder a un índice del `array` y asignarle una cadena con un código incremental del producto.



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 quisieramos imprimir en el documento de la página web, todos los códigos que contiene el `array` del ejemplo anterior, tal y como vemos en la Figura 4.6.

```
document.write(codigos_productos[0] + "<br>");  
document.write(codigos_productos[1] + "<br>");  
document.write(codigos_productos[2] + "<br>");  
document.write(codigos_productos[3] + "<br>");  
document.write(codigos_productos[4] + "<br>");  
document.write(codigos_productos[5] + "<br>");  
document.write(codigos_productos[6] + "<br>");  
document.write(codigos_productos[7] + "<br>");  
document.write(codigos_productos[8] + "<br>");  
document.write(codigos_productos[9] + "<br>");
```

Sin embargo, utilizando un bucle `for`, podemos evitar la escritura de todas estas líneas de código y escribir unas instrucciones mucho más limpias y eficientes:

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

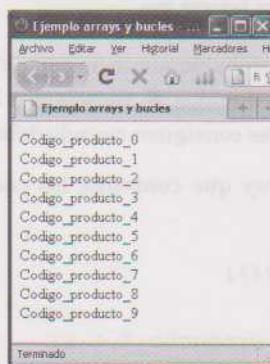


Figura 4.6. Uso de arrays en bucles

4.3.4 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.

El objeto *Array* tiene dos propiedades: *length* y *prototype*. La primera propiedad, *length*, 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.length
```

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<códigos_productos.length; i++) {  
    document.write(códigos_productos[i] + "<br>");  
}
```

La segunda propiedad del objeto *Array*, *prototype*, es mucho más compleja que la anterior. 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_método = 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 creamos nuevos *arrays*.

```
var páginas_comerciales = new Array();  
Array.prototype.dominio = ".com";  
  
var páginas_gubernamentales = new Array();  
páginas_gubernamentales.dominio = ".gov"  
  
document.write("Extensión de las páginas comerciales: " +  
    páginas_comerciales.dominio);  
document.write("Extensión de las páginas gubernamentales:  
    " + páginas_gubernamentales.dominio);  
  
    .gov
```

De este modo, el valor de la propiedad `dominio` en el primer `array` será `.com`, mientras que el valor de la misma propiedad en el segundo `array` será `.gov`.



La definición de nuevos métodos y propiedades en los objetos la abordaremos más a fondo en la sección relacionada con la creación de objetos.

4.3.5 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. Los principales métodos del objeto `Array` los vemos en la Tabla 4.3.

Tabla 4.3 Métodos del objeto `Array`

Método	Descripción	Sintaxis
<code>push()</code>	Añade nuevos elementos al <code>array</code> y devuelve la nueva longitud del <code>array</code> .	<code>nombre_array.push(valor1, valor2, ...)</code>
<code>concat()</code>	Selecciona un <code>array</code> y lo concatena con otros elementos en un nuevo <code>array</code> .	<code>nombre_array.concat(valor1, valor2, ...)</code>
<code>join()</code>	Concatena los elementos de un <code>array</code> en una sola cadena separada por un carácter opcional.	<code>nombre_array.join([separador])</code>
<code>reverse()</code>	Invierte el orden de los elementos de un <code>array</code> .	<code>nombre_array.reverse()</code>
<code>unshift()</code>	Añade nuevos elementos al inicio de un <code>array</code> y devuelve el número de elementos del nuevo <code>array</code> modificado.	<code>nombre_array.unshift(valor1, valor2, ...)</code>
<code>shift()</code>	Elimina el primer elemento de un <code>array</code> .	<code>nombre_array.shift()</code>
<code>pop()</code>	Elimina el último elemento de un <code>array</code> .	<code>nombre_array.pop()</code>
<code>slice()</code>	Devuelve un nuevo <code>array</code> con un subconjunto de los elementos del <code>array</code> que ha usado el método.	<code>nombre_array.slice(indice_inicio, [indice_final])</code>
<code>sort()</code>	Ordena alfabéticamente los elementos de un <code>array</code> . Podemos definir una nueva función para ordenarlos con otro criterio.	<code>nombre_array.sort([función])</code>
<code>splice()</code>	Elimina, sustituye o añade elementos del <code>array</code> dependiendo de los argumentos del método.	<code>nombre_array.splice(inicio, [numero_elem_a_borrar], [valor1, valor2, ...])</code>

A continuación mostramos algunos ejemplos de cada uno de los métodos del objeto Array.

- **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 type="text/javascript">
    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. En este ejemplo tenemos dos arrays que contienen algunos equipos de primera y de segunda división del fútbol español. Posteriormente, concatenamos estos equipos de fútbol en un nuevo array llamado equipos_copa_del_rey e imprimimos los valores de los elementos.

```
<script type="text/javascript">
    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 type="text/javascript">
    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. En el siguiente ejemplo tenemos un array con los números ordenados del 1 al 10. Si se aplica el método reverse() y escribimos el resultado en el documento, veremos que ahora van del 10 al 1.

```
<script type="text/javascript">
    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. En el siguiente ejemplo tenemos algunas de las últimas sedes de los juegos olímpicos. Si quisieramos agregar una sede más al inicio del *array*, el método `unshift()` es ideal para esta tarea:

```
<script type="text/javascript">
    var sedes_JJOO = new Array("Atenas", "Sydney",
        "Atlanta");
    var numero_sedes = sedes_JJOO.unshift("Pekin");
    document.write("Últimas " + numero_sedes + " sedes
        olímpicas: " + 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 quisieramos eliminar la primera de ellas, podemos utilizar el método `shift()` de la siguiente manera:

```
<script type="text/javascript">
    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. En el siguiente ejemplo tenemos un *array* con tres premios. El método `pop()` permite eliminar y devolver el último premio del *array*, en este caso el tercer premio.

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

- **sort()**. Este método ordena alfabéticamente un *array*. Sin embargo, podemos crear nuevos criterios de ordenación en una función y pasarle el nombre de la función como parámetro del método. En el siguiente ejemplo tenemos una lista de apellidos, la cual ordenamos alfabéticamente e imprimimos por pantalla:

```
<script type="text/javascript">
    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 type="text/javascript">
    var coches = new Array("Ferrari", "BMW", "Fiat");
    coches.splice(2,0,"Seat"); 1º) posición en la cual añadiremos o
    document.write(coches); 2º) cuántos eliminaremos
    </script> 3º) Elementos nuevos a añadir
```

4.3.6 ARRAYS MULTIDIMENSIONALES

Gracias a los apartados anteriores hemos comprobado la importancia que tienen los *arrays* en JavaScript, al igual que en la mayor parte de los lenguajes de programación. Hasta el momento hemos estudiado *arrays* de una sola dimensión, es decir, una estructura de datos en la que es necesario conocer solo un índice para acceder a cada elemento. Esta estructura es análoga a una lista de elementos. Sin embargo, 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, tal y como podemos ver en la Tabla 4.4.

Tabla 4.4 Array bidimensional visto como tabla

[Filas, Columnas]	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 Google⁴.

```
var palabras_espana = new Array(5);
palabras_espana[0] = 'facebook';
palabras_espana[1] = 'tuenti';
palabras_espana[2] = 'youtube';
palabras_espana[3] = 'hotmail';
palabras_espana[4] = 'marca';

var palabras_suiza = new Array(5);
palabras_suiza[0] = 'facebook';
palabras_suiza[1] = 'youtube';
palabras_suiza[2] = 'hotmail';
palabras_suiza[3] = 'google';
palabras_suiza[4] = 'blick';

var palabras_portugal = new Array(5);
palabras_portugal[0] = 'facebook';
palabras_portugal[1] = 'youtube';
palabras_portugal[2] = 'hotmail';
palabras_portugal[3] = 'jogos';
palabras_portugal[4] = 'download';

var palabras_mas_buscadas = new Array(3)
palabras_mas_buscadas[0] = palabras_espana;
palabras_mas_buscadas[1] = palabras_suiza;
palabras_mas_buscadas[2] = palabras_portugal;
```

⁴ <http://www.google.com/zeitgeist/>

```

        document.write("<td>" + palabras_mas_buscadas [i][j] + "</td>")
    }
}
document.write("</table>")

```

En este último ejemplo, además del uso de un bucle anidado para acceder a cada elemento del *array* bidimensional, hemos utilizado la generación de las etiquetas HTML destinadas a la creación de tablas (*<table>*) con sus respectivas líneas (*<tr>*) y celdas (*<td>*).

ACTIVIDADES 4.3



- A partir del siguiente *array*: var palabras = new Array('botella', 'zeta', 'androide', 'minuto'); ordene alfabéticamente sus elementos utilizando el método designado para ello y muestra el resultado por pantalla.
- Cree un *script* que tome una serie de palabras ingresadas por el usuario y almacene esas palabras en un *array*. Posteriormente, manipule ese *array* para mostrar en una nueva ventana los siguientes datos:
 - a. La primera palabra ingresada por el usuario.
 - b. La última palabra ingresada por el usuario.
 - c. El número de palabras presentes en el *array*.
 - d. Todas las palabras ordenadas alfabéticamente.

Podemos ver un ejemplo del resultado de esta actividad en la Figura 4.7.

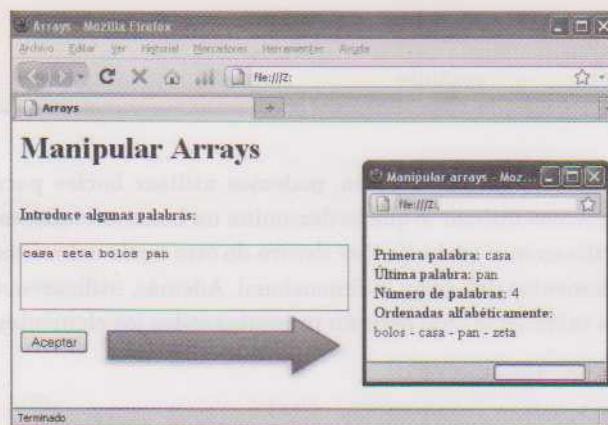


Figura 4.7. Manipulación de arrays

4.4 OBJETOS DEFINIDOS POR EL USUARIO

En el capítulo anterior de este libro, presentamos los principales objetos predefinidos del lenguaje JavaScript. Los objetos están organizados de una manera jerárquica en la que el objeto `Window` se encuentra en el nivel más alto, seguido por otros objetos importantes como el objeto `Document` o el objeto `Frame`. Los *arrays*, los cuales hemos estudiado ampliamente en la sección anterior, también son objetos de JavaScript. Cada uno de estos objetos tiene una serie de propiedades y métodos muy útiles para desarrollar aplicaciones web. 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.

4.4.1 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;  
}
```

Al igual que con la creación de nuevas funciones, en la creación de los objetos empleamos la palabra clave `function`. Además de esta palabra clave, podemos darle un nombre al nuevo tipo de objeto que estamos creando y unos valores iniciales que corresponderán a cada una de las propiedades que definamos. La explicación de la creación de propiedades y métodos de los objetos la abordaremos en el siguiente apartado. Para una mejor comprensión, a continuación realizaremos un ejemplo de creación de un nuevo objeto que representa un coche. Este nuevo tipo de objeto presenta tres propiedades que corresponden a la marca y el modelo del coche, además de su año de fabricación:

```
<script type="text/javascript">  
    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 type="text/javascript">
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 fabricaci&on: " + coches[i].año + "<br>");
}
</script>
```

En la Figura 4.8 vemos el resultado de la ejecución del ejemplo anterior. En este caso mostramos cada uno de los valores de la propiedad marca, modelo y año de fabricación.

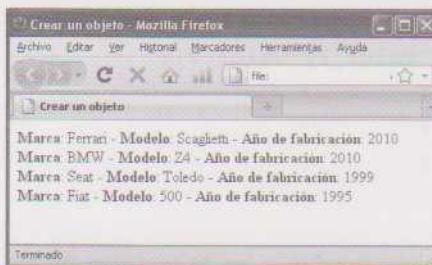


Figura 4.8. Ejemplo de creación de objetos personalizados

4.4.2 DEFINICIÓN DE PROPIEDADES Y MÉTODOS

En el apartado anterior hemos visto cómo crear un nuevo objeto personalizado con algunas propiedades. En la 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 cada

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("Peugeot", "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.anho);
}

function Coche(nombre_in, modelo_in, anho_in){
    this.marca = nombre_in;
    this.modelo = modelo_in;
    this.anho = anho_in;
    this.imprimeDatos = imprimeDatos;
}

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

ACTIVIDADES 4.4



- Utilice el código empleado para la generación de las cuatro instancias del objeto Coche y modifíquelo para que los valores de cada una de sus propiedades se impriman en una tabla HTML (<table>). Utilice la generación de código HTML desde código JavaScript. Cada instancia del objeto debe ocupar una línea (<tr>) y el valor de cada propiedad debe ocupar una celda (<td>) de dicha línea. El resultado debe ser similar al de la Figura 4.9.

Marca	Modelo	Año de fabricación
Ferrari	Scaglietti	2010
BMW	Z4	2010
Seat	Toledo	1999
Fiat	500	1995

Figura 4.9. Valores de las propiedades del objeto Coche en una tabla HTML



RESUMEN DEL CAPÍTULO

En este capítulo hemos presentado las principales funciones predefinidas de JavaScript, con una descripción detallada y un ejemplo de cada una de ellas. Cada una de estas funciones realiza una tarea específica, la cual se puede invocar en cualquier parte del código a través del nombre de la función.

A pesar de que JavaScript proporciona diferentes funciones predefinidas, en muchas ocasiones, sobre todo en el desarrollo de aplicaciones web avanzadas, es necesario crear nuevas funciones que realicen tareas personalizadas por el usuario. Hemos explicado cómo definir y cómo invocar estas funciones personalizadas.

A continuación, hemos presentado un nuevo tipo de objeto llamado `Array`. Este tipo de datos es un contenedor de elementos relacionados entre sí. El objeto `Array` presenta diferentes ventajas a la hora de manipular una gran cantidad de datos relacionados. En este capítulo hemos abordado la declaración, inicialización y uso de los *arrays*, además, hemos presentado los diferentes métodos y propiedades de este objeto.

Por último, hemos explicado la forma de crear objetos definidos por el usuario. Tal y como sucede con las funciones, JavaScript proporciona una serie de objetos predefinidos pero, en ciertas ocasiones, es bastante útil crear objetos personalizados con sus respectivos métodos y propiedades.



EJERCICIOS PROPUESTOS

- 1. Cree un *script* que solicite un valor numérico al usuario en base octal (8) y posteriormente muestre su equivalente en base decimal (10). Utilice el método `alert()` para mostrar el resultado en una ventana emergente.
- 2. Cree una aplicación que solicite dos números enteros al usuario. Estos números serán los parámetros de la función que se debe definir y que devolverá la suma de dichos números. Utilice el método `alert()` para mostrar el resultado por pantalla. Es necesario que recuerde el uso del método `parseInt()` para controlar los datos que ingresa el usuario.
- 3. Cree un *array* llamado `meses`. Este *array* deberá almacenar el nombre de los doce meses del año. Muestre por pantalla el nombre de cada uno de ellos utilizando un bucle `for`.
- 4. Cree un *script* que defina un objeto llamado `Producto_alimenticio`. Este objeto debe presentar las propiedades `código`, `nombre` y `precio`, además del método `imprimeDatos`, el cual escribe por pantalla los valores de sus propiedades. Posteriormente, cree tres instancias de este objeto y guárdelas en un *array*. Con la ayuda del bucle `for`, utilice el método `imprimeDatos` para mostrar por pantalla los valores de los tres objetos instanciados.