CF

GRADO SUPERIOR

CICLOS FORMATIVOS

R.D. 1538/2006

Desarrollo Web en Entorno Cliente

JUAN MANUEL VARA MESA
MARCOS LÓPEZ SANZ
DAVID GRANADA
EMANUEL IRRAZÁBAL
JESÚS JAVIER JIMÉNEZ HERNÁNDEZ
JÉNIFER VERDE MARÍN

Ra-Ma®

www.ra-ma.es/cf

Índice

Introducción

Históricamente, la aproximación más utilizada en el desarrollo web ha sido trasladar toda la interacción entre el usuario y la aplicación al servidor web. De esta forma el usuario utiliza su navegador web para enviar sus peticiones a un servidor que las procesa y le devuelve el resultado que se mostraba en el navegador. No obstante, las mejoras hardware han hecho que esta tendencia varíe. Los usuarios finales disponen de máquinas con mayor capacidad de procesamiento, por tanto, ya no es necesario que sea el servidor quien se ocupe de la mayor carga de procesamiento. Así, la tendencia generalizada pasa por desarrollar aplicaciones web que trasladan la mayor carga de procesamiento al navegador web del cliente, con lo que reducen la necesidad constante de intercambiar información con el servidor y, por tanto, eliminan el retardo derivado de la latencia de la Red.

Por todo ello, y ante el auge y explosión de Internet como herramienta de comunicación y trabajo y la creciente demanda de profesionales cualificados del sector, este libro surge con el propósito de acercar al lector los aspectos más importantes que encierra el desarrollo web en el entorno cliente. Con tal propósito, puede servir de apoyo también para estudiantes de los Ciclos Formativos de Grado Superior de Desarrollo de Aplicaciones Web y las Ingenierías Técnicas.

Para todo aquel que use este libro en el entorno de la enseñanza (ciclos formativos o universidad) se ofrecen varias posibilidades: utilizar los conocimientos aquí expuestos para inculcar aspectos genéricos del desarrollo web o simplemente centrarse en preparar a fondo alguno de ellos. La extensión de los contenidos aquí incluidos hace imposible su desarrollo completo en la mayoría de los casos.

Ra-Ma pone a disposición de los profesores una página web para el desarrollo del tema que incluye las soluciones a los ejercicios expuestos en el texto, un glosario, bibliografía y diversos recursos para suplementar el aprendizaje de los conocimientos de este módulo. Puede solicitar el medio de acceso a dichos recursos a *editorial@ra-ma.com*, acreditándose como docente y siempre que el libro sea utilizado como texto base para impartir las clases.

1 Selección de arquitecturas y herramientas de programación

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las diferentes alternativas existentes para la navegación web en función de las diferentes tecnologías web que se ejecutan en un cliente.
- ✓ Reconocer las capacidades de la ejecución de código en el lado del cliente de acuerdo a los componentes arquitectónicos de un navegador web.
- ✓ Identificar los principales lenguajes y tecnologías de programación en entorno cliente.
- ✓ Conocer las técnicas de integración del código con documentos HTML

En este primer capítulo presentamos los conceptos necesarios para comprender el contexto de ejecución de páginas web como forma de acceder a recursos de aplicaciones y sistemas de información web. Para ello, introducimos las características principales de los navegadores más comunes utilizados hoy en día. Describiremos, así mismo, los diferentes lenguajes y tecnologías de programación, del lado del cliente, aplicables en este tipo de entornos. Por último, se hace un recorrido introductorio por algunas de las técnicas de integración que el desarrollador tiene a su alcance para intercalar código con las etiquetas HTML.

1.1 EVOLUCIÓN Y CARACTERÍSTICAS DE LOS NAVEGADORES WEB

La *World Wide Web* (o "la Web", como se conoce comúnmente) representa un universo de información accesible globalmente a través de la red Internet. Está formada por un conjunto de recursos interconectados que conforman el conocimiento humano actual. El funcionamiento de la Web es posible debido a la coexistencia de una serie de componentes software y hardware. Estos elementos abarcan desde los componentes físicos de Internet (*hubs*, repetidores, puentes, pasarelas, encaminadores, etc.) y los protocolos de comunicaciones (TCP, IP, HTTP, FTP, SMTP, etc.), hasta la utilización del sistema de nombres de dominio (DNS) para la búsqueda y recuperación de recursos o la utilización de software específico para proveer y consumir dichos recursos.

En este contexto, el desarrollo en entornos web debe tener en cuenta la distribución de los elementos y la función que tiene cada uno de ellos. La configuración arquitectónica más habitual se basa en el modelo denominado *Cliente/Servidor*, basado en la idea de servicio, en el que el cliente es un componente consumidor de servicios y el servidor es un proceso proveedor de servicios. Además, esta relación está robustamente cimentada en el intercambio de mensajes como el único elemento de acoplamiento entre ambos. En este libro, y en este capítulo en concreto, nos vamos a centrar en las características de los componentes software que se utilizan en el cliente.

Uno de los componentes más habituales en el cliente es el navegador web, que permite acceder al contenido ofrecido por los servidores de Internet sin la necesidad de que el usuario instale un nuevo programa (con excepciones). Podemos encontrarnos muchos tipos de clientes en función de sus capacidades, los lenguajes soportados o las facilidades de configuración. Los más livianos, o "ligeros", son los que por sí solos no pueden ejecutar ninguna operación real más allá de la de conectarse al servidor. Sin embargo, actualmente la tendencia es a disponer de clientes complejos, que utilizan lenguajes como Java o funciones avanzadas en DHTML para otorgar mayor funcionalidad y flexibilidad al usuario. Estos navegadores pueden no solo conectarse al servidor, sino que también son capaces de procesar o sincronizar datos para su uso sin necesidad de que el usuario intervenga.

En cualquier caso, debemos entender que un navegador web, o explorador web (*browser*), es una aplicación, distribuida habitualmente como software libre, que permite a un usuario acceder (y, normalmente, visualizar) a un recurso publicado por un servidor web a través de Internet y descrito mediante una dirección URL (*Universal Resource Locator*). Como ya hemos dicho, lo más habitual es que utilicemos los exploradores web para "navegar" por recursos de tipo hipertexto, comúnmente descritos en HTML, ofrecidos por servidores web de todo el mundo a través de Internet.

Desde la creación de la Web a principios de los años 90, los navegadores web han evolucionado desde meros visualizadores de texto que, aunque no ofrecían capacidades multimedia (visualización de imágenes), cumplían su propósito (Links, Lynx, W3M); hasta los actuales navegadores, totalmente preparados para soportar cualquier tipo de

interacción y funcionalidad requerida por el usuario. A continuación describimos una pequeña lista de algunos de los exploradores más relevantes a lo largo de la corta historia de los clientes de navegación web:

- **Mosaic**. Se considera uno de los primeros navegadores web y el primero con capacidades gráficas. Las primeras versiones se diseñaron para ser ejecutado sobre UNIX pero, debido a su gran aceptación, pronto fue portado a las plataformas de Windows y Macintosh. Se utilizó como base para las primeras versiones de Internet Explorer y Mozilla. Su desarrollo se abandonó en 1997.

- **Netscape Navigator** (después **Communicator**). Fue el primer navegador en incluir un módulo para la ejecución de código *script* (JavaScript). Se le considera como el perdedor de la "Guerra de los navegadores", que tuvo lugar entre Netscape y Microsoft por el dominio del mercado de navegadores web a finales de los años 90. Sus características, sin embargo, se consideran la base de otros navegadores, como Mozilla Firefox.

- **Internet Explorer**. Es el navegador de Microsoft. Su cuota de distribución y uso ha sido muy elevada gracias a su integración con los sistemas Windows. En los últimos años su utilización ha ido descendiendo paulatinamente debido al aumento de usuarios que optan por otros navegadores, como Firefox o Chrome. A fecha de finales de 2011, la última versión publicada es la 9.0, que incorpora numerosos avances en cuanto a soporte de estándares web, personalización de la navegación, seguridad, etc.

- **Mozilla Firefox**. Se trata de un navegador de código abierto multiplataforma de gran aceptación en la comunidad de desarrolladores web. Existen gran variedad de utilidades, extensiones y herramientas que permiten la personalización tanto del funcionamiento del navegador como de su apariencia. Fue uno de los primeros en incluir la navegación por pestañas. Además, es un navegador multiplataforma, lo que le ha llevado a poder recortar parte de la cuota de distribución que desde los inicios de la década de los 2000 venía teniendo Internet Explorer.

- **Google Chrome**. De reciente creación (septiembre de 2008), es el navegador de Google compilado a partir de componentes de código abierto. En boca de sus desarrolladores, sus características principales son la seguridad, velocidad y estabilidad. En numerosos test comparativos este navegador ha demostrado ser uno de los más rápidos y seguros gracias, entre otras razones, a estar construido siguiendo una arquitectura multiproceso en la que cada pestaña es ejecutada de forma independiente.

- **Safari**. Es el navegador por defecto de los sistemas de Apple, aunque también se han desarrollado versiones para su funcionamiento en las plataformas Windows. Las últimas versiones incorporan las características habituales de navegación por pestañas, corrector ortográfico en formularios, almacenamiento de direcciones favoritas ("marcadores"), bloqueador de ventanas emergentes, soporte para motores de búsqueda personalizados o un gestor de descargas propio.

- **Dolphin Browser**. Debido al auge de los dispositivos móviles inteligentes (*smartphones* y *tablets*) y de los sistemas operativos para estos, tenemos que hacer referencia obligatoriamente a uno de los navegadores más populares para estas plataformas. Específico para el sistema operativo Android, fue uno de los primeros en incluir soporte para navegación multitáctil. Utiliza un motor de renderizado de páginas similar al de Chrome o Safari.

En la siguiente gráfica puede verse una tendencia de la utilización de navegadores en el período 2008-2012 (obtenida de *gs.StatCounter.com*):

Figura 1.1. *Estadísticas de uso de navegadores (2008-2012)*

Podemos diferenciar los navegadores anteriores de acuerdo a una serie de criterios:

- **Plataforma de ejecución**. No todos los navegadores pueden ser ejecutados en cualquier sistema operativo. Safari, por ejemplo, es exclusivo de los sistemas de Apple aunque tiene versiones para Windows.
- **Características del navegador**. La mayoría de los navegadores ofrecen funcionalidades adicionales asociadas a la experiencia del usuario a la hora de navegar por la Red. Algunas de las características soportadas de forma nativa (sin necesidad de instalar extensiones) incluyen la administración de marcadores, gestores de descarga, almacenamiento seguro de contraseñas y datos de formularios, corrección ortográfica o definición de herramientas de búsqueda.
- **Personalización de la interfaz**. Las funciones de accesibilidad que definen la experiencia del usuario con un navegador web también son un aspecto diferencial. Entre los aspectos más destacados podemos mencionar el soporte para la navegación por pestañas, la existencia de bloqueadores de ventanas emergentes, la integración de visualizadores de formatos de ficheros (como PDF), opciones de *zoom* o funciones avanzadas de búsqueda de texto en páginas web.
- **Soporte de tecnologías Web**. Actualmente, una de las mayores preocupaciones de los desarrolladores de navegadores web es el grado de soporte de los estándares de la Web. Por ello, podemos clasificar los navegadores de acuerdo a su nivel de soporte de tecnologías como CSS (hojas de estilo en cascada), Java, lenguajes de *scripting* del cliente (JavaScript), RSS o Atom (sindicación de contenidos), XHTML (HTML con formato de XML), etc.
- **Licencia de software**. Existen navegadores de código libre, como Mozilla Firefox (licencia GNU GPL) o Google Chrome (licencia BSD), y navegadores propietarios como Internet Explorer (Microsoft) o Safari (Apple). Salvo raras excepciones (OmniWeb) todos los navegadores son gratuitos.

1.2 ARQUITECTURA DE EJECUCIÓN

Cada navegador web tiene su propia forma de interpretar la interacción con un usuario. El resultado de esta interacción, en cualquier caso, se inicia con el usuario indicando la dirección del recurso al que quiere acceder y termina con la visualización del recurso por parte del navegador en la pantalla del usuario (salvo interacciones posteriores del usuario con la página). La forma de realizar este proceso depende del propósito del navegador y de la configuración del mismo. De esta forma, un navegador puede estar más centrado en ofrecer una respuesta más rápida, en mostrar una respuesta más fiel al contenido del recurso obtenido, en priorizar los aspectos de seguridad de las comunicaciones con el servidor, etc.

Para poder llevar a cabo el proceso descrito anteriormente, cada navegador está formado por una serie de elementos y componentes determinados que conforman lo que se denomina *arquitectura del navegador*. A pesar de que cada navegador tiene su propia arquitectura, la gran mayoría de ellos coinciden en una serie de componentes básicos y comunes en todos ellos, es lo que llamamos *arquitectura de referencia*. De acuerdo al estudio llevado a cabo por Grosskurth y Godfrey los componentes básicos incluidos en la arquitectura de referencia de un navegador web son los que pueden verse en la siguiente figura:

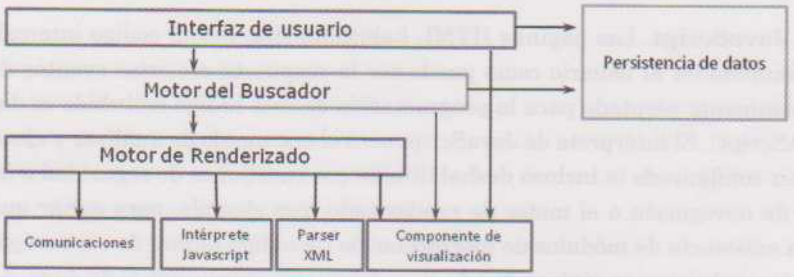


Figura 1.2. *Arquitectura de referencia de un navegador web*

Los componentes de esta arquitectura de referencia son:

- **Subsistema de interfaz de usuario**. Es la capa que actúa de interfaz entre el usuario y el motor del buscador (o de navegación). Ofrece funcionalidades tales como la visualización de barras de herramientas, progreso de carga de la página, gestión inteligente de las descargas, preferencias de configuración de usuario o impresión. En algunos casos puede comunicarse con el sistema operativo para el manejo de sesiones de usuario o el almacenamiento de preferencias de visualización o configuración.
- **Subsistema del motor del buscador o motor de navegación**. Este subsistema es un componente que ofrece una interfaz de alto nivel para el motor de renderizado. Su función principal es la de cargar una dirección determinada (URL o URI) y soportar los mecanismos básicos de navegación tales como ir a la página anterior o siguiente o la recarga de la página. Además, es el componente que gestiona las alertas de JavaScript (mensajes de usuario) y el proceso de carga de una página (es quien le provee de información a la interfaz de usuario al respecto). Finalmente, es el encargado de consultar y administrar las preferencias de ejecución del motor de renderizado.

- **Subsistema de renderizado**. Este componente es el encargado de producir una representación visual del recurso obtenido a partir del acceso a una dirección web. El código de una página web es *interpretado* por este módulo. En función de los lenguajes, estándares y tecnologías soportadas por el navegador, este módulo será capaz de mostrar documentos HTML, XML, hojas de estilo CSS e incluso contenido embebido en la página (audio/vídeo) e imágenes. Además, este módulo establece las dimensiones exactas de cada elemento a mostrar y, en ocasiones, es el responsable de posicionar dichos elementos en una página.

  Algunos de los motores de renderizado más utilizados son:

  - *Gecko*, utilizado en Firefox, Mozilla Suite y otros navegadores como Galeon.
  - *Trident*, el motor de Internet Explorer para Windows.
  - *WebKit*, el motor de Google Chrome, Epiphany y Safari.
  - *Presto*, el motor de Opera.
  - *Tasman*, el motor de Internet Explorer para Mac.

- **Subsistema de comunicaciones**. Es el subsistema encargado de implementar los protocolos de transferencia de ficheros y documentos utilizados en Internet (HTTP, FTP, etc.). Además, es el responsable de identificar la codificación de los datos obtenidos en función de su tipo, de tal forma que es capaz de identificar si el recurso obtenido es de tipo texto, audio, vídeo, etc. (codificado en estándar MIME, *Multipurpose Internet Mail Extensions*). Dependiendo de las capacidades personalizadas para el navegador, este subsistema puede almacenar una caché de elementos accedidos recientemente.

- **Intérprete de JavaScript**. Las páginas HTML habitualmente llevan código intercalado para la provisión de ciertas funcionalidades al usuario como puede ser la respuesta a ciertos eventos del ratón o del teclado. El lenguaje comúnmente aceptado para la programación de este código embebido es JavaScript (siguiendo el estándar ECMAScript). El intérprete de JavaScript será el encargado de analizar y ejecutar dicho código. Este módulo puede ser configurado (e incluso deshabilitado) por cuestiones de seguridad o facilidad de navegación desde el motor de navegación o el motor de renderizado (por ejemplo, para evitar que aparezcan ventanas emergentes). La existencia de módulos de interpretación de código difiere de un navegador a otro. Por ello, es posible que existan subsistemas intérpretes de otros lenguajes, como *applets* de Java, AJAX o ActionScript.

- **Parser XML**. Con el fin de poder acceder más fácilmente a los contenidos definidos en un documento HTML (en realidad XHTML), los navegadores web suelen incluir un módulo (*parser*) que permite cargar en memoria una representación en árbol (árbol DOM, *Document Object Model*) de la página. De esta forma, el acceso a los diferentes elementos de una página por parte del navegador es mucho más rápido.

- **Componente de visualización**. Este subsistema ofrece funcionalidades relacionadas con la visualización de los contenidos de un documento HTML en una página web. Ofrece primitivas de dibujo y posicionamiento en una ventana, un conjunto de componentes visuales predefinidos (*widgets*) y un conjunto de fuentes tipográficas a los subsistemas principales del navegador web. Suele estar muy relacionado con las librerías de visualización del sistema operativo.

- **Subsistema de persistencia de datos**. Funciona como almacén de diferentes tipos de datos para los principales subsistemas del navegador. Estos datos suelen estar relacionados con el almacenamiento de historiales de navegación y el mantenimiento de sesiones de usuario en disco. Otros datos de alto nivel que también son gestionados por este subsistema incluyen las preferencias de configuración del navegador (de barras de herramientas, por ejemplo) o el listado de marcadores. A bajo nivel, este sistema administra también los certificados de seguridad y las *cookies*.

1.3 LENGUAJES Y TECNOLOGÍAS DE PROGRAMACIÓN EN ENTORNO CLIENTE

Los lenguajes de programación del entorno de cliente son aquellos que se ejecutan en el navegador web, dicho de otro modo, en el lado del cliente dentro de una arquitectura Cliente/Servidor. El lenguaje cliente principal es HTML (lenguaje de marcado de hipertexto, *HyperText Markup Language*), ya que la mayoría de páginas del servidor son codificadas siguiendo este lenguaje para describir la estructura y el contenido de una página en forma de texto. Existen algunas alternativas y variaciones de este lenguaje tales como XML (lenguaje de marcas extensible, *eXtensible Markup Language*), DHTML (*Dynamic HTML*) o XHTML (*eXtensible HTML*). Con el fin de mejorar la interactividad con el usuario, en este grupo de lenguajes cliente podemos incluir todos los lenguajes de *script*, tales como JavaScript (el más utilizado dentro de esta rama) o VBScript. También existen otros lenguajes más independientes, como ActionScript (para crear contenido Flash) o AJAX (como extensión a JavaScript para comunicación asíncrona). A continuación vamos a analizar brevemente estos lenguajes.

1.3.1 HTML Y DERIVADOS

HTML es una particularización de un lenguaje anterior, SGML (*Standard Generalized Markup Language*), un sistema para sistema para la organización y etiquetado de documentos estandarizado en 1986 por la organización ISO. El término HTML se suele referir a ambas cosas, tanto al tipo de documento como al lenguaje de marcas.

El *Hyper Text Markup Language* (lenguaje de marcado de hipertexto) es el lenguaje de marcas de texto más utilizado en la *World Wide Web*. Fue creado en 1989 por Tim Berners-Lee a partir de dos elementos previos para crear dicho lenguaje: por un lado, el concepto de *hipertexto* como herramienta básica para conectar dos elementos (documentos o recursos) entre sí; y SGML, como lenguaje básico para colocar etiquetas o marcas en un texto. Debemos tener en cuenta que HTML no es propiamente un lenguaje de programación como puede serlo Java o VisualBasic, sino que se basa en la utilización de un sistema de etiquetas cerrado aplicado a un documento de texto. Además, este lenguaje no necesita ser compilado, sino que es interpretado (ejecutado a medida que se avanza por el documento HTML). Una característica particular de HTML es que, ante algún error de sintaxis que presente el texto, HTML no lo detectará y seguirá con la interpretación del siguiente fragmento de documento. El entorno para desarrollar HTML puede ser simplemente un procesador de textos.

Con el lenguaje HTML se pueden hacer gran variedad de acciones, desde organizar simplemente el texto y los objetos de una página web, pasando por crear listas y tablas, hasta llegar a la esencia de la Web: los hipervínculos. Un hipervínculo es un enlace de una página web o un archivo a otra página web u otro archivo. Cuando un usuario hace clic en el hipervínculo, el destino se mostrará en un explorador web, se abrirá o se ejecutará, en función del tipo de recurso destino. El destino es con frecuencia otra página web, pero también puede ser una imagen, un archivo multimedia, un documento de Microsoft Office, una dirección de correo electrónico, un programa, etc.

Con el tiempo, HTML ha ido evolucionando dando lugar a lenguajes derivados de este, como XML, XHTML o DHTML, en función de la flexibilidad ofrecida al conjunto de etiquetas admitido o de la integración de HTML con otros lenguajes que permitan dotar de más dinamismo e interactividad a las páginas creadas con HTML.

- **XML** es un lenguaje de etiquetado extensible muy simple, pero con unas reglas de codificación muy estrictas que juegan un papel fundamental en el intercambio de una gran variedad de datos. Es un lenguaje muy similar a HTML (también deriva de SGML), pero cuyo objetivo principal es describir datos para su transferencia eficiente y no mostrarlos, como es el caso de HTML.
- El lenguaje **XHTML** es también muy similar al lenguaje HTML. De hecho, XHTML no es más que una adaptación de HTML al lenguaje XML. Técnicamente, como ya hemos comentado, HTML es descendiente directo del lenguaje SGML, mientras que XHTML lo es del XML (que, a su vez, también es descendiente de SGML). Las páginas y documentos creados con XHTML son muy similares a las páginas y documentos HTML. Actualmente, los procesos de estandarización de HTML y XHTML siguen procesos paralelos.
- El HTML Dinámico (**DHTML**) consiste en una forma de aportar interactividad a las páginas web. El origen de DHTML hay que buscarlo en la versión 4.0 de los navegadores Netscape Communicator e Internet Explorer (y posteriores versiones de ambos navegadores), que permitieron la integración de HTML con lenguajes de *scripting* (JavaScript), hojas de estilo personalizadas (CSS) y la identificación de los contenidos de una página web en formato de árbol (DOM). Es la combinación de estas tecnologías la que permite aumentar la funcionalidad e interactividad de la página.

  La principal aportación de DHTML es servir de herramienta con la que podemos crear efectos que requieren poco o ningún ancho de banda ya que se ejecutan en el entorno del cliente. Se puede utilizar para crear animaciones, juegos, aplicaciones, etc., para introducir nuevas formas de navegar a través de los sitios web y para crear un auténtico entramado de capas que solo con HTML sería imposible abordar. Aunque muchas de las características del DHTML se podrían duplicar con otras herramientas como Java o Flash, DHTML ofrece la ventaja de que no requiere ningún tipo de extensión para poder utilizarlo.

Podríamos decir que HTML es el lenguaje más importante en el ámbito de la *World Wide Web* puesto que casi todos los lenguajes utilizados en la Web, de acuerdo a la tendencia mostrada en la última década, terminan confluyendo hacia una representación en HTML para que nuestro navegador lo pueda leer y visualizar en la pantalla del cliente.

1.3.2 CSS

Las CSS (Hojas de Estilo en Cascada, *Cascade Style Sheets*) sirven para separar el formato que se quiere dar a la página web de la estructura de la página web y las demás instrucciones. Utilizamos CSS, por ejemplo, cuando queremos que en determinados párrafos de nuestra página web se use un determinado tipo y tamaño de letra, un color de fuente y un color de fondo. En vez de tener que definir párrafo por párrafo todos los atributos del formato que queremos dar, solo hace falta que lo definamos una vez, en la hoja de estilo (CSS). Nos basta con poner una referencia en nuestro documento HTML que la dirija al formato que queramos darle, definido en la hoja de estilo. De esta forma, solo debemos poner esa referencia en cada párrafo, en vez de especificar el formato uno por uno.

1.3.3 JAVASCRIPT

JavaScript es un lenguaje de programación de *scripting* (interpretado) y, normalmente, embebido en un documento HTML. Se define como orientado a objetos, débilmente tipado y con características dinámicas. Se utiliza principalmente su forma del lado del cliente, con un intérprete implementado como parte de un navegador web. Su objetivo principal es el de permitir realizar mejoras en la interfaz de usuario y, de esta forma, crear páginas web dinámicas. Existe, no obstante, una forma de JavaScript del lado del servidor.

Inicialmente, se diseñó con una sintaxis similar al lenguaje C, aunque adopta nombres y convenciones propias del lenguaje de programación Java. Sin embargo, tenemos que dejar claro que Java y JavaScript no están relacionados y tienen semánticas y propósitos diferentes. Actualmente, existen dos especificaciones estándares denominadas ECMAScript e ISO/IEC 16262, que son generalizaciones del lenguaje JavaScript (que se creó antes que los estándares). A partir de la versión 5.1 de ECMAScript, este lenguaje está totalmente alineado con el estándar ISO 16262. JavaScript, y otros lenguajes similares como ActionScript para Adobe Flash, se consideran "dialectos" del estándar.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Tradicionalmente se venía utilizando en páginas web HTML para realizar operaciones y únicamente en el marco de la aplicación cliente, sin acceso a funciones del servidor. Ya que este lenguaje es uno de los más predominantes entre los lenguajes dedicados al cliente web, hablaremos sobre su sintaxis y uso en capítulos posteriores.

1.3.4 APPLETS DE JAVA

Una manera de incluir funcionalidades complejas en el ámbito de una página web se puede hacer integrando pequeños componentes (objetos independientes) en dicha página. Cuando la tecnología utilizada en estos componentes es Java los denominamos *applets* (es importante que tengamos en cuenta que estos fragmentos de código Java se ejecutan en el cliente). Estos *applets* se programan en Java y, por tanto, se benefician de la potencia y flexibilidad que este lenguaje nos ofrece. Es otra manera de incluir código para ejecutar en los clientes que visualizan una página web. Se trata de pequeños programas que se transfieren con las páginas web y que el navegador ejecuta en el espacio de la página (a través de un módulo o extensión concretos).

Los *applets* se programan en Java y se envían al cliente precompilados, es por ello que la manera de trabajar de estos varía un poco con respecto a los lenguajes de *script* como JavaScript. Una de las principales ventajas de utilizar *applets* es que son mucho menos dependientes del navegador que los *scripts* en JavaScript, incluso independientes del sistema operativo del ordenador donde se ejecutan. Además, en principio Java es más potente que JavaScript, por lo que el número de aplicaciones de los *applets* podrá ser mayor.

Como desventajas notables frente al uso de JavaScript, cabe señalar que los *applets* son más lentos de procesar y que tienen un espacio delimitado en la página donde se ejecutan, es decir, no se mezclan con todos los componentes de la página ni tienen acceso a ellos. Como consecuencia directa y, en general, con los *applets* de Java no podremos realizar directamente acciones tales como abrir ventanas secundarias, controlar marcos (*frames*), obtener la información de formularios, administrar capas, etc.

1.3.5 AJAX

AJAX, acrónimo de *Asynchronous JavaScript And XML* (JavaScript Asíncrono y XML), es un conjunto de técnicas y métodos de desarrollo web para la creación aplicaciones web interactivas. El primer aspecto que define a AJAX es que este tipo de aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios que acceden a una página web. La segunda característica es que, al contrario que con una página web HTML/XHTML/DHTML, en la que la comunicación se interrumpe una vez el cliente recibe la página, con AJAX se mantiene una comunicación asíncrona con el servidor en segundo plano (sin que el usuario sea consciente de dicha comunicación). La consecuencia directa de esta técnica es que podemos realizar cambios sobre las páginas del cliente sin necesidad de que éste proceda a recargarlas. Este hecho implica un aumento de la interactividad con el usuario y de la velocidad en las aplicaciones.

El fundamento de AJAX se encuentra en la utilización de un objeto específico de JavaScript denominado *XMLHttpRequest*, disponible y aceptado por la mayoría de los navegadores actuales. AJAX no es una tecnología en sí misma, sino que en realidad es una combinación de 4 tecnologías existentes:

- Lenguaje XHTML/HTML y hojas de estilo en cascada (CSS), con los que se especifican la estructura y contenidos de la página web.
- DOM, como forma de organizar en árbol los contenidos de una página para así poder acceder más fácilmente a un elemento determinado.
- El objeto `XMLHttpRequest`, que es el que tiene implementadas las operaciones necesarias para comunicarse asíncronamente con el servidor.
- XML, como lenguaje utilizado por el objeto `XMLHttpRequest` para recuperar e intercambiar información con el servidor (aunque cualquier formato es válido en principio).

Como podemos ver, AJAX está basado en estándares abiertos y ampliamente soportados, como JavaScript o DOM, por lo que es válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores. Entre las desventajas que pueden detectarse con respecto al desarrollo de aplicaciones cliente con AJAX podemos mencionar una curva de aprendizaje del desarrollo de este tipo de aplicaciones más elevada o el hecho de que, al tratarse de comunicaciones asíncronas, el navegador no es capaz de guardar un historial real de los contenidos a los que se ha accedido.

1.3.6 ADOBE FLASH Y ACTIONSCRIPT

Flash es una tecnología de animación actualmente bajo licencia de Adobe y que utiliza ActionScript como lenguaje principal. La principal ventaja de Flash es la capacidad para crear gráficos y animaciones vectoriales.

La evolución de Flash lo ha llevado de ser una simple herramienta de dibujo y animación de escritorio a ser una base para la programación multimedia y, más recientemente, convertirse en una completa aplicación de desarrollo para la Web. Posee su propio lenguaje de programación orientado a objetos derivado del estándar ECMAScript (denominado ActionScript) que puede adaptarse a la mayoría de los navegadores y sistemas operativos. En los últimos años, la tendencia muestra que la tecnología Flash, por los motivos que se verán más adelante, está en claro declive, pudiéndose observar una mayor prevalencia de otras tecnologías y lenguajes como HMTL 5.

Esta circunstancia adversa no es inconveniente para que entendamos que las posibilidades de Flash son extraordinarias, puesto que cada nueva versión ha mejorado a la anterior. Aunque su uso más frecuente es el de crear animaciones sus usos pueden ser otros (reproducción de vídeo, por ejemplo). El uso de Flash ha permitido crear aplicaciones interactivas de gran complejidad y visualmente muy atractivas que no sería posible crear utilizando DHTML o XHTML directamente. Además, gracias a las características dinámicas del lenguaje utilizado, la utilización de Flash permite aumentar el grado de interactividad del usuario con la página web.

En el lado opuesto, entre las desventajas que tradicionalmente se le han detectado al desarrollo de aplicaciones y páginas web basadas en esta tecnologías podemos destacar, entre otras, el hecho de que, al tratarse de creación de animaciones de índole vectorial, el consumo de procesador (y de batería en dispositivos móviles) es más elevado o que se trate de un software 100% propietario, coartando la libertad de extender o mejorar el núcleo de Flash.

1.4 INTEGRACIÓN DEL CÓDIGO CON LAS ETIQUETAS HTML

Para que podamos desarrollar aplicaciones web que se ejecuten en el lado del cliente lo primero que debemos saber es que el documento base estará escrito en HTML (o XHTML en su defecto). En esta sección nos centraremos en algunas de las formas que un desarrollador de páginas web tiene a su disposición a la hora de integrar código de *scripting* en documentos HTML. Las particularidades de otras tecnologías se verán en los capítulos correspondientes.

La integración de JavaScript y HTML/XHTML podemos hacerla de diferentes formas. Esta flexibilidad se refleja en, al menos, tres formas de incluir código JavaScript en páginas web.

1.4.1 JAVASCRIPT EN EL MISMO DOCUMENTO HTML

HTML se basa en el uso de unas etiquetas predefinidas para marcar el texto. Una de estas etiquetas es `<script>` (y `</script>` para indicar la finalización de un bloque de código embebido). Esta etiqueta puede incluirse en cualquier parte del documento, aunque se recomienda que el código JavaScript, salvo para propósitos concretos de generación de contenido a visualizar, se defina dentro de la cabecera del documento HTML (entre las etiquetas `<head>` y `</head>`). Podemos ver un ejemplo a continuación:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type"
      content="text/html;charset=iso-8859-1" />
    <title>Ejemplo 1</title>
    <script type="text/javascript">
      alert("Prueba de JavaScript");
    </script>
  </head>
  <body>
    <h1>Ejemplo 1: código embebido</h1>
  </body>
</html>
```

El ejemplo anterior muestra una página XHTML. Para que sea válida tenemos que incluir la cabecera "`<!DOCTYPE...`" y añadir el atributo `type` con el valor correcto a la etiqueta `<script>`. Los valores válidos para este atributo están estandarizados. Para el caso de JavaScript, el valor correcto es `text/javascript`.

Esta técnica suele utilizarse cuando se definen instrucciones que se referenciarán desde cualquier parte del documento o cuando se definen funciones con fragmentos de código genéricos. La mayor desventaja de este método es que, si ese fragmento de código lo queremos utilizar en otras páginas, debemos incluirlo en cada una de ellas, lo cual es un inconveniente cuando tenemos que realizar modificaciones de dicho código.

1.4.2 JAVASCRIPT EN UN ARCHIVO EXTERNO

Las mismas instrucciones de JavaScript que se incluyen entre un bloque `<script></script>` pueden almacenarse en un fichero externo con extensión `.js`. Al igual que sucede con los documentos HTML, los ficheros `.js` pueden crearse con cualquier editor de texto. A continuación se muestra el contenido de un fichero externo que contiene código JavaScript.

Archivo mensaje.js:

```
alert("Prueba de JavaScript");
```

La forma de acceder y enlazar esos ficheros `.js` con el documento HTML/XHTML es a través de la propia etiqueta `<script>`. No existe un límite en el número de ficheros `.js` que pueden enlazarse en un mismo documento HTML/XHTML. El siguiente ejemplo muestra cómo se enlazaría un documento HTML/XHTML con el fichero anterior `mensaje.js`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
≈Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type"
      content="text/html;charset=iso-8859-1" />
    <title>Ejemplo 2</title>
    <script type="text/javascript"
      src="/inc/mensaje.js"></script>
  </head>
  <body>
    <h1>Ejemplo 2: fichero externo</h1>
  </body>
</html>
```

Al igual que en el ejemplo anterior, para que esta técnica funcione, además del atributo `type` tenemos que incluir el atributo `src`. Este atributo contendrá un valor que indicará la ruta relativa (con respecto al fichero HTML/XHTML) de la ruta donde se encuentra el archivo JavaScript que se quiere enlazar. En este caso está dentro de una carpeta denominada `inc`. La única restricción de la etiqueta `<script>` es que solo puede enlazarse un archivo en cada etiqueta. Podemos incluir el número de etiquetas `<script>` que necesitemos.

Entre las ventajas de este método está que la vinculación de un mismo fichero externo puede hacerse desde varios documentos HTML/XHTML distintos. De esta forma, en el caso de que haya que modificar algo, solo hay que hacerlo en un único fichero. Cualquier modificación realizada en el fichero externo se ve reflejada inmediatamente en todas las páginas que lo enlacen.

1.4.3 JAVASCRIPT EN ELEMENTOS HTML

El último método suele utilizarse habitualmente como forma de controlar los eventos que suceden asociados a un elemento HTML concreto (aunque también puede utilizarse con otros fines). Consiste en insertar fragmentos de JavaScript dentro de atributos de etiquetas HTML de la página:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type"
      content="text/html;charset=iso-8859-1" />
    <title>Ejemplo 3</title>
  </head>
  <body>
    <p onclick="alert('Prueba de JavaScript');">
      Ejemplo 3: código en atributos
    </p>
  </body>
</html>
```

La principal desventaja es que el código JavaScript se intercala con el código HTML y, dependiendo de la complejidad y longitud de éste, el mantenimiento y modificación del código puede resultar más complicado.

ACTIVIDADES 1.1

- En este capítulo se ha mencionado la "Guerra de los Navegadores". Se propone que el alumno profundice en este período, identifique sus consecuencias y debata con los compañeros acerca del momento actual (considerado como la "Segunda Guerra de los Navegadores").
- La arquitectura de referencia presentada en la Figura 1.2 puede aplicarse a cualquier navegador. Se propone que el alumno aplique esta arquitectura investigando los diferentes módulos y componentes de un navegador actual como Google Chrome.
- La tecnología Flash está siendo sustituida progresivamente por capacidades similares dentro del estándar HTML 5. Se propone que el alumno investigue cuál es el soporte que HTML 5 propone para la visualización de contenido multimedia (puede consultarse en la página del estándar HTML del W3C).
- La forma de vincular un fichero externo mediante el atributo `src` de la etiqueta `<script>` requiere una ruta relativa. Se propone que el alumno estudie las reglas de esta vinculación y los metacaracteres permitidos (puede consultarse en la página del estándar HTML del W3C).

RESUMEN DEL CAPÍTULO

El desarrollo de aplicaciones en el lado del cliente parte de la base de un componente software específico que permite acceder a los contenidos ofrecidos por un servidor, como parte de la arquitectura Cliente/Servidor, a través de Internet. Este componente se conoce con el nombre de "navegador web". En los últimos años muchos tipos de navegadores web han aparecido con diferentes características y para ser ejecutados en diferentes plataformas. Un aspecto común a todos ellos es el soporte para la gran mayoría de tecnologías que se pueden utilizar en el desarrollo de aplicaciones en el entorno del cliente. Entre estas tecnologías destaca HTML como lenguaje de descripción de marcas en documentos de texto accedidos a través de la Web. Otros lenguajes y tecnologías, como JavaScript, CSS, AJAX, Flash o los *applets* de Java, han permitido dotar al cliente de numerosas herramientas que facilitan la interactividad con el usuario mejorando su experiencia de navegación en la Web.

Finalmente, debemos ser conscientes de las restricciones y particularidades que cada lenguaje requiere si queremos integrar estas tecnologías con el lenguaje HTML y crear páginas web atractivas y dinámicas.

EJERCICIOS PROPUESTOS

1. Implementar los ejemplos propuestos en el apartado 1.4 y probarlos con los navegadores web más utilizados que aparecen en la Figura 1.1.

TEST DE CONOCIMIENTOS

1 Señale la respuesta correcta con respecto a las características de un navegador web:
- a) No pueden realizar peticiones al servidor sin intervención directa del usuario.
- b) Todos los navegadores web actuales son gratuitos.
- c) Todos los navegadores actuales soportan la ejecución de JavaScript. pág. 22
- d) No pueden almacenar ningún tipo de dato relacionado con la navegación del cliente.

2 ¿Cuál de los siguientes módulos no forma parte de la arquitectura de referencia de un navegador web?
- a) Módulo de gestión de usuarios.
- b) Módulo de persistencia de datos.
- c) Módulo de comunicaciones.
- d) Motor de renderizado.

3 Señale la respuesta correcta con respecto a las características de XHTML:
- a) Al igual que HTML, no tiene por qué estar bien construido para que sea válido.
- b) Es la unión de los principios de XML con el lenguaje HTML.
- c) No permite la inclusión de código JavaScript.
- d) Ninguna los anteriores.

4 Señale la respuesta correcta con respecto a JavaScript:
- a) No está soportado por ninguno de los navegadores actuales.
- b) Es un estándar que indica la forma en la que un servidor debe ejecutar un programa externo.
- c) No se puede utilizar para programar *applets*. pág 22
- d) Se puede utilizar para programar Flash.

5 ¿Cuál de los siguientes fragmentos de código sería correcto para integrar código JavaScript en HTML?
- a) `<h1 onclick="<script>alert("Prueba de JavaScript "); </script>">.`
- b) `<script type="text" src="/inc/mensaje.js"> </script>.`
- c) `<h1 src="/inc/mensaje.js">Mensaje</h1>.`
- d) Ninguna de las anteriores.

4. Applets se programan en Java y Flash utiliza ActionScript

Interfaz Usuario
Motor del Buscador
Motor Renderizado
↓ Parse XML
↓ ↓ Intérprete JS
Comunicaciones
→ Persistencia datos
Componente Visualización

<script type="text/JavaScript"
src="archivo.js">
</script>

2 Introducción al lenguaje JavaScript

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer las principales características del lenguaje JavaScript.
- ✓ Dominar la sintaxis básica del lenguaje.
- ✓ Comprender y utilizar los distintos tipos de variables y operadores presentes en el lenguaje JavaScript.
- ✓ Conocer las diferentes sentencias condicionales de JavaScript y saber realizar operaciones complejas con ellas.

En el capítulo anterior hemos visto algunos de los lenguajes y tecnologías de programación en el entorno del cliente. Algunos de estos lenguajes permiten mejorar la interactividad con el usuario y uno de los lenguajes más utilizados para este propósito es JavaScript. En este capítulo veremos las principales características de este lenguaje, comenzando con el ejemplo "Hola Mundo", el cual se suele usar como un primer ejercicio típico que sirve como introducción al estudio de cualquier lenguaje de programación. A continuación, estudiaremos la sintaxis del lenguaje, además de los conceptos relacionados con los tipos de datos, las variables y los operadores utilizados por JavaScript. Por último, veremos las sentencias condicionales, las cuales nos permitirán aumentar la complejidad de los ejemplos y ejercicios propuestos durante todo el capítulo.

2.1 CARACTERÍSTICAS DE JAVASCRIPT

JavaScript es un lenguaje de programación interpretado que se utiliza fundamentalmente para dotar de comportamiento dinámico a las páginas web. Por ello, cualquier navegador web actual incorpora un intérprete para código JavaScript.

El primer lenguaje de *scripting* para la Web fue LiveScript, desarrollado por Netscape para proporcionar una alternativa "ligera" a Java a la hora de soportar comportamiento dinámico tanto en el lado del servidor como en el del cliente. No obstante, dado el auge y popularidad de Java, en 1995 Netscape y Sun aprovecharon el lanzamiento de la versión 2 de Navigator, el navegador web de Netscape para anunciar que el lenguaje pasaba a denominarse JavaScript. Por otro lado, el éxito de JavaScript provocó que, incluso cuando ya existía VBScript, otro lenguaje de *scripting* para la Web basado en BASIC, Microsoft lanzase su propia versión de JavaScript, denominada JScript e integrada en sus navegadores a partir de Internet Explorer 3.

A pesar de que las diferencias entre JavaScript y JScript no fueran muy grandes, el hecho de que cada navegador soportase su propio lenguaje obligaba a veces a los desarrolladores a programar dos veces la misma funcionalidad (una con cada lenguaje) e introducir en sus páginas web sentencias condicionales para distinguir en qué navegador se estaba ejecutando la página e invocar la ejecución del código escrito con el lenguaje soportado por el navegador en cuestión.

En respuesta a este tipo de problemas, la ECMA (*European Computer Manufacturers Association*) emprendió un esfuerzo de estandarización que desembocó en la publicación del estándar ECMAScript[1]. Aunque a día de hoy, tanto JavaScript como JScript son conformes a dicho estándar, es el término JavaScript el que se impuso y se utiliza como denominador común para referirse al propio lenguaje y al estándar.

A continuación enumeramos algunas características de JavaScript:

- La sintaxis de JavaScript se asemeja a la de C++ y la de Java. De hecho, JavaScript está basado en el concepto de objeto, pero no es un lenguaje orientado a objetos.

[1] *http://www.ecma-international.org/publications/standards/Ecma-262.htm*

Cuando utilizamos JavaScript en el contexto de la programación web en el lado del cliente tendemos a confundir los objetos JavaScript con los objetos del documento HTML que manejamos con JavaScript, es decir, los objetos del *Document Object Model* (DOM). Está distinción se hace evidente cuando utilizamos JavaScript en otros contextos, como en la programación en el lado del servidor o en las aplicaciones Flash.

- Los objetos JavaScript utilizan herencia basada en prototipos, muy diferente de la herencia basada en clases propia de los lenguajes orientados a objetos, como Java. En JavaScript los objetos heredan propiedades directamente de otros objetos sin necesidad de que sean instancias de una misma clase (o de clases que participen en una misma jerarquía).
- Probablemente la característica más importante de JavaScript sea que es un lenguaje débilmente tipado, lo que permite que una variable pueda contener valores de distintos tipos en diferentes momentos de la ejecución del programa. A cambio, muchos errores de programación no aparecen hasta que el programa es ejecutado, ya que el compilador es incapaz de detectarlos.
- Una característica *poco deseable* de JavaScript es el hecho de que, por defecto, todas las variables son globales. Es decir, aquellas variables que no son definidas dentro de otra variable se ubican en un espacio de nombres común denominado *global object*. En general, el uso de variables globales no es una buena práctica de programación.

2.2 "HOLA MUNDO" CON JAVASCRIPT

La forma más inmediata de empezar a experimentar con JavaScript es escribir secuencias de comandos simples. En este sentido, una ventaja de JavaScript es que un explorador web y un simple editor de textos constituyen un completo entorno de desarrollo; no necesitamos comprar ni descargar ningún software especial para empezar a experimentar con JavaScript.

Para ejecutar cualquier programa JavaScript tenemos que embeber el código JavaScript en una página HTML, de la misma forma que insertamos cualquier otro contenido HTML: utilizando etiquetas para marcar el principio y el fin del bloque de código JavaScript. En este caso utilizamos la etiqueta `<script></script>`. De esta forma el navegador sabe que el bloque de texto contenido entre la etiqueta de inicio y fin no se corresponde con código HTML, si no que se trata de código que debe ser procesado antes de mostrar el resultado en pantalla. Así, el navegador, que incluye un intérprete de JavaScript, se encargará automáticamente de ejecutar el código cuando cargue la página y reflejar el efecto de la ejecución sobre la página que tiene que mostrar.

Hay dos formas de embeber el código JavaScript utilizando la etiqueta `<script>`:

- Incluirlo directamente en la página HTML mediante la etiqueta `<script>`. Crea un fichero `HolaMundo.html` y copia y pega el siguiente código en el fichero:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv= "content-type" content="text/html;
      charset=utf-8">
    <title>Hola Mundo</title>
  </head>
  <body>
    <script>
      alert('Hola mundo en JavaScript')
    </script>
  </body>
</html>
```

- Utilizar el atributo `src` de dicha etiqueta para especificar el fichero que contiene el código JavaScript.

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;
      charset=utf-8">
    <title>Hola Mundo</title>
    <script type="text/javascript" src="2.3HolaMundo.js">
    </script>
  </head>
  <body></body>
</html>
```

El fichero `2.3-HolaMundo.js` debe contener esta línea de código:

```
alert('Hola mundo en JavaScript')
```

Si abriésemos los dos ficheros `HolaMundo.html` con un navegador web, suponiendo que antes hayamos creado el fichero `2.3-HolaMundo.js` en el mismo directorio, el resultado sería exactamente el mismo: el navegador muestra el mensaje "Hola mundo en JavaScript".

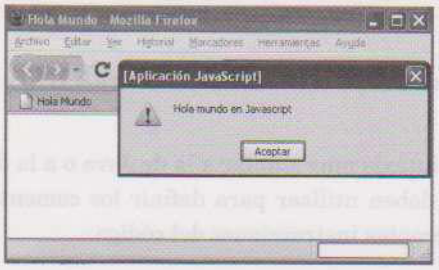


Figura 2.1. "Hola mundo" en JavaScript

No obstante, la forma recomendada de proceder es la segunda, ya que al localizar el código JavaScript en un fichero externo estamos facilitando la reutilización y modularización de dicho código.

Dado que hay distintos lenguajes de *script*, el atributo `type` nos permite decirle al navegador cuál es el utilizado para codificar el *script* que encontrará a continuación. De esta forma, el navegador sabrá cuál es el intérprete que tiene que utilizar para ejecutar el *script*. Podemos omitirlo, ya que los navegadores más conocidos utilizan JavaScript como lenguaje de *script* por defecto, pero es una buena práctica especificarlo y, además, es obligatorio de acuerdo a la norma de la W3C, la cual es la encargada de la especificación del estándar.

ACTIVIDADES 2.1

- Cree un nuevo fichero HTML vacío (puedes tomar el fichero `2.2-HolaMundo.html` como ejemplo y eliminar la etiqueta `<script>`). Añada el siguiente guión JavaScript en el cuerpo de la página (entre las etiquetas `<body>` y `</body>`):

```
<script type="text/javascript">
  document.bgColor = "red";
  alert('Cambiamos el color de la p\xE1gina');
</script>
```

- Guarde la página con otro nombre (`Actividad-2.1-CambiarColor.html`), ábrala con su navegador y compruebe el resultado.

Podemos observar que en la primera actividad hemos utilizado una cadena de escape para mostrar correctamente los caracteres tildados. Para evitar este problema debemos emplear la cadena `\xdd`, donde *dd* corresponde al carácter especial especificado por dos dígitos hexadecimales según el estándar Unicode. Existen diferentes secuencias de escape que son útiles a la hora de introducir no solo caracteres tildados, sino también saltos de línea (`\n`), tabuladores (`\t`), etc.

2.3 EL LENGUAJE JAVASCRIPT: SINTAXIS

El lenguaje JavaScript tiene una sintaxis muy similar a la de Java o a la del lenguaje C++. La sintaxis especifica aspectos, como los caracteres que se deben utilizar para definir los comentarios, la forma de los nombres de las variables o el modo de separar las diferentes instrucciones del código.

2.3.1 MAYÚSCULAS Y MINÚSCULAS

Una de las primeras dificultades que encuentran los programadores novatos en JavaScript es que este lenguaje distingue entre mayúsculas y minúsculas, a diferencia de HTML, que no realiza ninguna diferencia entre ellas. Esta regla cobra importancia cuando utilizamos variables, objetos, funciones o cualquier otro símbolo del lenguaje.

Por ejemplo, no es lo mismo utilizar la función `alert()` que `Alert()`. Tal y como hemos visto en el ejemplo de `HolaMundo.html`, la primera muestra un texto en una ventana emergente del navegador, mientras que la segunda no existe a menos que la defina el programador.

2.3.2 COMENTARIOS EN EL CÓDIGO

Al igual que la mayor parte de los lenguajes de programación, JavaScript permite insertar comentarios dentro del código. Estás líneas de código no son interpretadas por el navegador. Su función principal es la de facilitar la lectura del código al programador.

Existen dos formas de insertar comentarios. Una de ellas es a través de la doble barra (`//`), con la cual comentamos una sola línea de código. La otra forma que podemos utilizar para comentar un código es a través de los signos `/*` al inicio del comentario y los signos `*/` al final del mismo. De este modo podemos comentar varias líneas de código. En el siguiente código vemos un ejemplo de las dos formas que existen para insertar comentarios:

```
<script type="text/javascript">
  // Este modo permite comentar una sola línea
  /* Este modo permite realizar
  comentarios de
  varias líneas */
</script>
```

2.3.3 TABULACIÓN Y SALTOS DE LÍNEA

JavaScript ignora los espacios, las tabulaciones y los saltos de línea presentes entre los símbolos del código. La única restricción en cuanto a los saltos de línea la vemos en el siguiente apartado.

En el momento en que los programas empiezan a ser complejos, podemos apreciar la utilidad de emplear las tabulaciones y los saltos de línea adecuados para mejorar la presentación y la legibilidad del código. A continuación,

podemos observar la diferencia entre un código bien estructurado que facilita la lectura y un código que no está bien estructurado.

Versión 1:

```
<script type="text/javascript">var i,j=0;
for (i=0;i<5;i++){ alert("Variable i: "+i);
for (j=0;j<5;j++){ if (i%2==0){
document.write
(i + "-" + j + "<br>");}}}</script>
```

Versión 2:

```
<script type="text/javascript">
  var i,j=0;
  for (i=0;i<5;i++){
    alert("Variable i: "+i;
    for (j=0;j<5;j++){
      if (i%2==0){
        document.write(i + "-" + j + "<br>");
      }
    }
  }
</script>
```

Podemos notar que en la segunda versión del ejemplo es mucho más fácil comprender a cuál sentencia corresponde cada instrucción. En el ejemplo hemos utilizado sentencias condicionales y otros conceptos que estudiaremos a lo largo del capítulo.

2.3.4 EL PUNTO Y COMA

Normalmente, se suele insertar un signo de punto y coma (;) al final de cada instrucción de JavaScript, al igual que se hace en lenguajes de programación como C, C++ o Java. Este signo tiene la utilidad de separar y diferenciar cada instrucción. No obstante, podemos omitir dicho signo si cada instrucción de JavaScript se encuentra en una línea independiente.

Por ejemplo, las siguientes instrucciones podrían escribirse de este modo:

```
pi_egipcio = 3.16
pi_griego = 3.14
```

Sin embargo, si queremos definir los dos valores en una misma línea, es necesario utilizar al menos el primer punto y coma:

```
pi_egipcio = 3.16; pi_griego = 3.14;
```

La omisión del punto y coma no es una buena práctica de programación y un simple olvido nos puede hacer perder un tiempo muy valioso, con lo cual es aconsejable habituarnos a su uso.

2.3.5 PALABRAS RESERVADAS

JavaScript contiene una serie de palabras que no podemos utilizar para definir nombres de variables, funciones o etiquetas. Según la versión 5.1 de ECMAScript, las palabras reservadas son las que vemos a continuación:

Tabla 2.1 Palabras reservadas

`break`	`delete`	`if`	`this`	`while`
`case`	`do`	`in`	`throw`	`with`
`catch`	`else`	`instanceof`	`try`	
`continue`	`finally`	`new`	`typeof`	
`debugger`	`for`	`return`	`var`	
`default`	`function`	`switch`	`void`	

Además de las anteriores palabras reservadas del lenguaje, el estándar ECMAScript contempla el uso de otras palabras reservadas en futuras versiones, como por ejemplo `class`, `const`, `enum` y `export`, entre otras. Aunque los interpretadores actuales de JavaScript permiten el uso de estas posibles futuras palabras clave, es aconsejable revisar la versión del estándar a la hora de la lectura de este libro e indagar cuáles son en el momento actual, con el fin de evitar su uso en la realización de programas.

ACTIVIDADES 2.2

- Visite la página web *http://www.ecmascript.org/* y consulte todas las palabras reservadas para las futuras versiones del estándar.

2.4 TIPOS DE DATOS

En todo programa informático manipulamos constantemente diferentes tipos de valores como por ejemplo números o textos. Los tipos de datos especifican qué tipo de valor se guardará en una determinada variable. Este concepto es importante a la hora de determinar cómo podemos combinar ciertas variables o si es posible hacerlo.

Los tres tipos de datos primitivos de JavaScript son: números, cadenas de texto y valores booleanos. Además de estos tres tipos de datos, existe el tipo de datos compuesto llamado `objeto`. Este último representa una colección de valores primitivos o de valores compuestos por otros objetos. Esta colección de valores nos permite definir tipos de datos como por ejemplo las matrices y las funciones, las cuales explicaremos en los siguientes capítulos del libro.

2.4.1 NÚMEROS

En JavaScript, a diferencia de la mayoría de lenguajes de programación, existe solo un tipo de datos numérico. Todos los números que utilicemos, ya sean valores enteros o valores de punto flotante, se representarán a través del formato de punto flotante de 64 bits definido por el estándar IEEE 754[2]. Este formato es el llamado `double` en los lenguajes Java o C++.

Además de los valores enteros en base 10, en JavaScript es posible utilizar valores hexadecimales, es decir, en base 16. Para ello, es necesario iniciar el valor con `0x` seguido por una secuencia de dígitos hexadecimales.

La mayor parte de las implementaciones de JavaScript admite también la representación de valores en base 8, aunque el estándar ECMAScript no lo admite. Debido a esto es aconsejable evitar el uso de los valores en formato octal.

2.4.2 CADENAS DE TEXTO

El tipo de datos que utiliza JavaScript para representar cadenas de texto, es llamado `string`. Con este tipo de datos es posible representar una secuencia de letras, dígitos, signos de puntuación o cualquier otro carácter de Unicode. La cadena de caracteres la debemos definir entre comillas dobles o comillas simples (" o ').

Dado que las comillas dobles o simples las utilizamos para delimitar las cadenas de texto, es posible que surja la duda de qué hacer para representar estos signos. Para ello, necesitamos utilizar secuencias de escape haciendo uso de la barra invertida (\). La combinación de esta barra con otros caracteres permite que el navegador pueda representar un carácter, que sin la ayuda de la barra invertida no podría representarse. En la Tabla 2.2 podemos ver las principales combinaciones de secuencias de escape.

[2] *http://grouper.ieee.org/groups/754/*

Tabla 2.2 Secuencias de escape

Secuencia de escape	Resultado
\\	Barra invertida
\'	Comilla simple
\"	Comillas dobles
\n	Salto de línea
\t	Tabulación horizontal
\v	Tabulación vertical
\f	Salto de página
\r	Retorno de carro
\b	Retroceso

2.4.3 VALORES BOOLEANOS

El tipo de datos booleano, también conocido como valores lógicos, es el tipo de datos más simple debido a que admite solo dos valores: `true` o `false` (verdadero o falso). Debemos definir estos valores sin ningún tipo de comillas. Este tipo de datos es bastante útil a la hora de evaluar expresiones lógicas o verificar condiciones particulares en nuestros programas.

ACTIVIDADES 2.3

- Cree un fichero HTML que permita mostrar el siguiente texto en una ventana emergente, a través de la función `alert()`:

  I'm = I am
  I don't = I do not

- En la Figura 2.2 puede ver el resultado esperado. Tenga en cuenta que debe usar secuencias de escape para poder representar las comillas simples y el salto de línea.

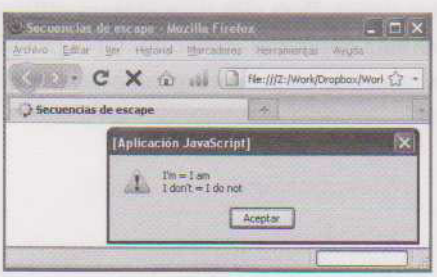


Figura 2.2. *Ejemplo del uso de las secuencias de escape*

2.5 VARIABLES

Las variables se pueden definir como zonas de la memoria de un ordenador que se identifican con un nombre y en las cuales se almacenan ciertos datos. Las variables nos permiten manipular y guardar datos que, en algunos casos, no sabremos su valor a priori. El desarrollo de un *script* conlleva generalmente dos aspectos relacionados con las variables:

- ✓ Declaración de variables.
- ✓ Inicialización de variables.

A continuación entraremos en los detalles de cada uno de estos dos aspectos.

2.5.1 DECLARACIÓN DE VARIABLES

El primer paso para utilizar una variable es definirla. En JavaScript las variables se definen a través de la palabra clave `var` seguida por el nombre que queramos darle a la variable, tal y como vemos en el siguiente ejemplo:

```
var mi_variable_1;
var mi_variable_2;
```

Es posible declarar más de una variable en una sola línea de código a través de la separación por comas:

```
var mi_variable_1, mi_variable_2;
```

En los anteriores ejemplos hemos declarado las variables, pero no les hemos asignado ningún valor. El contenido de las variables estará indefinido hasta que una parte del código almacene un valor en ellas. Este proceso lo describimos en el siguiente apartado.

2.5.2 INICIALIZACIÓN DE VARIABLES

Podemos realizar la asignación de un valor a una variable de tres formas:

- Asignación directa de un valor concreto.
- Asignación indirecta a través de un cálculo en el que se implican a otras variables o constantes.
- Asignación a través de la solicitud del valor al usuario del programa.

A continuación, podemos ver los ejemplos respectivos de cada caso:

```
var mi_variable_1 = 30;
var mi_variable_2 = mi_variable_1 + 10;
var mi_variable_3 = prompt('Introduce un valor:');
```

En todas ellas debemos utilizar el operador de asignación (el signo igual, "="). Si en algún fragmento del código intentamos utilizar una variable que no haya sido inicializada, JavaScript generará un error.

Podemos inicializar una variable sin utilizar la palabra clave `var`. De este modo JavaScript declara implícitamente dicha variable, pero tenemos que tener en cuenta que esta variable será una variable global. Esto quiere decir que su valor tendrá un ámbito global, incluso dentro de funciones locales. De este modo corremos el riesgo de modificar un valor en el cual se basa algún otro fragmento del programa. Afortunadamente, este problema lo podemos evitar simplemente si nos acordamos siempre de utilizar la palabra clave `var`.

Una vez declarada una variable y haberle asignado un valor, podemos utilizarla en otras instrucciones del programa. En el momento en que usamos una variable, el navegador accede a la memoria del ordenador, recupera su valor y lo sustituye en la instrucción.

ACTIVIDADES 2.4

- Cree un nuevo fichero HTML y, al igual que en las anteriores actividades, copie el siguiente código JavaScript dentro del cuerpo de la página:

```
<script type="text/javascript">
  var primer_saludo = "hola";
  var segundo_saludo = primer_saludo;
  primer_saludo = "hello";
  alert(segundo_saludo);
</script>
```

- Antes de ver el resultado, ¿cuál cree que será el valor en la ventana emergente?

2.6 OPERADORES

Hasta el momento hemos utilizado ejemplos con sentencias bastante sencillas pero, a partir de ahora, podremos construir diferentes expresiones en las que tendremos una colección de símbolos, palabras o números con los que podremos realizar cálculos más complejos. Es posible construir este tipo de expresiones gracias al uso de los operadores de JavaScript.

JavaScript utiliza principalmente cinco tipo de operadores:

- Operadores aritméticos.
- Operadores lógicos.
- Operadores de asignación.
- Operadores de comparación.
- Operadores condicionales.

Los elementos utilizados en una expresión y unidos a través de un operador se definen como operandos. A continuación presentamos los diferentes tipos de operadores.

2.6.1 OPERADORES ARITMÉTICOS

Los operadores aritméticos permiten realizar cálculos elementales entre variables numéricas. En la Tabla 2.3 podemos ver las cuatro operaciones aritméticas elementales: suma, resta, multiplicación y división, además de otros tres operadores menos comunes: módulo, incremento y decremento.

Tabla 2.3 Operadores aritméticos

Operador	Nombre	Descripción
+	Suma	Efectúa la suma entre los operandos.
-	Resta	Efectúa la resta entre los operandos.
*	Multiplicación	Efectúa la multiplicación entre los operandos.
/	División	Efectúa la división entre los operandos.
%	Módulo	Extrae la parte entera del resultado de la división entre los operandos.
++	Incremento	Permite incrementar un valor.
--	Decremento	Permite decrementar un valor.

El operador módulo divide un número entre otro y lo que devuelve es el resto de dicha división. Por ejemplo, si quisiéramos comprobar que el año 2012 es un año bisiesto (teniendo en cuenta algunas excepciones, los años bisiestos son divisibles por 4) debemos realizar la siguiente operación:

```
var anyo = 2012;
var bisiesto = anyo % 4;
```

Si la variable `bisiesto` almacena el valor 0, podemos afirmar que 2012 es un año bisiesto.

Los operadores incremento y decremento permiten realizar una de las operaciones más comunes en los lenguajes de programación, es decir, incrementar o decrementar un valor en una unidad. Es posible utilizar estos operadores antes o después de la variable, aunque su efecto es diferente en ambos casos. Por ejemplo, el siguiente código almacena el valor 2 en ambas variables

```
variable_1 = 1;
variable_2 = ++variable_1;
```

Mientras que en este otro ejemplo la primera variable contendrá al valor 2 y la segunda 1:

```
variable_1 = 1;
variable_2 = variable_1++;
```

2.6.2 OPERADORES LÓGICOS

Los operadores lógicos tienen la utilidad de combinar o manipular diferentes expresiones lógicas con el fin de evaluar si el resultado de esta combinación es verdadero o falso. Generalmente, se utiliza este tipo de operadores en el caso en que debamos tomar decisiones dentro de un programa. Podemos ver los tres operadores lógicos que existen en la Tabla 2.4.

Tabla 2.4 Operadores lógicos

Operador	Nombre	Descripción
&&	Y	Ejecuta la operación booleana AND sobre los valores. Devuelve true solo si todos los valores son `true`. Devuelve `false` en caso contrario.
||	O	Ejecuta la operación booleana OR sobre los valores. Devuelve true en el caso en que al menos uno de los valores sea `true`. Devuelve `false` en caso contrario.
!	No	Invierte el valor booleano de su operando.

2.6.3 OPERADORES DE ASIGNACIÓN

Además del operador de asignación igual (=), JavaScript cuenta con otros operadores de asignación que tienen un característica en común con los operadores aritméticos de incremento y decremento. Esta característica es la de ahorrarnos tiempo y disminuir la cantidad de código escrito en los programas. Gracias al uso de estos operadores, podemos obtener métodos abreviados que permiten evitar el tener que escribir dos veces la variable que se encuentra a la izquierda del operador. La Tabla 2.5 muestra todos los operadores de asignación.

Tabla 2.5 Operadores de asignación

Operador	Nombre	Descripción
+=	Suma y asigna	Ejecuta una suma y asigna el valor al operando de la izquierda.
-=	Resta y asigna	Ejecuta una resta y asigna el valor al operando de la izquierda.
*=	Multiplica y asigna	Ejecuta una multiplicación y asigna el valor al operando de la izquierda.
/*	Divide y asigna	Ejecuta una división y asigna el valor al operando de la izquierda.
%=	Módulo y asigna	Ejecuta el módulo y asigna el valor al operando de la izquierda.

Por ejemplo, podríamos usar el operador de restar y asignar de este modo:

```
var deudas = 1500; // tenemos unas deudas de 1500 euros

deudas -= 300; // nuestras deudas disminuyen de 300 euros
// esto es lo mismo que escribir deudas = deudas - 300
```

2.6.4 OPERADORES DE COMPARACIÓN

Los operadores de comparación, tal y como indica su nombre, nos permiten comparar todo tipo de variables con el fin de verificar sus valores. El resultado de dicha comparación nos devolverá un valor booleano y determina el orden relativo de dos valores. La comparación ejecutada por estos operadores se puede realizar solo sobre números y cadenas. Si utilizamos otro tipo de datos, estos se convertirán automáticamente para intentar que la comparación se pueda llevar a cabo.

Tabla 2.6 Operadores de comparación

Operador	Nombre	Descripción
<	Menor que	Verifica si el operando a la izquierda del operador es menor que el operando de la derecha. Devuelve `true` en ese caso o `false` en caso contrario.
<=	Menor o igual que	Verifica si el operando a la izquierda del operador es menor o igual que el operando de la derecha. Devuelve `true` en ese caso o false en caso contrario.
==	Igual	Verifica si los dos operandos son iguales. Devuelve `true` en ese caso o `false` en caso contrario.
>	Mayor que	Verifica si el operando a la izquierda del operador es mayor que el operando de la derecha. Devuelve `true` en ese caso o false en caso contrario.
>=	Mayor o igual que	Verifica si el operando a la izquierda del operador es mayor o igual que el operando de la derecha. Devuelve `true` en ese caso o `false` en caso contrario.
!=	Diferente	Verifica si los dos operandos son diferentes. Devuelve `true` en ese caso o `false` en caso contrario.
===	Estrictamente igual	Verifica si el operando a la izquierda del operador es igual y del mismo tipo de datos que el operando de la derecha. Devuelve `true` en ese caso o `false` en caso contrario.
!==	Estrictamente diferente	Verifica si el operando a la izquierda del operador es diferente y/o de tipo diferente que el operando de la derecha. Devuelve `true` en ese caso o `false` en caso contrario.

2.6.5 OPERADORES CONDICIONALES

Existe otro tipo de operador un poco más complejo, pero bastante útil a la hora de tomar decisiones en los programas de JavaScript. Se trata del operador condicional. Con este operador podemos indicarle al navegador que ejecute una acción en concreto después de evaluar una expresión. El operador condicional consta de tres partes: la primera es la expresión a evaluar, la segunda es la acción a realizar si la expresión es verdadera, y la tercera parte es la acción a realizar si la expresión es falsa.

Tabla 2.7 Operadores condicionales

Operador	Nombre	Descripción
?:	Condicional	Si la expresión antes del operador es verdadera, se utiliza el primer valor a la derecha. En caso contrario se utiliza el segundo valor a la derecha.

Por ejemplo, si queremos avisar al usuario que no se puede efectuar una división por cero, es posible hacerlo mediante un operador condicional:

```
<script type="text/javascript">
  var dividendo = prompt("Introduce el dividendo: ");
  var divisor = prompt("Introduce el divisor: ");
  var resultado;
  divisor != 0 ? resultado = dividendo/divisor :
    alert("No es posible la división por cero");
    alert("El resultado es: " + resultado);
</script>
```

ACTIVIDADES 2.5

- La precedencia de los operadores determina el orden en que se ejecuta una expresión. Realice una búsqueda con el fin de encontrar alguna de las tablas que indiquen el orden de precedencia de los operadores de JavaScript.

2.7 SENTENCIAS CONDICIONALES

Los ejemplos que hemos utilizado hasta el momento han sido ejemplos lineales en los que las sentencias las ejecutábamos unas detrás de otras, desde la primera hasta la última. Podemos controlar la toma de decisiones y el posterior resultado por parte del navegador a través del uso de sentencias condicionales. Dichas sentencias permiten evaluar condiciones y ejecutar ciertas instrucciones si la condición es verdadera y ejecutar otras instrucciones diferentes si la condición es falsa.

Existen tres tipos de sentencias condicionales: la sentencia `if`, la sentencia `switch` y las sentencias en bucle `while` y `for`. La sentencia `if` indica al navegador si debe ejecutar una parte de código en base al valor lógico de una expresión condicional. La sentencia `switch` compara el valor de una variable con una serie de valores conocidos. Si uno de los valores conocidos coincide con el valor de la variable, se ejecuta el código asociado a dicho valor conocido. Las sentencias en bucle permiten al navegador ejecutar un fragmento de código de forma repetida mientras la condición sea verdadera.

2.7.1 SENTENCIA IF

La sentencia `if` es una sentencia fundamental a la hora de realizar controles en la ejecución de los programas. De este modo, el navegador tomará una decisión y ejecutará ciertas instrucciones de forma condicional. Su sintaxis es la siguiente:

```
if(expresión){
  instrucciones
}
```

La expresión puede ser una comparación o una expresión lógica que devuelve los valores `true` o `false`. Las instrucciones son el código que ejecutaremos si la expresión devuelve el valor `true`. JavaScript ignora las instrucciones en el caso en que la expresión devuelva el valor `false`.

El uso de las llaves `{}` no es obligatorio en el caso en que debamos ejecutar una sola instrucción. De lo contrario, sí que es obligatorio su uso.

La segunda forma de utilizar la sentencia `if` es agregando la palabra clave `else`. De este modo podemos ejecutar instrucciones en el caso en que la expresión devuelva el valor `false`. Así es como definimos una sentencia `if-else`:

```
if(expresión){
  instrucciones_si_true
}else {
  instrucciones_si_false
}
```

El diagrama de flujo de control de la sentencia `if-else` lo vemos en la Figura 2.3:

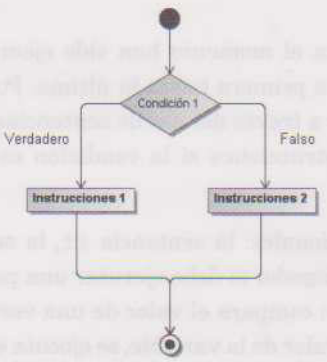


Figura 2.3. *Diagrama de flujo de control de la sentencia* `if-else`

La tercera forma de utilizar la sentencia `if` es mediante la sentencia `if-else-if`. La Figura 2.4 muestra su diagrama de flujo de control. De este modo, le pedimos al navegador que evalúe una expresión, y si ésta devuelve el valor `false`, el navegador evaluará una nueva expresión. Si ninguna de las dos o más expresiones utilizadas devuelve el valor `true`, es posible utilizar un último `else`, donde es posible escribir el código que se ejecutará en este caso. La estructura de la sentencia `if-else-if` es la siguiente:

```
if(expresión_1){
  instrucciones_1
}else if (expresión_2){
  instrucciones_2
}else{
  instrucciones_3
}
```

En el caso en que debamos utilizar demasiadas sentencias del tipo `else-if`, es preferible utilizar la sentencia `switch`, que estudiaremos en el siguiente apartado. Este es un error muy común en los programadores principiantes. El navegador no tendrá ningún problema en ejecutar el código, pero la lectura y mantenimiento de una sentencia compleja de varios `if-else` se puede convertir en una tarea difícil.

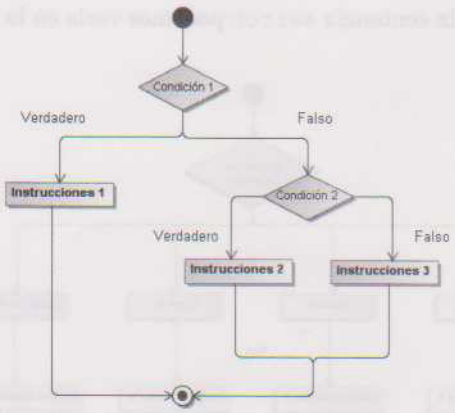


Figura 2.4. *Diagrama de flujo de control de la sentencia* `if-else-if`

Una vez que conozcamos y dominemos el uso de la sentencia `if`, será posible escribir tipos de código con tomas de decisiones más complejas a través de `if` anidados. Los `if` anidados nos permitirán solucionar problemas que se nos presentan más a menudo, como por ejemplo la toma de decisiones basadas en decisiones precedentes.

2.7.2 SENTENCIA SWITCH

En algunos casos es necesario comparar el valor de una variable con algunos valores conocidos. Para estas situaciones, JavaScript proporciona la sentencia `switch`. En esta sentencia se tiene una expresión a evaluar y una serie de posibles valores de dicha expresión, llamados casos, en los que se encuentran las instrucciones a ejecutar cuando coincidan el valor de la expresión y el valor de cada caso. La sintaxis de esta sentencia es la siguiente:

```
switch (expresión){
  case valor1:
    instrucciones a ejecutar si expresión = valor1
  break;
  case valor2:
    instrucciones a ejecutar si expresión = valor2
  break;
  case valor3:
    instrucciones a ejecutar si expresión = valor3
  break
  default:
    instrucciones a ejecutar si expresión es diferente a
    los valores anteriores
}
```

Cada caso termina con la palabra clave `break`. La utilidad de esta palabra clave es la de detener la ejecución del `switch` en el momento en que uno de los casos resulte verdadero. De este modo no perdemos tiempo en evaluar los demás casos. El diagrama de flujo de la sentencia `switch` podemos verlo en la Figura 2.5:

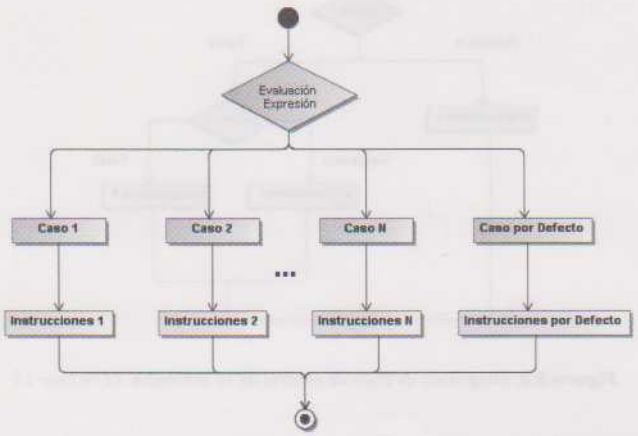


Figura 2.5. *Diagrama de flujo de control de la sentencia* `switch`

Las instrucciones que se encuentren dentro de la sentencia opcional llamada `default`, se ejecutarán solo cuando ninguno de los valores de cada caso coincida con el valor de la expresión.

2.7.3 BUCLE WHILE

El bucle más sencillo en JavaScript es el bucle `while`. A través de bucles de este tipo, el navegador ejecutará una o más instrucciones continuamente hasta que una cierta condición deje de ser verdadera. La Figura 2.6 muestra su diagrama de flujo. Este bucle es el más utilizado cuando no disponemos de la información que nos indique el número de veces que debemos repetir la iteración del bucle. El bucle `while` consta de tres partes fundamentales: la palabra clave `while`, la expresión condicional y las instrucciones que se ejecutan en el caso en que la expresión condicional sea verdadera. Su sintaxis es la siguiente:

```
while (expresión){
  instrucciones
}
```

Una variación del bucle `while` es el denominado `do-while`. Su estructura es la siguiente:

```
do{
  instrucciones
} while (expresión)
```

De este modo nos asegura que la ejecución del bucle se lleve a cabo al menos una vez y solo al final, se evalúa si se debe seguir ejecutando o no. Sin embargo, podemos realizar este proceso a través de un bucle `while`. Esta es una práctica recomendable visto que el bucle `do-while` se introdujo solo a partir de la versión 1.2 de JavaScript, con lo cual no todos los navegadores soportan este último bucle.

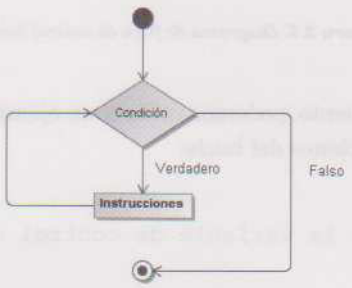


Figura 2.6. *Diagrama de flujo de control del bucle* `while`

2.7.4 BUCLE FOR

El bucle `for` permite que un navegador ejecute las instrucciones que se encuentren dentro del mismo bucle hasta que la expresión condicional devuelva el valor `false`. Este tipo de sentencia consta de cinco partes: la palabra clave `for`, el valor inicial de la variable de control, la condición basada en dicha variable, el incremento o decremento de la variable y el cuerpo del bucle.

Su sintaxis general es la siguiente:

```
for(valor_inicial_variable; expresión_condicional;
  incremento_o_decremento_de_la_variable){
    cuerpo_del_bucle
}
```

Tal y como podremos verificar, este bucle es una herramienta bastante potente a la hora de poder realizar diferentes actividades en pocas líneas de código. El diagrama de flujo de este bucle los vemos en la Figura 2.7. En este diagrama, la fase del incremento del contador se efectúa después de llevar a cabo las instrucciones del bucle, sin embargo, este incremento lo podemos realizar inmediatamente después de hacer la comprobación inicial de la condición o en alguna de las mismas instrucciones del bucle.

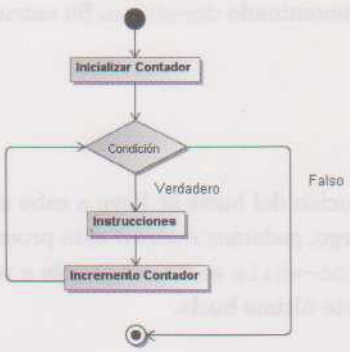


Figura 2.7. *Diagrama de flujo de control bucle* `for`

Para comprender mejor su funcionamiento, podemos utilizar un ejemplo que muestra en pantalla un valor que se incrementa en cada una de las diez iteraciones del bucle:

```
for(i=0; i<10; i++){
  document.write("El valor de la variable de control es: "
    + i + "<br>")
}
```

ACTIVIDADES 2.6

- Abra el fichero `Actividad-2.6-if.html` e introduzca la información requerida por el navegador. Posteriormente mire el código para que vea un ejemplo del uso de la sentencia condicional `if`.

RESUMEN DEL CAPÍTULO

En este capítulo hemos presentado una breve introducción de las características principales del lenguaje JavaScript, además de un primer ejemplo clásico denominado "Hola Mundo", el cual se utiliza en el aprendizaje de cualquier lenguaje de programación. Por otro lado, hemos introducido la sintaxis básica del lenguaje, teniendo en cuenta algunos de los errores más frecuentes en los programadores con poca experiencia.

A continuación, hemos presentado los tipos de datos usados por JavaScript: números, cadenas de texto y valores booleanos. Estos tipos de datos son usados por las variables, las cuales presentan dos fases principales para un correcto funcionamiento: la **declaración** y la **inicialización**.

Posteriormente, hemos presentado todas las categorías de los operadores del lenguaje: operadores aritméticos, operadores lógicos, operadores de asignación, operadores de comparación y operadores condicionales.

Los tipos de datos, las variables y los operadores son conceptos fundamentales en JavaScript. Estos conceptos se pueden definir como la base de cualquier aplicación de este lenguaje.

Por último, hemos introducido las principales sentencias condicionales: `if`, `switch`, `while` y `for`. De cada una de ellas, hemos enseñado las posibles variantes con su respectiva sintaxis. Con estos conceptos se ha aprendido a utilizar un navegador para llevar a cabo tareas de toma de decisiones o ejecución de instrucciones de forma repetitiva.

EJERCICIOS PROPUESTOS

1. Cree un fichero HTML vacío y llámelo `EjercicioPropuesto-2.1-Numeros.html`. Añada el siguiente código JavaScript en el cuerpo de la página (entre las etiquetas `<body>` y `</body>`):

```
<script type="text/javascript">
   var maxValue = Number.MAX_VALUE;
   var minValue = Number.MIN_VALUE;
   alert("Max Value: " + maxValue);
   alert("Min Value: " + minValue);
   alert("Valor especial: " + maxValue*2);
</script>
```

De este modo podremos comprobar el número más grande representable por JavaScript, el número más cercano a cero y el valor especial que representa el infinito.

2. Utilice el siguiente *script* para comprobar el operador lógico `&&`:

```
<script type="text/javascript">
  var operando_1 =
  eval(prompt("Introduce el primer
    valor lógico (true o false):", true));
  var operando_2 =
  eval(prompt("Introduce el segundo
    valor lógico (true o false):", true);
  var resultado_logico = operando_1 &&
  operando_2;
  alert("Resultado: " + resultado_
  logico);
</script>
```

Con este ejercicio pedimos al usuario que introduzca dos valores lógicos y posteriormente compruebe el resultado del operador `&&`.

3. Utilice el siguiente *script* en un fichero HTML y compruebe su funcionamiento. Posteriormente, cambie la línea `countdown--` por `countdown++` y concluya explicando por qué el navegador no deja de solicitarnos valores.

```
<script type="text/javascript">
  var countdown = prompt("Introduce un
  número para
    iniciar la cuenta atrás: ");
  while (countdown>0){
    alert(countdown+ "... ");
    countdown--;
  }
</script>
```

4. Realice el mismo ejercicio anterior, pero sustituyendo el bucle `while` por un bucle `for`.

TEST DE CONOCIMIENTOS

1 ¿Qué navegador web actual incorpora un intérprete para código JavaScript?
- a) Firefox.
- b) Safari.
- c) Chrome.
- d) Todos los anteriores.

2 ¿El lenguaje JavaScript distingue entre mayúsculas y minúsculas?
- a) Solamente en el código que se encuentre dentro de las etiquetas `<body>` y `</body>`.
- b) Solamente en el nombre de las variables.
- c) Distingue en todo el código JavaScript.
- d) Solamente en el uso de las palabras clave.

3 Para terminar una instrucción en JavaScript se utiliza:
- a) Un punto y coma.
- b) La palabra clave `end`.
- c) La etiqueta `</script>`.
- d) Un punto y coma o un salto de línea. pág. 35

4 ¿Cuál de los siguientes comentarios es correcto en JavaScript?
- a) `/ Comentario`
- b) `*/ Comentario */`
- c) `// Comentario`
- d) `<!--Comentario-->`

5 ¿Cuál es el número más grande representable por JavaScript?
- a) No existe un límite.
- b) 9.99×10^{308}.
- c) $1.7976931348623157 \times 10^{308}$.
- d) Ninguno de los anteriores.

6 En la expresión `a + b`, ¿qué parte de la expresión son las letras *a* y *b*?
- a) Operadores.
- b) Operandos.
- c) Sumas.
- d) Incrementos.

7 ¿Cuál es el resultado de la expresión `10<100 ? 'Verdadero' : 'Falso'`?
- a) True.
- b) 10<100.
- c) Falso.
- d) Verdadero.

8 ¿Para qué sirve la palabra `else` en una sentencia `if-else`?
- a) Para anidar una sentencia `if`.
- b) Para definir una nueva expresión condicional.
- c) Para ejecutar instrucciones en el caso en que la expresión condicional sea falsa. pág 46
- d) Para ejecutar instrucciones en el caso en que la expresión condicional sea verdadera.

9 ¿Qué sentencia condicional permite ejecutar instrucciones independientemente de si una condición sea verdadera o falsa?
- a) `if`.
- b) `do-while`.
- c) `while`.
- d) `for`.

3 Utilización de los objetos predefinidos de JavaScript

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer cuáles son los principales objetos predefinidos de JavaScript.
- ✓ Comprender las propiedades y métodos de los principales objetos de JavaScript.
- ✓ Aprender a generar código HTML desde sentencias JavaScript.
- ✓ Dominar el uso de los marcos de HTML y aprender a realizar interacciones entre ellos con JavaScript.
- ✓ Manipular y gestionar la creación y apariencia de las ventanas del navegador, además de la comunicación entre ellas.

Hasta el momento, en los capítulos anteriores del libro, hemos realizado operaciones de programación que se han limitado a asignar valores a variables, manipular variables, calcular expresiones o usar sentencias condicionales. JavaScript fue diseñado especialmente para realizar otro tipo de actividades como, por ejemplo, manipular los navegadores y las páginas web.

Algunas de las operaciones más comunes para las cuales se diseñó JavaScript son: abrir una nueva ventana en el navegador, escribir un texto en un documento, redirigir el navegador a otra ubicación, validar los datos de un formulario o cambiar la página de un marco. La realización de algunas de estas actividades las abordaremos en capítulos posteriores de este libro. Lo importante en este punto es entender que conceptos como ventana, documento, ubicación, formulario y marco se denominan objetos predefinidos de JavaScript. Dichos objetos son elementos programables que podemos manipular para cambiar sus propiedades, realizar tareas a través de sus métodos o ejecutar un evento relacionado con el mismo objeto. Propiedades, métodos y eventos son conceptos fundamentales para comprender el funcionamiento de un objeto.

- Las propiedades son las características de un objeto. Por ejemplo, algunas de las propiedades de una página son su título (`title`) o su color de fondo (`bgColor`).
- Los métodos son funciones o tareas específicas que pueden realizar los objetos. Por ejemplo, el método `sqrt()` del objeto llamado `Math` realiza el cálculo de la raíz cuadrada de un número.
- Los eventos son situaciones que pueden llegar a realizarse o no. Por ejemplo, podemos detectar y decidir qué hacer cuando el usuario pulse el botón derecho del ratón. Cada objeto puede reconocer una serie de eventos.

Para una mejor comprensión de estos conceptos podemos pensar en un término cotidiano como es un televisor y considerarlo como un objeto:

- El color, el fabricante, el precio o el tamaño son características que se consideran propiedades del televisor.
- Con un televisor es posible ver programas, en algunos casos grabarlos, o consultar la información del teletexto. Estas tareas serían los métodos del televisor.
- Cuando pulsamos el botón de encendido o pulsamos el botón para cambiar de canal, el televisor responde de una forma predeterminada que corresponde a los eventos pertenecientes al televisor.

La suma de propiedades, métodos y eventos definen una especificación general del objeto que en este caso hemos llamado televisor.

En este capítulo estudiaremos principalmente los dos primeros conceptos de los objetos, es decir, las propiedades y los métodos. Inicialmente estudiaremos los objetos predefinidos del lenguaje, haciendo una distinción entre aquellos que están principalmente relacionados con el navegador y aquellos que están relacionados con los elementos HTML. Gracias al estudio y conocimiento de estos objetos podremos introducir temas como la generación de elementos HTML, la interacción entre los marcos de una página web y por último, la gestión de las ventanas del navegador.

3.1 OBJETOS NATIVOS DE JAVASCRIPT

JavaScript proporciona una serie de objetos definidos nativamente que no dependen del navegador en el que los utilicemos y que, por tanto, podemos manipular en cualquier momento.

La creación de un objeto se lleva a cabo a través de la palabra clave `new`. La sintaxis para la creación de un objeto es la siguiente:

```
var mi_objeto = new Object();
```

A la variable llamada `mi_objeto` se le asigna una nueva (`new`) instancia de un objeto que puede ser cualquiera de los objetos predefinidos de JavaScript o los definidos por el usuario. En el siguiente capítulo abordaremos el tema de la creación de objetos personalizados. Dentro del paréntesis introducimos los parámetros necesarios para la inicialización del objeto. En JavaScript se accede a las propiedades y a los métodos de los objetos de forma similar a los demás lenguajes de programación, es decir, con el operador punto (".").

```
mi_objeto.nombre_propiedad;
mi_objeto.nombre_función([parámetros]);
```

Los objetos de JavaScript se ordenan de modo jerárquico, tal y como puede vemos en la Figura 3.1. En lo más alto de la jerarquía se encuentra el objeto `Window`, dado que representa la ventana del navegador. Este objeto posee más propiedades y métodos que los demás objetos de JavaScript. En el mimo nivel se encuentran los objetos predefinidos del lenguaje.

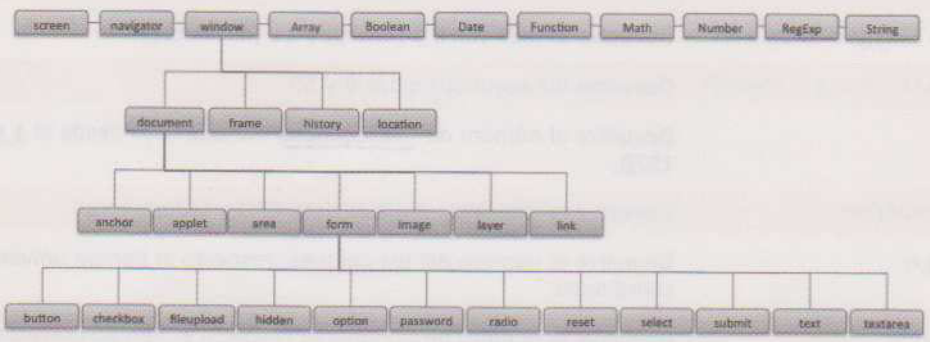


Figura 3.1. *Jerarquía de los objetos de JavaScript*

A continuación describimos las propiedades y los métodos de algunos de los principales objetos nativos de JavaScript, es decir, los objetos `Date`, `Math`, `Number` y `String`, dejando para el siguiente apartado los objetos que están directamente relacionados con el navegador.

3.1.1 EL OBJETO DATE

El objeto `Date` nos permite realizar controles relacionados con el tiempo en las aplicaciones web. Este objeto no presenta ninguna propiedad. Sin embargo, tiene una serie de métodos que podemos dividirlos en tres subconjuntos:

- **Métodos de lectura**. Estos métodos son aquellos que empiezan con el prefijo `get`. La principal función de este subconjunto de métodos es la de consultar las diferentes partes de una instancia del objeto `Date`.
- **Métodos de escritura**. Los métodos de escritura empiezan con el prefijo `set`. Tal y como indica su prefijo, la utilidad de estos métodos es inicializar las diferentes partes de una instancia del objeto `Date`.
- **Métodos de conversión**. Estos métodos convierten objetos de tipo `Date` en cadenas de texto o en milisegundos según los diferentes criterios que veremos a continuación.

Tabla 3.1 Métodos del objeto *Date*

Método	Descripción
`getDate()`	Devuelve el número correspondiente al día del mes entre 1 y 31.
`getDay()`	Devuelve el número correspondiente al día de la semana entre 0 (domingo) y 6 (sábado).
`getFullYear()`	Devuelve un número de 4 cifras correspondiente al año.
`getHours()`	Devuelve la hora entre 0 y 23.
`getMilliseconds()`	Devuelve los milisegundos entre 0 y 9999.
`getMinutes()`	Devuelve los minutos entre 0 y 59.
`getMonth()`	Devuelve el mes entre 0 (enero) y 11 (diciembre).
`getSeconds()`	Devuelve los segundos entre 0 y 59.
`getTime()`	Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970.
`getTimezoneOffset()`	Devuelve la diferencia entre la hora GMT y la hora local.
`getUTCDate()`	Devuelve el número del día del mes, respecto al tiempo universal coordinado.
`getUTCDay()`	Devuelve el número de la semana, respecto al tiempo universal coordinado.
`getUTCFullYear()`	Devuelve el número del año, respecto al tiempo universal coordinado.
`getUTCHours()`	Devuelva la hora, respecto al tiempo universal coordinado.
`getUTCMilliseconds()`	Devuelve los milisegundos, respecto al tiempo universal coordinado.
`getUTCMinutes()`	Devuelve los minutos, respecto al tiempo universal coordinado.
`getUTCMonth()`	Devuelve el número del mes, respecto al tiempo universal coordinado.

get → Devuelve set → establece

`getUTCSeconds()`	Devuelve los segundos, respecto al tiempo universal coordinado.
`parse()`	Analiza una fecha y devuelve el número de milisegundos pasados desde el 1 de enero de 1970 hasta la fecha analizada.
`setDate()`	Establece el valor del día del mes.
`setFullYear()`	Establece el valor del año.
`setHours()`	Establece el valor de la hora.
`setMilliseconds()`	Establece el valor de los milisegundos.
`setMinutes()`	Establece el valor de los minutos.
`setMonth()`	Establece el valor del mes.
`setSeconds()`	Establece el valor de los segundos.
`setTime()`	Establece una fecha, agregando o sustrayendo un número específico de milisegundos hasta o desde la medianoche del 1 de enero de 1970.
`setUTCDate()`	Establece el valor del día del mes, respecto al tiempo universal coordinado.
`setUTCFullYear()`	Establece el valor del año, respecto al tiempo universal coordinado.
`setUTCHours()`	Establece el valor de la hora, respecto al tiempo universal coordinado.
`setUTCMilliseconds()`	Establece el valor de los milisegundos, respecto al tiempo universal coordinado.
`setUTCMinutes()`	Establece el valor de los minutos, respecto al tiempo universal coordinado.
`setUTCMonth()`	Establece el valor del mes, respecto al tiempo universal coordinado.
`setUTCSeconds()`	Establece el valor de los segundos, respecto al tiempo universal coordinado.
`toDateString()`	Convierte la fecha del objeto `Date` en una cadena de caracteres.
`toLocaleDateString()`	Convierte la fecha del objeto `Date` en una cadena de caracteres, según las convenciones locales.
`toLocaleTimeString()`	Devuelve la parte del tiempo de un objeto `Date` en una cadena según las convenciones locales.
`toLocaleString()`	Convierte un objeto `Date` en una cadena de texto, según las convenciones locales.
`toTimeString()`	Convierte la parte de tiempo de un objeto `Date` en una cadena.
`toUTCString()`	Convierte un objeto `Date` en una cadena según el tiempo universal.
`UTC()`	Devuelve el número de milisegundos en una cadena de tipo fecha desde la medianoche del 1 de enero de 1970.

Todas las operaciones relacionadas con fechas recaen sobre el objeto `Date`. Para instanciar un objeto de este tipo existen dos formas: crear un objeto con la fecha actual y crear un objeto con una fecha diferente a la actual. Esto depende de los parámetros que definamos a la hora de crear el objeto. A continuación, veremos un ejemplo con dos objetos de tipo `Date` en el cual realizamos una operación de resta entre ellos.

```
<script type="text/javascript">
  var fecha_actual = new Date();
  var fecha_maya = new Date(2012, 11, 21);
  alert("La fecha actual es: " + fecha_actual);
  alert("El calendario Maya termina el: " + fecha_maya);
  var tiempo_restante = fecha_maya-fecha_actual;
  alert("Quedan " + tiempo_restante + " milisegundos para
    que termine el calendario Maya");
</script>
```

3.1.2 EL OBJETO MATH

El objeto `Math` permite realizar operaciones matemáticas en JavaScript. En ejemplos anteriores hemos realizado operaciones matemáticas básicas como son las operaciones aritméticas. Con el objeto `Math` es posible realizar otro tipo de tareas más complejas como, por ejemplo, calcular raíces cuadradas o funciones trigonométricas. Las propiedades almacenan una serie de constantes que son útiles para realizar ciertos cálculos matemáticos. Estas constantes deben ser conocidas por todo aquel que haya realizado algún curso de matemáticas avanzadas.

Tabla 3.2 Propiedades del objeto *Math*

Propiedad	Descripción
E	Devuelve la constante de Euler.
LN2	Devuelve el logaritmo natural de 2.
LN10	Devuelve el logaritmo natural de 10.
LOG2E	Devuelve el logaritmo de E en base 2.
LOG10E	Devuelve el logaritmo de E en base 10.
PI	Devuelve el valor de Pi.
SQRT1_2	Devuelve la raíz cuadrada de 1/2.
SQRT2	Devuelve la raíz cuadrada de 2.

Tabla 3.3 Métodos del objeto *Math*

Método	Descripción
`abs()`	Devuelve el valor absoluto.
`acos()`	Devuelve el arcocoseno.
`asin()`	Devuelve el arcoseno.
`atan()`	Devuelve el arcotangente.
`ceil()`	Devuelve el número entero superior.
`cos()`	Devuelve el coseno.
`exp()`	Devuelve el exponencial.
`floor()`	Devuelve el número entero inferior.
`log()`	Devuelve el logaritmo natural.
`max()`	Devuelve el número máximo entre los números pasados como argumento.
`min()`	Devuelve el número mínimo entre los números pasados como argumento.
`pow()`	Devuelve el resultado de un número elevado a una potencia pasada como argumento.
`random()`	Devuelve un número aleatorio entre 0 y 1.
`round()`	Redondea un número al número entero más próximo.
`sin()`	Devuelve el seno.
`sqrt()`	Devuelve la raíz cuadrada.
`tan()`	Devuelve la tangente.

Para utilizar el objeto `Math` no necesitamos definir un objeto con el constructor `new`. Las propiedades y los métodos son accesibles a través del nombre del tipo de objeto seguido por el nombre de la propiedad o el método. Los datos sobre los cuales queramos realizar la operación matemática, debemos pasarlos como argumentos de los métodos.

A continuación, utilizamos la propiedad `Math.PI` y el método `Math.pow()` para calcular el área de un círculo.

```
<script type="text/javascript">
  var r = prompt("Ingresa el radio de un c\xedrculo:");
  var area = Math.PI * Math.pow(r, 2);
  alert("El \xelrea del c\xedrculo es de: " + area + " cms
    cuadrados");
</script>
```

3.1.3 EL OBJETO NUMBER

Un objeto más para realizar tareas relacionadas con tipos de datos numéricos es el objeto `Number`. Es poco común crear objetos de tipo `Number` a través del constructor `new`, ya que por lo general se asignan directamente los valores numéricos a una variable. El objeto `Number` se utiliza principalmente para indicar el máximo y el mínimo valor posible que podemos representar con JavaScript e informar al usuario cuando sobrepase esos límites en alguna operación matemática.

Tabla 3.4 Propiedades del objeto *Number*

Propiedad	Descripción
`MAX_VALUE`	Devuelve el mayor número posible en JavaScript.
`MIN_VALUE`	Devuelve el menor número posible en JavaScript.
`NaN`	Representa el valor especial Not a Number.
`NEGATIVE_INFINITY`	Representa el infinito negativo.
`POSITIVE_INFINITY`	Representa el infinito positivo.

Tabla 3.5 Métodos del objeto *Number*

Método	Descripción
`toExponential()`	Convierte el número en una notación exponencial.
`toFixed()`	Formatea el número con la cantidad de dígitos decimales que pasemos como parámetro.
`toPrecision()`	Formatea el número con la longitud que pasemos como parámetro.

En el siguiente ejemplo muestra todas las propiedades del objeto `Number`, además de la construcción de una instancia y el uso de uno de sus métodos:

```
<script type="text/javascript">
  alert("Propiedad MAX_VALUE: " + Number.MAX_VALUE)
  alert("Propiedad MIN_VALUE: " + Number.MIN_VALUE)
  alert("Propiedad NaN: " + Number.NaN)
  alert("Propiedad NEGATIVE_INFINITY: " +
    Number.NEGATIVE_INFINITY)
  alert("Propiedad POSITIVE_INFINITY: " +
    Number.POSITIVE_INFINITY)
```

```
    var N1 = new Number(3.141592653589793);
    alert("Pi formateado: " + N1.toPrecision(3))
</script>
```

3.1.4 EL OBJETO STRING

El objeto `String` es aquel que permite manipular las cadenas de texto. Este objeto posee solamente una propiedad que indica el tamaño de la cadena. Sin embargo, presenta diferentes métodos que podemos aplicar a toda variable a la cual asignemos como valor una cadena de texto.

En capítulos anteriores del libro hemos utilizado diferentes ejemplos que usan cadenas de texto. A todas estas variables les podremos aplicar los diferentes métodos que vemos en la Tabla 3.7.

Tabla 3.6 Propiedades del objeto *String*

Propiedad	Descripción
`Lenght`	Corresponde a la longitud de una cadena de texto.

Tabla 3.7 Métodos del objeto *String*

Método	Descripción
`anchor()`	Devuelve una cadena convertida en un ancla de HTML.
`big()`	Aumenta el tamaño de una cadena.
`blink()`	Crea el efecto de una cadena parpadeando.
`bold()`	Muestra una cadena en negrita.
`charAt()`	Permite acceder a un carácter en concreto de una cadena.
`charCodeAt()`	Devuelve un carácter en concreto en su formato Unicode.
`concat()`	Concatena dos o más cadenas y devuelve una nueva con dicha concatenación.
`fixed()`	Convierte una cadena en una cadena con fuente monoespaciada.
`fontcolor()`	Modifica el color de la fuente de una cadena.
`fontsize()`	Modifica el tamaño de la fuente de una cadena.
`fromCharCode()`	Convierte valores Unicode en caracteres.

`indexOf()`	Devuelve la posición de la primera ocurrencia del carácter pasado como parámetro.
`italics()`	Muestra una cadena en cursiva.
`lastIndexOf()`	Devuelve la posición de la última ocurrencia del carácter pasado como parámetro.
`link()`	Muestra una cadena como un hipervínculo HTML con el enlace le pasemos como parámetro.
`match()`	Busca una coincidencia en una cadena y devuelve todas las coincidencias encontradas.
`replace()`	Busca una coincidencia en una cadena y si existe, la remplaza por otra cadena pasada como parámetro.
`search()`	Busca una coincidencia en una cadena y devuelve la posición de la coincidencia.
`slice()`	Extrae una parte de una cadena en base a los parámetros que indiquemos como índices de inicio y final.
`small()`	Disminuye el tamaño de una cadena.
`split()`	Corta una cadena en base a un separador que pasamos como parámetro.
`strike()`	Muestra una cadena tachada.
`sub()`	Muestra una cadena como subíndice.
`substr()`	Devuelve una subcadena en base a un índice y longitud pasados como parámetros.
`substring()`	Devuelve una subcadena en base a un índice de inicio y de final pasados como parámetros.
`sup()`	Muestra una cadena como superíndice.
`toLowerCase()`	Convierte una cadena en minúsculas.
`toUpperCase()`	Convierte una cadena en mayúsculas.

Los métodos del objeto `String` no respetan los estándares de la W3C (*World Wide Web Consortium*), organismo encargado de la estandarización de las tecnologías de Internet. Por este motivo, es importante prestar mucha atención al uso de estos métodos y, en muchos casos, es preferible el uso de un diseño basado en hojas de estilo en cascada (CSS).

En el siguiente ejemplo utilizamos algunos de los métodos que permiten establecer un formato sobre las cadenas de texto:

```
<script type="text/javascript">
  var texto = new String("Prueba de texto ");
  document.write("N\xfamero de letras de la cadena de
    texto: " + texto.length + "<br>")
  document.write("Cursiva: " + texto.italics() + "<br>");
  document.write("Negrita: " + texto.bold() + "<br>");
  document.write("Rojo: " + texto.fontcolor("#FF0000") +
    "<br>");
  document.write("Grande: " + texto.fontsize(20) + "<br>");
  document.write("Tachado: " + texto.strike() + "<br>");
</script>
```

ACTIVIDADES 3.1

En el apartado del objeto `Math` hemos utilizado un ejemplo para calcular el área de un círculo. Realice una aplicación similar pero que calcule el área de un triángulo en el cual la base y la altura las proporcione el usuario.

Recuerde que el área de un triángulo es igual a: (base * altura)/2.

3.2 INTERACCIÓN DE LOS OBJETOS CON EL NAVEGADOR

En la sección anterior hemos presentado algunos de los principales objetos predefinidos de JavaScript. Sin embargo, existen otro tipo de objetos que permiten el control del navegador en sí mismo. Estos objetos nos permiten controlar diferentes características del navegador como, por ejemplo, qué mensaje mostramos en la barra de estado o cómo crear nuevas ventanas.

A continuación presentaremos los principales objetos que permiten manipular y gestionar algunas de las características del navegador.

3.2.1 EL OBJETO NAVIGATOR

El objeto `Navigator` permite identificar las características de la plataforma sobre la cual se está ejecutando la aplicación web escrita con JavaScript. En concreto, permite conocer datos como el tipo de navegador que se está utilizando, su versión y el sistema operativo del usuario. Este objeto se suele utilizar para obtener este tipo de información y en base al resultado, tomar una decisión sobre qué tipo de código ejecutar. Esto se debe a que no todos los navegadores se comportan del mismo modo con el mismo código. Por ello, en algunos casos, es necesario adaptar algún fragmento de código para cada navegador.

Tabla 3.8 Propiedades del objeto *Navigator*

Propiedad	Descripción
appCodeName	Devuelve el código del nombre del navegador.
appName	Devuelve el nombre del navegador.
appVersión	Devuelve la versión del navegador.
cookieEnable	Determina si las *cookies* están habilitadas o no.
platform	Devuelve la plataforma sobre la cual se está ejecutando el navegador.
userAgent	Devuelve una información completa sobre el agente de usuario, el cual es normalmente el navegador.

Tabla 3.9 Métodos del objeto *Navigator*

Método	Descripción
javaEnable()	Informa si el navegador está habilitado para soportar la ejecución de programas escritos en Java.

En el siguiente ejemplo realizamos algunas consultas sobre el navegador, además de verificar si está preparado para la ejecución de *applets* de Java. En este caso simplemente imprimimos el resultado en la pantalla, pero en aplicaciones web avanzadas es posible realizar tareas más complejas en base a este resultado.

```
<script type="text/javascript">
  document.write("Navegador: " + navigator.appName +
    "<br>");
  document.write("Versi\xf3n: " + navigator.appVersion +
    "<br>");

  if (navigator.javaEnabled()){
    document.write("El navegador S\xcd est\xe1 preparado
      para soportar los Applets de Java")
  }
  else{
    document.write("El navegador NO est\xe1 preparado para
      soportar los Applets de Java")
  }
</script>
```

3.2.2 EL OBJETO SCREEN

El objeto `Screen` corresponde a la pantalla utilizada por el usuario. Este objeto cuenta con seis propiedades, aunque no posee ningún método. Todas sus propiedades son solamente de lectura, lo que significa que podemos consultar los valores de las propiedades del objeto, pero no las podemos modificar.

Tabla 3.10 Propiedades del objeto *Screen*

Propiedad	Descripción
`availHeight`	Corresponde a la altura disponible de la pantalla para el uso de ventanas.
`availWidth`	Corresponde a la anchura disponible de la pantalla para el uso de ventanas.
`colorDepth`	Corresponde al número de colores que puede representar la pantalla.
`height`	Corresponde a la altura total de la pantalla.
`pixelDepth`	Corresponde a la resolución de la pantalla expresada en bits por píxel.
`width`	Corresponde a la anchura total de la pantalla.

La altura y anchura disponibles para las ventanas es menor que la altura y anchura total de la pantalla, debido a que cada sistema operativo ocupa una parte de la pantalla con barras de tareas o menús.

El uso del objeto `Screen` y sus propiedades, es muy común en los diseñadores de páginas web que necesitan saber la resolución de la pantalla del usuario para poder adaptar sus diseños antes de cargarlos. Otro uso bastante común de este objeto, es consultar todas sus propiedades con el fin de adaptar la posición y tamaño de las ventanas emergentes que abra la aplicación web.

```
<script type="text/javascript">
  document.write("<br>Altura total: " + screen.height);
  document.write("<br>Altura disponible: " +
    screen.availHeight);
  document.write("<br>Anchura total: " + screen.width);
  document.write("<br>Anchura disponible: " +
    screen.availWidth);
  document.write("<br>Profundidad de color: " +
    screen.colorDepth);
</script>
```

3.2.3 EL OBJETO WINDOW

Como hemos mencionado anteriormente, el objeto `Window` se considera el objeto más importante del lenguaje JavaScript. A partir de él, podemos gestionar las ventanas del navegador y utilizar una serie de propiedades y métodos que presentamos en las siguientes tablas.

El objeto `Window` se considera como un objeto implícito, ya que no es necesario nombrarlo para acceder a los objetos que se encuentran en el nivel ubicado bajo su nivel de jerarquía. Por ejemplo, tal y como hemos realizado en alguna de las actividades propuestas en las secciones anteriores, si quisiésemos acceder a la propiedad que cambia el color de fondo de una página, deberíamos escribir `window.document.bgColor`. En los ejemplos anteriores hemos podido omitir la escritura del objeto `Window`, dado que JavaScript reconoce que todos los demás objetos están debajo de su nivel de jerarquía.

Tabla 3.11 Propiedades del objeto *Window*

Propiedad	Descripción
`closed`	Corresponde al valor booleano que indica si la ventana está cerrada o no.
`defaultStatus`	Corresponde a la cadena de texto de la barra de estado del navegador.
`document`	Corresponde al documento actual de la ventana.
`frames`	Corresponde al conjunto de marcos de la ventana.
`history`	Corresponde al conjunto de elementos que representan las URL visitadas.
`innerHeight`	Corresponde a la altura utilizable de la ventana.
`innerWidth`	Corresponde al ancho utilizable de la ventana.
`length`	Corresponde al número de `frames` de la ventana.
`location`	Corresponde a la URL de la barra de direcciones.
`locationbar`	Corresponde a la barra de direcciones del navegador.
`menubar`	Corresponde a la barra de menú del navegador.
`name`	Corresponde al nombre de la ventana.
`opener`	Corresponde a la referencia al objeto `Window` que haya abierto una ventana nueva.
`outerHeight`	Corresponde a la altura exterior de la página.
`outerWidth`	Corresponde al ancho exterior de la página.
`pageXoffset`	Corresponde a la posición horizontal de la ventana.
`pageYoffset`	Corresponde a la posición vertical de la ventana.
`parent`	Corresponde a la referencia al objeto `Window` que contiene los marcos de una página de marcos.
`personalbar`	Corresponde a la barra de herramientas personal.
`scrollbars`	Corresponde a las barras de desplazamiento vertical y horizontal.
`self`	Corresponde a la ventana actual.
`status`	Corresponde a la cadena con el mensaje que contiene la barra de estado.
`toolbar`	Corresponde a la barra de herramientas del navegador.
`top`	Corresponde a la ventana de nivel superior.

Todas estas propiedades podemos manipularlas con el fin de mostrar o no una parte de la ventana del navegador o modificar algunas características como su tamaño o posición. A lo largo del capítulo mostraremos ejemplos de cómo realizar estas operaciones. Algunos de los métodos del objeto `Window` ya los hemos usado en ejemplos o actividades de secciones anteriores. La mayor parte de estos métodos están relacionados con la gestión de las ventanas y la navegación de las páginas web.

Tabla 3.12 Métodos del objeto *Window*

Método	Descripción
`alert()`	Genera un cuadro de diálogo con un mensaje y un botón *Aceptar*.
`back()`	Regresa a la página anterior según el historial.
`blur()`	Desactiva una página.
`close()`	Cierra una página.
`confirm()`	Genera un cuadro de diálogo con los botones *Aceptar* y *Cancelar*.
`find()`	Realiza una búsqueda de texto en una página.
`focus()`	Activa una ventana.
`forward()`	Avanza una página según el historial.
`home()`	Carga la página definida como página por defecto del navegador.
`moveTo()`	Mueve la ventana activa.
`open()`	Abre una nueva ventana.
`print()`	Imprime una página.
`prompt()`	Genera un cuadro de diálogo con un cuadro de texto para que el usuario introduzca valores.
`resizeTo()`	Modifica el tamaño de una ventana.
`setinterval()`	Evalúa una expresión después de un tiempo especificado.
`setTimeOut()`	Inicia un registro de tiempo.
`scrollBy()`	Mueve el contenido de una ventana en función de una cantidad especificada en píxeles.
`scrollTo()`	Mueve el contenido de una ventana especificando una nueva posición de la esquina superior izquierda.
`stop()`	Detiene una página.

Algunos de los métodos ya los hemos utilizado en ejemplos anteriores del libro y otros más completos los presentaremos en los siguientes apartados dedicados a la gestión de las ventanas.

3.2.4 EL OBJETO DOCUMENT

El objeto `Document` se refiere a los documentos que se cargan en la ventana del navegador. Dado que con este objeto es posible manipular las propiedades y el contenido de los elementos principales de las páginas web, probablemente sea el objeto más utilizado en los programas escritos en JavaScript.

El objeto `Document` cuenta con una serie de subobjetos como son los vínculos, puntos de anclaje, imágenes o formularios. Esta colección de subobjetos representa el verdadero potencial del objeto `Document`. A continuación presentamos las principales propiedades y métodos de dicho objeto.

Tabla 3.13 Propiedades del objeto *Document*

Propiedad	Descripción
`alinkColor`	Corresponde al color de los vínculos activos de la página.
`anchors`	Corresponde a los puntos de anclaje (etiquetas `<a name>`) del documento.
`applets`	Corresponde a los *applets* (etiquetas `<applet>`) Java del documento.
`bgColor`	Corresponde al color de fondo del documento.
`cookie`	Corresponde a un fichero guardado en el equipo del cliente del navegador con información sobre el usuario.
`domain`	Corresponde al nombre del dominio por defecto para el documento.
`embeds`	Corresponde a los objetos embebidos (etiqueta `<embed>`) en el documento.
`fgColor`	Corresponde al color del texto del documento.
`forms`	Corresponde a los formularios (etiqueta `<form>`) del documento.
`images`	Corresponde a las imágenes (etiqueta `<images>`) del documento.
`lastModified`	Corresponde a la última fecha en la que se modificó el documento.
`layers`	Corresponde a las capas (etiqueta `<layer>`) del documento.
`linkColor`	Corresponde al color de los enlaces aún no visitados.
`links`	Corresponde a los vínculos (etiqueta `<a href>`) del documento.
`plugins`	Corresponde a las referencias y llamadas de los plugins del documento.
`referrer`	Corresponde a la dirección del documento usado para ir al documento actual.
`title`	Corresponde al título (etiqueta `<title>`) del documento.
`URL`	Corresponde a la dirección del documento.
`vlinkColor`	Corresponde al color de los enlaces visitados.

La mayor parte de estas propiedades contiene una lista con cada uno de los elementos presentes en el documento HTML. Una vez que se obtiene la lista, es posible acceder a cada elemento para gestionar o modificar cada uno de ellos. El objeto `Document` tiene diferentes métodos bastante útiles, que resumimos en la Tabla 3.14.

Tabla 3.14 Métodos del objeto *Document*

Método	Descripción
`captureEvents()`	Intercepta un evento para que pueda ser manipulado por el documento.
`close()`	Cierra el documento activo.
`getSelection()`	Devuelve el texto seleccionado en el documento.
`handleEvent()`	Activa el manejador del evento especificado.
`home()`	Carga la página de inicio.
`open()`	Activa el documento.
`releaseEvents()`	Libera los eventos que han sido interceptados.
`routeEvents()`	Intercepta un evento y lo pasa a lo largo de la jerarquía del objeto que lo lanzó.
`write()`	Escribe datos en el documento.
`writeln()`	Escribe datos además de un salto de línea en el documento.

3.2.5 EL OBJETO HISTORY

El objeto `History` almacena las referencias de todos los sitios web visitados. Estas referencias se guardan en una lista y sus propiedades y métodos se utilizan principalmente para que el usuario de una aplicación web pueda desplazarse para adelante y para atrás. Sin embargo, al ser una lista de referencias, no podemos acceder a los nombres de las direcciones URL visitadas, ya que son información privada del usuario.

Tabla 3.15 Propiedades del objeto *History*

Propiedad	Descripción
`current`	Corresponde a la cadena que contiene la URL de la entrada actual del historial.
`length`	Corresponde al número de páginas que han sido visitadas.
`next`	Corresponde a la cadena que contiene la siguiente entrada del historial.
`previous`	Corresponde a la cadena que contiene la anterior entrada del historial.

Tabla 3.16 Métodos del objeto *History*

Método	Descripción
`back()`	Carga la URL del documento anterior del historial.
`forward()`	Carga la URL del documento siguiente del historial.
`go()`	Carga la URL del documento especificado por el índice que pasamos como parámetro dentro del historial.

Por ejemplo, en el fichero HTML que queramos usar este objeto, podríamos crear dos botones para que el usuario pueda desplazarse adelante o atrás según su historial de navegación.

```
<form>
  <input type="button" value="Atras"
    onClick="history.back();">
  <input type="button" value="Adelante"
    onClick="history.forward;">
</form>
```

3.2.6 EL OBJETO LOCATION

El objeto `Location` corresponde a la URL de la página web en uso. Sus principales funciones son las de consultar las diferentes partes que forman una URL, como por ejemplo el dominio, el protocolo o el puerto. De este modo, podemos extraer cada componente de la URL y trabajar con ellos de forma separada. Gracias a este objeto podemos recargar una página, cargar una nueva o remplazar una por otra.

Tabla 3.17 Propiedades del objeto *Location*

Propiedad	Descripción
`hash`	Corresponde a la cadena que representa el anclaje de la URL. Es decir, la parte de la URL que va después de la etiqueta #.
`host`	Corresponde a la cadena que representa el nombre del dominio del servidor y el número del puerto dentro de la URL.
`hostname`	Corresponde a la cadena que representa el nombre del dominio del servidor dentro de la URL.
`href`	Corresponde a la URL completa.
`pathname`	Corresponde a la cadena que sigue al nombre del servidor.
`port`	Corresponde al número de puerto de la URL.
`protocol`	Corresponde al protocolo utilizado por la página web.
`search`	Corresponde a la cadena de búsqueda que se encuentra después de un signo de interrogación de la URL.

Tabla 3.18 Métodos del objeto *Location*

Método	Descripción
`assign()`	Carga un nuevo documento.
`reload()`	Carga de nuevo el documento actual.
`replace()`	Sustituye la URL del documento actual por otra URL.

En el siguiente ejemplo utilizamos tres de las propiedades del objeto `Location`, además de crear un botón que al presionarlo, llama al método `location.reload()`, con lo cual cargamos de nuevo el documento actual. Este botón podría ser una alternativa al botón *refresh* de un navegador:

```
<script type="text/javascript">
  document.write("URL completa: " + location.href +
    "<br>");
  document.write("Pathname: " + location.pathname +
    "<br>");
  document.write("Protocolo: " + location.protocol +
    "<br>");
</script>
<form>
  <input type="button" value="Recargar"
    onclick="location.reload();">
</form>
```


En el ejemplo anterior hemos utilizado un controlador de eventos llamado `onclick`. Este tipo de controladores son un enlace entre los objetos del documento y el código JavaScript. La descripción y análisis de los eventos los estudiaremos más adelante.

ACTIVIDADES 3.2

- En un documento HTML cree un botón que al presionarlo, utilice el método `resizeTo(500, 500)` para modificar el tamaño de la ventana.

3.3 GENERACIÓN DE ELEMENTOS HTML DESDE CÓDIGO JAVASCRIPT

Uno de los principales objetivos de JavaScript en el desarrollo web en la parte del cliente es convertir un documento HTML estático en una aplicación web dinámica. Por ejemplo, es muy común que los *scripts* detecten el tipo y la versión del navegador que estamos utilizando y, en base a esto, escribir las etiquetas adecuadas para cada navegador. De igual modo, se suelen utilizar los valores de determinadas variables para ejecutar instrucciones que creen nuevas ventanas con contenido propio, en lugar de mostrarlo en la ventana actual. Cada ventana de un navegador presenta un documento HTML y es representada por un objeto `Window` que contiene un subobjeto `Document`. El objeto `Document` contiene a su vez una serie de objetos que representan todo el contenido del documento HTML, como por ejemplo el texto, las imágenes, los enlaces, los formularios, las tablas, etc.

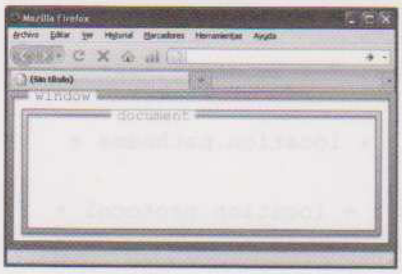


***Figura 3.2.** Objetos `Window` y `Document` representados en un navegador*

Con JavaScript es posible manipular y acceder a los objetos que representan el contenido de una página. De este modo, en lugar de crear solamente documentos estáticos, es posible crear documentos dinámicos. Este proceso se puede realizar gracias al método `document.write()`.

En ejemplos anteriores, hemos visto que con el método `document.write()` podemos escribir un resultado por pantalla. Existen dos formas de utilizar este método para generar contenido dinámico:

- Podemos utilizarlo dentro de una secuencia de instrucciones JavaScript para mostrar un resultado en el documento de la ventana actual del navegador. Por ejemplo, es posible definir el título de una página web basándose en el nombre del sistema operativo que se utilice para abrir el documento:

```
<script type="text/javascript">
  var SO = navigator.platform;
  document.write("<h1>Documento abierto con: " + SO +
  "</h1>");
</script>
```

- La segunda forma es muy similar a la anterior. El método `document.write()` podemos utilizarlo para crear documentos de nuevas ventanas del navegador. Esto es una práctica muy común en las páginas web que utilizan ventanas emergentes. Los detalles de la creación y la comunicación entre ventanas los abordaremos en otro apartado de este capítulo. Por el momento, en el siguiente código podemos observar cómo creamos una

nueva ventana sin ningún contenido, posteriormente accedemos a su respectivo documento y, finalmente, en la ventana nueva escribimos el título de la página web según el texto que ha ingresado el usuario.

```
<script type="text/javascript">
  var texto = prompt("Ingresa un titulo para la nueva
    ventana: ");
  var ventanaNueva = window.open();
  ventanaNueva.document.write("<h1>" + texto + "</h1>");
</script>
```

- La generación de código HTML a partir de código JavaScript no se limita solo a la creación de texto tal y como hemos visto en los ejemplos anteriores. Podemos crear y manipular todo tipo de objetos. El siguiente ejemplo muestra cómo generar un formulario para modificar la propiedad del color de fondo de la página:

```
<script type="text/javascript">
  document.write("<form name=\"cambiacolor\">");
  document.write("<b>Selecciona un color para el fondo de
    página:</b><br>");
  document.write("<select name=\"color\">");
  document.write("<option value=\"red\">Rojo</option>");
  document.write("<option value=\"blue\">Azul</option>");
  document.write("<option
    value=\"yellow\">Amarillo</option>");
  document.write("<option value=\"green\">Verde</option>");
  document.write("</select>");
  document.write("<input type=\"button\"
    value=\"Modifica el color\" onclick=\"document.bgColor
    =document.cambiacolor.color.value\">");
  document.write("</form>");
</script>
```

En la Figura 3.3 vemos el resultado de la generación de una página web dinámica. En esta página el usuario puede modificar la propiedad `document.bgColor` a través de un formulario HTML que se ha creado a partir de código JavaScript.

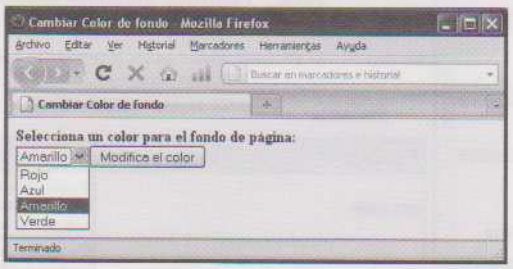


Figura 3.3. *Generación de código HTML con JavaScript. Cambio de color*

ACTIVIDADES 3.3

Abra el fichero `3.11-GeneraciónCódigo3.html` y modifique el ejemplo anterior agregando dos colores más al conjunto de posibles colores del fondo de la página.

3.4 APLICACIONES PRÁCTICAS DE LOS MARCOS

La mayor parte de las aplicaciones web se crean para ser ejecutadas en una sola ventana, aunque existe la posibilidad de dividir la ventana en dos o más partes independientes. En estos sectores podemos utilizar JavaScript para generar una interacción entre ellos. Estos sectores se denominan marcos. Algunas páginas web presentan una estructura en la cual una parte de la página permanece fija mientras que otra parte va cambiando en base a la navegación que se efectúa en otro marco. En realidad este efecto se produce creando diferentes páginas web y posicionando cada una de ellas en un marco diferente.

La Figura 3.4 muestra una página web que utiliza tres marcos. Esta es una página bastante conocida para aquellos que hayan realizado algún curso de programación en Java, ya que ésta es la página con la especificación API de este lenguaje. En el marco superior izquierdo se definen los paquetes de Java. Debajo de este último marco encontramos las clases, interfaces y/o excepciones pertenecientes a cada paquete y, por último, en el marco principal de la derecha, encontramos la descripción detallada de cada concepto del lenguaje. Estos tres marcos interactúan entre ellos. Por ejemplo, al elegir un paquete en el primer marco, el segundo marco cambia automáticamente para mostrar todas las clases relacionadas con dicho paquete. Si seleccionamos una de estas clases, el marco principal mostrará la descripción de dicha clase, con sus métodos, constructores, etc.

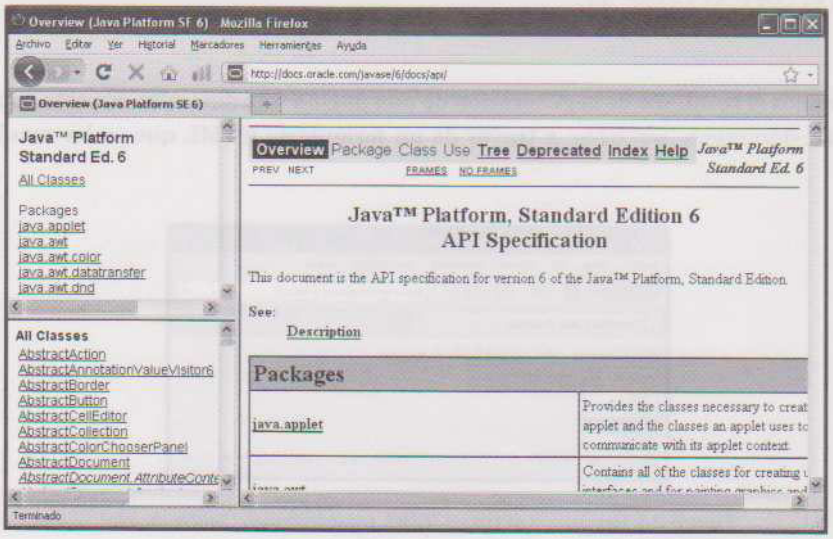


Figura 3.4. Página web definida con marcos HTML

Los marcos se definen utilizando HTML. Estos presentan la ventaja de poder crear páginas en las que es posible mantener constantemente algunos elementos como botones, enlaces, imágenes, etc. Con JavaScript podemos manipular los marcos y realizar una interacción entre ellos. De este modo es posible utilizar todas las ventajas del lenguaje sobre los marcos.

Para utilizar correctamente los marcos de HTML, necesitamos conocer un par de etiquetas con sus respectivos atributos:

- `<frameset>`: las funciones básicas de esta etiqueta son indicar al navegador que la página contiene marcos, definir el número de marcos que se utilizarán en la página y establecer el tamaño de cada marco. Los principales atributos son `cols` y `rows`, los cuales dividen respectivamente el conjunto de marcos vertical u horizontalmente.
- `<frame>`: esta etiqueta define las características de cada marco. En la Tabla 3.19 vemos sus principales atributos.

Tabla 3.19 Atributos de la etiqueta `<frame>`

Atributo	Descripción
`frameborder`	Define si mostrar o no el borde del marco.
`marginheight`	Permite cambiar los márgenes verticales del marco.
`marginwidth`	Permite cambiar los márgenes horizontales del marco.
`name`	Asigna un nombre al marco.
`noresize`	Evita que el usuario pueda modificar el tamaño del marco.
`scrolling`	Permite elegir si posiciona o no una barra de desplazamiento en el marco.
`src`	Indica la URL del documento HTML que contendrá el marco.

Todas las páginas que se dividan en al menos dos marcos, necesitarán al menos tres páginas web. La primera página web contendrá el primer marco que se denomina conjunto de marcos. Este conjunto es el que creamos utilizando la etiqueta `<frameset>` y contendrá las otras páginas web que se mostrarán en cada marco definido. Cada uno de los marcos definidos los creamos con la etiqueta `<frame>` y se considera un hijo del conjunto de marcos.

3.4.1 USO DE MARCOS CON JAVASCRIPT

Tal y como hemos mencionado anteriormente, JavaScript permite manipular los marcos. Cualquier marco puede hacer referencia a otro marco utilizando las propiedades `frames`, `parent` y `top` del objeto `Window`. Cada ventana tiene una propiedad llamada `frames`, la cual hace referencia a una lista que contiene todos los marcos.

El código JavaScript que ejecutamos en un conjunto de marcos puede hacer referencia a su primer marco a través del objeto `frames[0]`, a su segundo marco con `frames[1]`, y así sucesivamente.

Todos los marcos poseen una propiedad llamada `parent`, que hace referencia al objeto que contiene dichos marcos. Por lo tanto, es posible hacer referencia a un marco desde otro. Para ello, empleamos la propiedad `parent` seguida del nombre del marco al cual queremos acceder y, por último, el nombre del elemento que queramos manipular. Por ejemplo, debemos escribir el siguiente código si desde un marco queremos modificar el color de fondo de otro marco llamado `MarcoDePrueba`:

```
parent.MarcoDePrueba.document.bgColor
```

A continuación presentamos un ejemplo de un documento HTML en el cual hemos definido dos marcos divididos por columnas. El atributo `cols` lo hemos utilizado para que cada marco ocupe el 50% de la ventana:

```
<html>
  <head>
    <title>Ejemplos de control de marcos</title>
  </head>
    <frameset cols="50%,50%">
      <frame src="3.12a-Marcos1.html" name="Marco1" noresize>
      <frame src="3.12b-Marcos2.html" name="Marco2" noresize>
  </frameset>
  <body></body>
</html>
```

El primer marco (`Marco1`) contiene la página `3.12a-Marcos1.html`. En esta página hemos definido dos campos de selección con la etiqueta `<select>`, con el fin de elegir un color y uno de los dos marcos presentes.

```
<html>
  <body>
    <form name="form1">
      <select name="color">
        <option value="green">Verde
        <option value="blue">Azul
      </select>
      <br><br>
      <select name="marcos">
        <option value="0">Izquierda
        <option value="1">Derecha
      </select>
    </form>
  </body>
</html>
```

El segundo marco contiene la página `3.12b-Marcos2.html`. En esta página hemos agregado un botón con el controlador de eventos `onclick`, en el que hemos escrito un guión JavaScript que permite acceder a cada marco y modificar su color de fondo en base a las elecciones del primer marco.

```
<html>
  <body>
    <form>
      <input type="Button" value="Cambiar Color" onclick="
        campoColor = parent.Marco1.document.form1.color
        if(campoColor.selectedIndex==0){
          colorin = 'green';
        }else{
          colorin = 'blue';
        }
        campoFrame = parent.Marco1.document.form1.marcos
        if(campoFrame.selectedIndex==0){
          window.parent.Marco1.document.bgColor = colorin
        }else{
          window.parent.Marco2.document.bgColor = colorin
        }">
    </form>
  </body>
</html>
```

En el ejemplo anterior hemos utilizado una sentencia condicional para verificar el color elegido y el marco sobre el cual se aplicará dicho color. Existe una forma mucho más rápida y limpia de realizar este proceso gracias al uso de *arrays* y llamadas a funciones. Estos conceptos los veremos en el próximo capítulo del libro.

Podemos ver el resultado de la creación de estos marcos en la Figura 3.5.

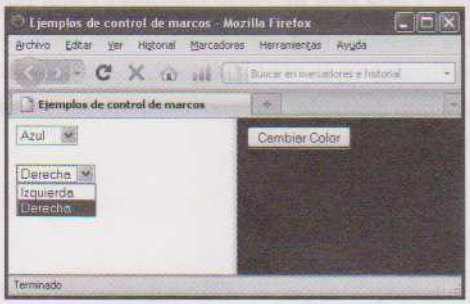


Figura 3.5. *Interacción entre marcos*

Una opción interesante es la de ocultar el borde de los marcos y crear páginas en las que no se note esa separación. El resultado aparece como una página web única, aunque en realidad esté dividida por varias páginas. El borde lo podemos ocultar definiendo los atributos `frameborder` y `border` iguales a cero.

El desarrollo de páginas web con múltiples marcos era una práctica común en la primera época de desarrollo con JavaScript. Actualmente, la mayoría de desarrolladores no utiliza el desarrollo de ventanas interactivas, aunque sí que es común el desarrollo con marcos en línea los cuales se denominan *iframes*.

ACTIVIDADES 3.4

- Modifique el código del ejemplo de marcos utilizado anteriormente, con el fin de ocultar el borde y que no se note la separación entre ellos.

3.5 GESTIÓN DE LAS VENTANAS

Hasta el momento, en los capítulos anteriores hemos visto cómo utilizar JavaScript para gestionar principalmente el contenido de las páginas web. En esta sección estudiaremos cómo gestionar los diferentes aspectos de las ventanas del navegador utilizando JavaScript.

Existen muchas operaciones comunes en las páginas web que visitamos a diario. Por ejemplo, es común que se abran ventanas nuevas al presionar un botón. Cada una de estas ventanas tiene tamaños, posiciones y estilos diferentes. Muchas de estas ventanas emergentes no contienen simplemente una página web estática. Su contenido suele ser dinámico y depende de las acciones realizadas en la ventana que la haya creado. En los siguientes apartados veremos cómo abrir y cerrar nuevas ventanas, cómo controlar la apariencia de las mismas y cómo crear una comunicación e interacción entre dos o más ventanas.

3.5.1 ABRIR Y CERRAR NUEVAS VENTANAS

Algo muy común que ocurre cuando navegamos por la Web es que se abra una ventana nueva cuando pulsamos un botón. Algunos sitios abren ventanas incluso sin que el usuario haga algo. Las ventanas se abren automáticamente cuando se carga una nueva página o simplemente cuando cerramos otra ventana. Tal y como hemos visto anteriormente en la descripción de los objetos de JavaScript, la ventana del navegador es un objeto más que presenta una serie de propiedades y métodos.

Las ventanas secundarias las podemos abrir utilizando solo código HTML mediante el atributo `target` de la etiqueta `href`:

```
<a href="enlace.html" target="_blank">
```

La desventaja de abrir una nueva ventana utilizando solo código HTML es que no tenemos ningún control sobre ella. La ventana se abrirá en base a la configuración predefinida del navegador del usuario. Por el contrario, con JavaScript tenemos un mayor control sobre las nuevas ventanas. Una nueva ventana vacía se crea con el método `open()`, el cual devuelve una referencia a dicha ventana:

```
nuevaVentana = window.open()
```

De este modo la variable llamada `nuevaVentana` contendrá una referencia a la ventana que hemos creado. Esta variable la podemos utilizar posteriormente para cualquier manipulación que queramos hacer sobre la ventana mientras se ejecuta JavaScript.

Las ventanas poseen diversas propiedades y métodos tal y como hemos visto en las secciones anteriores. En concreto, el método `open()` cuenta con los siguientes tres parámetros:

- El primero es la URL de la página web que estará contenida en la nueva ventana.
- El segundo parámetro es el nombre asignado a la nueva ventana.
- El tercer parámetro es una colección de atributos que definen la apariencia de la ventana.

De este modo, el método `open()` lo podemos utilizar definiendo estos tres parámetros, tal y como vemos en el siguiente ejemplo:

```
nuevaVentana = window.open("http://www.misitioWeb.com/ads",
  "Publicidad", "height=100, widht=100");
```

Utilizando los anteriores parámetros, abrimos una ventana que contiene la URL y el nombre especificados en los dos primeros parámetros y tiene una altura y anchura de 100 píxeles.

A continuación presentamos un ejemplo completo de una página web en la que creamos un botón que abre una nueva ventana al hacer clic sobre él. En esta nueva ventana establecemos los atributos de altura y anchura, además de crear etiquetas HTML para generar el título de la ventana y un texto en el que especificamos las propiedades modificadas.

```
<html>
  <head></head>
  <body>
    <center><h1> Ejemplo de Apariencia de una Ventana</h1>
    <br>
    <input type="Button" value="Abre una Ventana" onclick="
      myWindow1=window.open('', 'Nueva Ventana', 'width=300,
      height=200');
      myWindow1.document.write('<html>');
      myWindow1.document.write('<head>');
```

```
        myWindow1.document.write('<title>Ventana de Prueba
          </title>');
        myWindow1.document.write('</head>');
        myWindow1.document.write('<body>');
        myWindow1.document.writeln('Se han utilizado las
          propiedades: ');
        myWindow1.document.write('<li>height=200</li>
          <li>width=300</li>');
        myWindow1.document.write('</body>');
        myWindow1.document.write('</html>');
      "/>
    </center>
  </body>
</html>
```

Cualquiera de las ventanas que abramos, podemos igualmente cerrarlas al invocar el método `close()`. Como hemos dicho anteriormente, el método `open()` devuelve una referencia a una nueva ventana. Para cerrar la ventana debemos invocar el método `close()` sobre dicha referencia.

Para probar su funcionamiento, podemos utilizar el código del ejemplo anterior y agregar justo antes de la creación de la etiqueta `</body>` la línea:

```
myWindow1.document.write('<input type=button value=Cerrar
  onClick=window.close()>');
```

La apertura de múltiples ventanas a la vez es una característica por lo general bastante molesta para los usuarios. Sin embargo, en algunos casos puede ser útil saber cómo realizar esta operación. Para ello utilizamos un bucle `for` con el fin de ejecutar el método `window.open()` todas las veces deseadas. En el siguiente ejemplo abrimos cinco ventanas a la vez que presentan a su vez un botón para poder cerrarlas:

```
<html>
  <head></head>
  <body>
    <center><h1>Apertura de m&uacute;ltiples ventanas</h1>
      <input type="Button" value="Abre m&uacute;ltiples
        ventanas..." onclick="
        for(i=0;i<5;i++){
          myWindow=window.open('','','width=200,height=200');
          myWindow.document.write('<html>');
          myWindow.document.write('<head>');
          myWindow.document.write('<title>Ventana: '+i+'
            </title>');
          myWindow.document.write('</head>');
```

```
            myWindow.document.write('<body>');
            myWindow.document.write('Ventana ' + i);
            myWindow.document.write('<input type=button
              value=Cerrar onClick=window.close()>');
            myWindow.document.write('</body>');
            myWindow.document.write('</html>');
          }
        "/>
      </center>
    </body>
  </html>
```

En la Figura 3.6 podemos ver el resultado de la ejecución del ejemplo anterior, en el cual al presionar un botón en la ventana principal, generamos cinco ventanas emergentes, cada una de ellas con un botón para cerrar cada ventana.

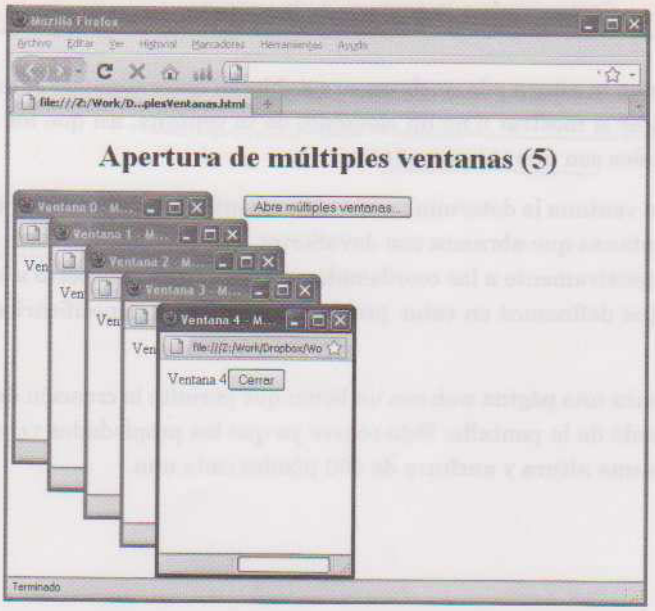


Figura 3.6. Apertura de múltiples ventanas

3.5.2 APARIENCIA DE LAS VENTANAS

Las ventanas cuentan con propiedades que podemos modificar con JavaScript. Estas propiedades permiten decidir el tamaño, la ubicación o los elementos que tendrá la ventana. La Tabla 3.20 contiene las principales propiedades que definen la apariencia de la ventana.

```
            myWindow.document.write('<body>');
            myWindow.document.write('Ventana ' + i);
            myWindow.document.write('<input type=button
              value=Cerrar onClick=window.close()>');
            myWindow.document.write('</body>');
            myWindow.document.write('</html>');
          }
        "/>
      </center>
    </body>
  </html>
```

En la Figura 3.6 podemos ver el resultado de la ejecución del ejemplo anterior, en el cual al presionar un botón en la ventana principal, generamos cinco ventanas emergentes, cada una de ellas con un botón para cerrar cada ventana.

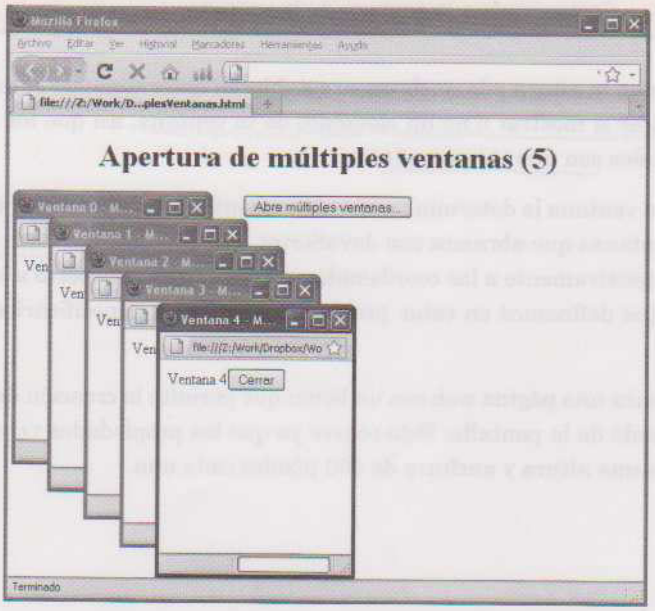


Figura 3.6. Apertura de múltiples ventanas

3.5.2 APARIENCIA DE LAS VENTANAS

Las ventanas cuentan con propiedades que podemos modificar con JavaScript. Estas propiedades permiten decidir el tamaño, la ubicación o los elementos que tendrá la ventana. La Tabla 3.20 contiene las principales propiedades que definen la apariencia de la ventana.

Tabla 3.20 Propiedades relacionadas con la apariencia de la ventana

Propiedad	Descripción
directories	Corresponde a los botones del directorio estándar del navegador.
height	Corresponde a la altura de la ventana.
menubar	Corresponde a la barra del menú.
resizable	Corresponde a la opción de cambiar o no el tamaño de la ventana.
scrollbars	Corresponde a las barras de desplazamiento.
status	Corresponde a la barra de estado.
toolbar	Corresponde a la barra de herramientas.
width	Corresponde a la anchura de la ventana.

Las propiedades que definen la altura y la anchura se establecen determinando el tamaño en píxeles. Las demás propiedades sirven para decidir si mostrar o no un elemento de la ventana, así que los posibles valores que pueden tomar estas últimas propiedades son uno (`1`) o cero (`0`).

La ubicación de una nueva ventana la determina automáticamente el navegador. Sin embargo, podemos establecer la ubicación exacta de las ventanas que abramos con JavaScript. Las propiedades `left` y `top` permiten definir esta ubicación y corresponden respectivamente a las coordenadas *x* e *y* en píxeles respecto a la esquina superior izquierda de la ventana. Los valores que definamos en estas propiedades debemos especificarlos en el tercer parámetro del método `open()`.

El siguiente ejemplo muestra una página web con un botón que permite la creación de una nueva ventada ubicada en la esquina superior izquierda de la pantalla. Esto ocurre ya que las propiedades `top` y `left` las ponemos iguales a cero. Además, establecemos una altura y anchura de 400 píxeles cada una.

```
<html>
  <head></head>
  <body>
    <center><h1> Apariencia de las ventanas</h1>
      <br>
      <input type="Button" value="Abre una ventana" onclick="
        myWindow1=window.open('','Nueva Ventana','top=0,
          left=0,width=400, height=400')
        myWindow1.document.write('<html>')
        myWindow1.document.write('<head>')
        myWindow1.document.write('<title>Ubicar una ventana
          </title>')
        myWindow1.document.write('</head>')
        myWindow1.document.write('<body>')
```

```
        myWindow1.document.write('top=0 <br> left=0 <br>
          width=400 <br> height=400')
        myWindow1.document.write('</body>')
        myWindow1.document.write('</html>')
      "/>
    </center>
  </body>
</html>
```

Debido a las diferentes resoluciones de pantalla de los usuarios, no siempre se consigue el efecto deseado cuando definimos la ubicación de las nuevas ventanas. Para evitar este inconveniente, podemos especificar posiciones relativas utilizando el objeto `Screen` y averiguar previamente la resolución de la pantalla con las propiedades `screen.height` y `screen.width`. Una vez conozcamos los datos de la resolución, podemos utilizar porcentajes de estos mismos para definir las propiedades `left` y `top`.

3.5.3 COMUNICACIÓN ENTRE VENTANAS

En apartados anteriores del libro hemos visto cómo desde una ventana se pueden abrir o cerrar nuevas ventanas. La primera se denomina ventana principal, mientras que las segundas se denominan ventanas secundarias. La comunicación e interacción entre estas ventanas no es un proceso del todo bidireccional. Por ejemplo, una ventana principal puede abrir y cerrar la secundaria, pero por motivos de seguridad, no es posible realizar lo contrario.

Las nuevas ventanas se declaran como objetos y se les asigna un nombre. Gracias a este nombre, desde la ventana principal podemos acceder a ella y establecer una comunicación e interacción con sus elementos y propiedades, además de aplicar métodos de JavaScript, tal y como hemos visto anteriormente con el método `close()`. La ventana secundaria tiene el atributo `window.opener`, el cual hace referencia a la ventana principal. De este modo, desde la ventana secundaria podemos acceder a los métodos y propiedades de la ventana principal.

En el siguiente ejemplo mostramos cómo acceder a la propiedad `location` del objeto `Window` de una ventana secundaria. Esta propiedad contiene la URL del documento activo. La ventana principal cuenta con un formulario que presenta un campo de texto en el que podemos escribir una URL, además de un botón que al presionarlo cambia la propiedad `location` de la ventana secundaria. Esta última ventana mostrará la URL que haya escrito el usuario.

```
<html>
  <head></head>
  <body>
    <script>
      var ventanaSecundaria = window.open("", "ventanaSec",
        "width=500, height=500");
    </script>
    <center><h1> Comunicaci&oacute;n entre ventanas </h1>
      <br>
      <form name=formulario>
        <input type=text name=url size=50
          value="http://www.">
```

```
        <input type=button value="Mostrar URL en ventana
          secundaria" onclick=" ventanaSecundaria.location =
          document.formulario.url.value;">
      </form>
    </center>
  </body>
</html>
```

ACTIVIDADES 3.5

- Abra el fichero `Actividad-3.5a-ComunicaciónVentanas.html` y observe que desde la ventana secundaria se envía información a la principal. Modifique este fichero para que el envío sea bidireccional. Es decir, que la información que se escriba en la ventana principal, se envíe a la ventana secundaria.

RESUMEN DEL CAPÍTULO

En este capítulo hemos presentado una completa descripción de los principales objetos nativos del lenguaje JavaScript. Además de estos objetos, hemos descrito otros que están relacionados directamente con las características y funcionamiento del navegador. De cada uno de estos objetos hemos mostrado sus principales propiedades y métodos.

A continuación, hemos expuesto el modo de generar elementos HTML desde código JavaScript. Además, hemos introducido el uso de los marcos HTML y la posibilidad de crear una interacción entre ellos gracias a JavaScript. Estos dos últimos temas nos permiten comprender los primeros pasos para poder generar páginas web dinámicas.

Por último, hemos introducido la gestión de las ventanas de un navegador. JavaScript permite abrir y cerrar una o más ventanas, modificar su apariencia gráfica y establecer una comunicación entre ellas.

EJERCICIOS PROPUESTOS

1. Cree una página web con un *script* que calcule los milisegundos que han pasado desde las 00:00 del 1 de enero del 2000 hasta la fecha actual.

2. Cree una página web que solicite una cadena de texto al usuario y devuelva la longitud de dicha cadena.

3. Cree un *script* que recoja el valor de la anchura y la altura total de la pantalla del usuario y calcule su diagonal. Recuerde: $diagonal = \sqrt{anchura^2 + altura^2}$.

4. Cree una página web que establezca las siguientes propiedades del objeto `Document`: color del texto = blanco, color de fondo = gris y título del documento = "Modificaciones del objeto `Document`".

TEST DE CONOCIMIENTOS

1 ¿El método `window.open()` necesita argumentos para poder abrir una nueva ventana?
- a) Verdadero.
- b) Falso.

2 ¿Qué propiedades se deben establecer para controlar el tamaño de una ventana?
- a) `top` y `left`.
- b) `frame` y `frameborder`.
- c) `width` y `height`.
- d) `resizable` y `size`.

3 El número devuelto al utilizar el método `getTime()` del objeto `Date` corresponde a:
- a) Días.
- b) Segundos.
- c) Milisegundos.
- d) Horas.

4 Se puede obtener el valor de la constante Pi mediante:
- a) `Math.3,14`.
- b) `Math.pi()`.
- c) `Math.PI`.
- d) Ninguna de las anteriores.

5 Se puede crear una ventana que no contenga las barras de desplazamiento si se establece:
- a) `toolbar = '0'`.
- b) `scrollbar = '1'`.
- c) `scrollbar = '0'`.
- d) `toolbar = '1'`.

6 ¿Qué método del objeto `Window` sirve para solicitar un valor al usuario?
- a) `alert()`.
- b) `prompt()`.
- c) `open()`.
- d) `print()`.

7 ¿Cómo se puede ocultar el borde de un marco?
a) `frameborder = 'hide'`.
b) `<frame> = '0'`.
c) `frame = '1'`.
d) `frameborder = '0'`.

8 ¿Cómo se define el número de marcos presente en un conjunto de marcos?
a) Definiendo el valor `frame`.
b) Definiendo el valor `frameset`.
c) Definiendo el valor `row`.
d) Definiendo el valor `cols`.

Nº marcos definido en frameset

9 ¿Cómo se cierra una ventana primaria desde una venta secundaria?
a) Con el método `window.close()`.
b) No se puede cerrar una ventana primaria desde una secundaria.
c) Con `opener.close()`.
d) Con `onclick = "close()"`.

10 ¿La comunicación entre ventanas es un proceso completamente bidireccional?
a) Verdadero.
b) Falso.

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 ISO[3]. Si por ejemplo quisiéramos 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>
```


`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 ±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. 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>
```

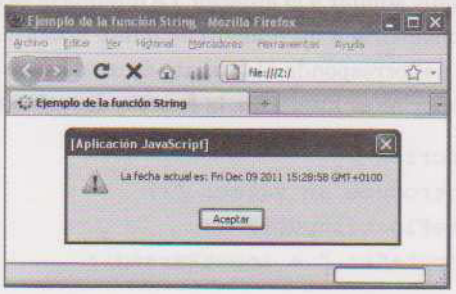


Figura 4.1. Ejemplo de la función `String()`

- **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>
```

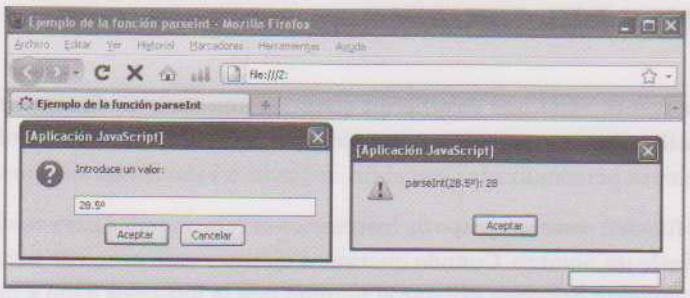


Figura 4.2. Ejemplo de la función `parseInt()`

- **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()`.

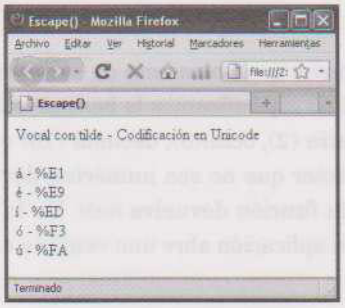


Figura 4.3. *Codificación de las vocales con tilde*

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 invocamos 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:

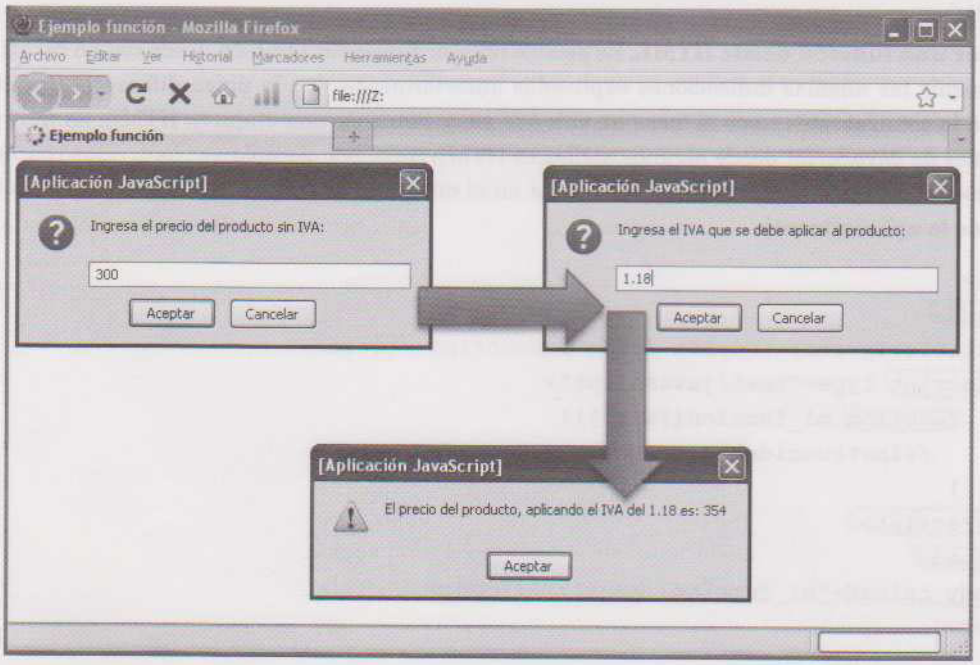


Figura 4.4. *Ejemplo de función que recibe dos parámetros del usuario*

Existen diferentes formar 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>
```

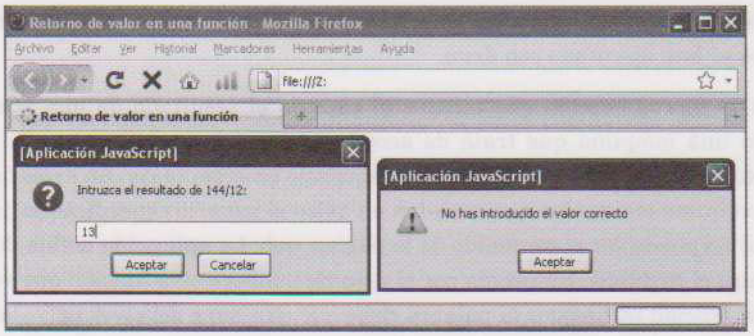


Figura 4.5. *Función que emplea el retorno de un valor*

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 quisiéramos 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 quisiéramos 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 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 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 quisiéramos 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>");
}
```

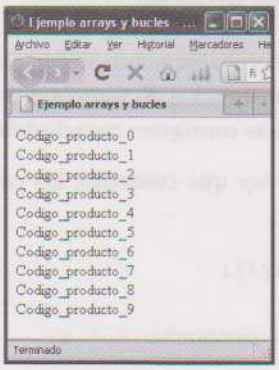


Figura 4.6. *Uso de arrays en bucles*

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: `lenght` y `prototype`. La primera propiedad, `lenght`, 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>");
}
```

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_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("Extensión de las páginas gubernamentales: "
  + paginas_gubernamentales.dominio);
```

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

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 quisiéramos 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("Pekín");
  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 quisiéramos 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");
  document.write(coches);
</script>
```

1º) posición en la cual añadiremos ó eliminaremos
2º) cuántos eliminaremos
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 Google[4].

```
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("</tr>")
}
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.

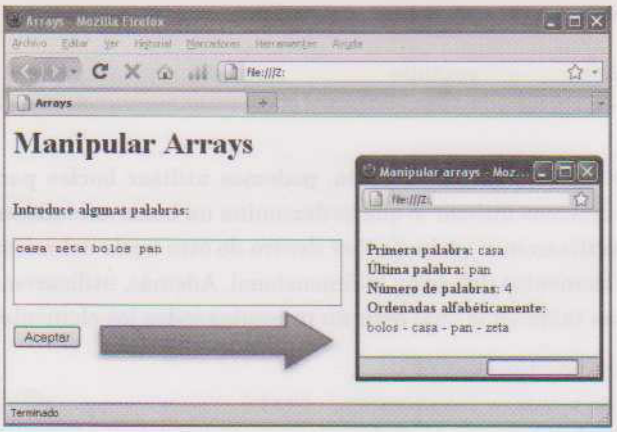


Figura 4.7. *Manipulación de arrays*

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 + " - A&ntilde;o
    de fabricaci&oacute;n: " + coches[i].anyo + "<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.

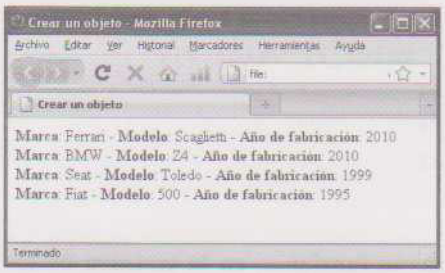


Figura 4.8. *Ejemplo de creación de objetos personalizados*

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("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();
```

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 HMTL (`<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.

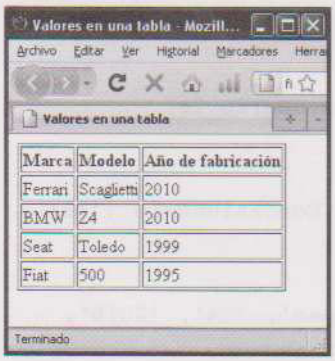


Figura 4.9. Valores de las propiedades del objeto `Coche` en una tabla HTML

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()` parar 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.

TEST DE CONOCIMIENTOS

1 ¿Qué palabra clave se utiliza para devolver un valor en una función?
- a) `send`.
- b) `return`.
- c) `function`.
- d) `var`.

2 ¿Cuál de estas instrucciones es correcta?
- a)
```
var input = prompt("Introduce un valor:");
if (input.isNaN) {alert ("Uso de NaN");}
```
- b)
```
var input = prompt("Introduce un valor:");
if (isNan = input) {alert ("Uso de NaN");}
```
- c)
```
var input = prompt("Introduce un valor:");
if (isNaN(input)) {alert ("Uso de NaN");}
```
- d)
```
var input = prompt("Introduce un valor:");
if (input == isNaN) {alert ("Uso de NaN");}
```

3 Una función se puede invocar desde:
- a) Código HTML y código JavaScript.
- b) Solo código HTML.
- c) Solo código JavaScript.
- d) Ninguna de las anteriores.

4 ¿Qué palabra clave se utiliza para definir una función?
- a) `create`.
- b) `function`.
- c) `new`.
- d) `object`.

5 Si se tiene un *array* llamado alimentos, ¿cómo se accede a su primer elemento?
- a) `alimentos.first`.
- b) `alimentos[1]`.
- c) `alimentos.top`.
- d) `alimentos[0]`.

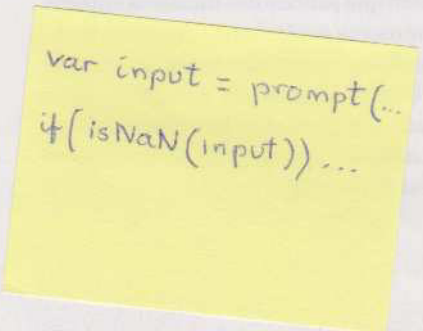
6 ¿Qué método se utiliza para eliminar el primer elemento de un *array*?
- a) `pop()`.
- b) `shift()`.
- c) `first()`.
- d) `delete()`.

7 ¿Qué método se utiliza para crear una cadena de texto a partir de los elementos de un *array*?
- a) `join()`.
- b) `concat()`.
- c) `toJoin()`.
- d) `string()`.

8 La propiedad `length` del objeto `Array` devuelve:
- a) El índice del último elemento.
- b) El número de elementos presentes en el *array*.
- c) Los elementos del *array* convertidos en una cadena de texto.
- d) Ninguna de las anteriores.

9 ¿Qué palabra clave se debe utilizar en la declaración de un objeto definido por el usuario?
- a) `function`.
- b) `object()`.
- c) `new`.
- d) Ninguna de las anteriores.

10 ¿Es posible añadir nuevas propiedades a objetos ya instanciados?
- a) Verdadero.
- b) Falso.



5 Interacción con el usuario. Eventos y formularios

OBJETIVOS DEL CAPÍTULO

- ✓ Reconocer las posibilidades de los lenguajes de marcas de capturar y gestionar los eventos producidos.
- ✓ Diferenciar los tipos de eventos que se pueden manejar.
- ✓ Crear código que capture y utilice eventos.
- ✓ Reconocer las capacidades del lenguaje relativas a la gestión de formularios web.
- ✓ Validar formularios web utilizando eventos y expresiones regulares para facilitar los procedimientos.
- ✓ Probar y documentar el código.

La interacción con el usuario, es la relación que establece el usuario con la aplicación web. Para ello en el lado del usuario se utilizan mecanismo de entrada como el teclado o el ratón. En el lado de la maquina se muestra la información a tratar, como por ejemplo un formulario web. El gran número de combinaciones posibles que surgen a la hora de actuar sobre una aplicación web, complica la forma en la que el código debe afrontar estas posibilidades. Los formularios contienen campos en ocasiones muy genéricos que deben ser adaptados al fin del campo en el formulario. Para manejar estas posibilidades existen los eventos que materializan los cambios que el usuario ocasiona en una aplicación web, asociando a los distintos objetos que nos vamos a encontrar una serie de posibles cambios que puede suceder sobre el objeto. A estos cambios se les llama *eventos*.

A continuación vamos a describir lo que es un evento y un formulario para poder asociar los conceptos de programación web a la funcionalidad que llevan asociada:

- **Evento**. Según la RAE un evento es una eventualidad, hecho imprevisto, o que puede acaecer. Cuando hablamos de una aplicación web, un evento es cualquier suceso relacionado con la aplicación web. Por lo tanto, un evento podría ser cerrar la ventana, dar un clic en un botón de radio o situarse con el ratón sobre la página. Es importante tener en cuenta que el conjunto de eventos de una aplicación web son los que pertenecen a la ventana de navegación donde se maneje esta aplicación, pudiéndose extender estos a un *popup* que se abra desde esta ventana.
- **Formulario**. Un formulario web, alojado en una página web, es una agrupación de objetos que tienen una función concreta. Estos objetos permiten al usuario introducir datos para enviarlos a un servidor y que sean procesados.

Figura 5.1. *Vista de un formulario dentro de un navegador*

Además del formulario mostrado en la imagen anterior, existen métodos para recuperar los campos en el lado del servidor. La página de destino del formulario será la encargada de recuperar estos datos.

5.1 MODELO DE GESTIÓN DE EVENTOS

Como hemos visto anteriormente, los eventos son mecanismos que se accionan cuando el usuario realiza un cambio sobre una página web. En una página web existen multitud de elementos. Estos elementos son los objetos que modelan la página web. El encargado de crear la jerarquía de objetos que compone una página web es el DOM. Por lo tanto, es el DOM (*Document Objet Model*) el encargado de gestionar los eventos.

Los eventos se sitúan en componentes de la pagina HTML, si hacemos un clic en un componente de la pagina, podríamos desencadenar una acción. Por un lado deberíamos marcar en el componente cual es el evento que queremos que desencadene la acción. Para poder controlar un evento necesitamos un manejador. Un manejador es una palabra reservada que indica la acción que va a manejar. En el caso del evento *click*, el manejador sería `onClick`. En este caso podemos asociar a un componente el manejador y desencadenar una acción cuando se haga un clic sobre ese componente.

Cuando ha sucedido el evento y este tiene un manejador asociado necesitamos la ayuda de un lenguaje de *script* como JavaScript o Visual Script. El lenguaje más usado es JavaScript. Este es un lenguaje interpretado que tiene la capacidad de actuar sobre los objetos DOM sin que enviemos una nueva petición al servidor. Es decir JavaScript puede modificar las propiedades de los objetos de una página en el mismo navegador.

A continuación vamos a ver un ejemplo de cómo JavaScript muestra un mensaje de alerta cuando hacemos clic sobre una imagen existente en la página web:

```
<IMG SRC="mundo.jpg" onclick="alert('Click en imagen');">
```

En el código anterior vemos que invocamos directamente a la función `alert()` de JavaScript. Si la acción a desencadenar es simple la podemos incluir directamente, aunque normalmente se realiza una llamada a una función. Será esta función la encargada de desencadenar la acción deseada. De esta forma estructuramos mejor el código y evitamos repetir sentencias si vamos a hacer varias veces una llamada para replicar un mismo comportamiento, pudiendo reutilizar una función en otra página si es preciso. A continuación mostramos la misma acción integrada en una página HTML llamando a una función:

```
<html>
  <head>
    <title>Pagina de Evento</title>
    <script>
      function func1() {
        alert("Click en imagen");
      }
    </script>
  </head>
  <body>
    <IMG SRC="mundo.jpg" onclick="func1();">
  </body>
</html>
```

En el código superior vemos como al el manejador `onclick` llama a la función, `func1` para que se ejecute. En la parte superior del código, dentro del `head` y entre las etiquetas `<script>...</script>` es donde debemos definir la función `func1` para que muestre la alerta cuando esta función sea llamada.

A continuación vamos a ver que la especificación de DOM define grupos de eventos dividiéndolos según su origen, estos son: eventos del ratón, del teclado, HTML y de DOM:

- **Eventos del ratón**. Estos eventos se originan cuando el usuario hace uso del ratón para realizar una acción. Cualquier movimiento de la flecha del ratón o acción sobre algunos de sus botones, puede desencadenar un evento. Cabe decir que no es necesario que la acción del ratón se realice sobre ningún objeto que interactúe con el usuario como puede ser un botón de radio o un *check*.
- **Eventos del teclado**. Estos eventos se originan cuando el usuario pulsa alguna tecla del teclado.
- **Eventos HTML**. Estos eventos se producen cuando hay algún cambio en la página del navegador. También pueden ocurrir cuando existe alguna interacción entre el cliente y el servidor.
- **Eventos DOM**. Los eventos DOM, o eventos de mutación, son los que se originan cuando existe algún cambio en la estructura DOM de la página.

5.1.1 EVENTOS DEL RATÓN

Los eventos del ratón son los más usuales puesto que la mayor parte de las acciones en una aplicación web las realizamos a través de este. A continuación vamos a ver cuáles son los eventos que define la especificación DOM:

- **Click**. Este evento se produce cuando pulsamos sobre el botón izquierdo del ratón. El manejador de este evento es `onclick`.
- **Dblclick**. Este evento se acciona cuando hacemos un doble clic sobre el botón izquierdo del ratón. El manejador de este evento es `ondblclick`.
- **Mousedown**. Este evento se produce cuando pulsamos un botón del ratón. El manejador de este evento es `onmousedown`.
- **Mouseout**. Este evento se produce cuando el puntero del ratón esta dentro de un elemento y este puntero es desplazado fuera del elemento. El manejador de este evento es `onmouseout`.
- **Mouseover**. Este evento al revés que el anterior se produce cuando el puntero del ratón se encuentra fuera de un elemento y éste se desplaza hacia el interior. El manejador de este evento es `onmouseover`.
- **Mouseup**. Este evento se produce cuando soltamos un botón del ratón que previamente teníamos pulsado. El manejador de este evento es `onmouseup`.
- **Mousemove**. Se produce cuando el puntero del ratón se encuentra dentro de un elemento. Es importante señalar que este evento se producirá continuamente una vez tras otra mientras el puntero del ratón permanezca dentro del elemento. El manejador de este evento es `onmousemove`.

Es importante destacar que el orden de ejecución de los eventos es el siguiente: *mousedown*, *mouseup*, *click*, *mousedown*, *mouseup*, *click*, *dblclick*. Siendo el último evento de la secuencia el doble clic.

out - fuera
over - encima

Cabe señalar que todos los elementos de las páginas web soportan los eventos del ratón, pero no todos los eventos que existen son soportados por todos los elementos de las páginas.

Cuando se produce un evento el objeto `event` se crea automáticamente. Este objeto contiene características adicionales sobre el evento que se ha producido. Estos datos pueden ser, la posición del ratón, el elemento que ha producido el evento, etc.

Para los eventos de ratón, el objeto `event` tiene una serie de propiedades, algunas de ellas son; las coordenadas del ratón (`screenX`, `screenY`), el nombre del evento (`type`), el elemento que origina el evento (`button`), etc.

Algunos navegadores como Internet Explorer permiten acceder al objeto `event` a través del objeto `Windows`. La especificación DOM, en cambio, indica que el único parámetro que se debe pasar a las funciones es el objeto `event`. A continuación mostramos cómo recuperar el parámetro del *array* en el código:

```
func1(){
  var elEvento = arguments[0];
}
```

No es necesario crear una variable que recupere el nombre del *array*, lo podemos indicar de forma explícita, como vemos en el siguiente código:

```
func1(elEvento) {
  ...
}
```

Puede parecer mágico que en la declaración de la función indiquemos un parámetro y en realidad no le pasemos ningún parámetro a la función. Los navegadores que siguen los estándares, crean ese parámetro y se lo pasan automáticamente a la función encargada de manejar ese evento.

5.1.2 EVENTOS DEL TECLADO

Los eventos de teclado son los que suceden cuando pulsamos una tecla. Según la especificación DOM existen los siguientes eventos relacionados con la actividad del teclado.

- **Keydown**. Este evento se produce cuando pulsamos una tecla del teclado. Si mantenemos pulsada una tecla de forma continua, el evento se produce una y otra vez hasta que soltemos la misma. El manejador de este evento es `onkeydown`.
- **Keypress**. Este evento se produce si pulsamos una tecla de un carácter alfanumérico (el evento no se produce si pulsamos **Enter**, la barra espaciadora, etc.). En el caso de mantener una tecla pulsada, el evento se produce de forma continuada. El manejador de este evento es `onkeypress`.
- **Keyup**. Este evento se produce cuando soltamos una tecla. El manejador de este evento es `onkeyup`.

Las propiedades de `event` para los eventos de teclado son: el código numérico de la tecla pulsada (`keyCode`), el código unicode del carácter correspondiente a la tecla pulsada (`charCode`), el elemento que origina el evento (`target`) y otras para identificar si hemos pulsado **shift** (`shiftKey`), **control** (`ctrlKey`), **alt** (`altKey`) o **meta** (`metaKey`).

Es importante destacar que el orden de secuencia de las teclas no es el mismo para un tipo de teclas que para otras. A continuación veremos cuál es el orden:

- Cuando pulsamos una tecla que corresponda a un carácter alfanumérico, la secuencia de eventos es la siguiente: *keydown*, *keypress*, *keyup*.
- En el caso de pulsar una tecla que no corresponda a un carácter alfanumérico, la secuencia de eventos es: *keydown*, *keyup*.
- Además existe la posibilidad de que dejemos una tecla pulsada. Para el primer caso, se repiten de forma continua los eventos *keydown*, *keypress*. En el segundo caso se repite de forma continuada el evento *keydown* solamente.

5.1.3 EVENTO HTML

Los eventos HTML, como hemos visto anteriormente, son los que actúan cuando hay cambios en la ventana del navegador. También se producen cuando hay ciertas interacciones entre cliente y el servidor. A continuación vamos describir cuáles son estos eventos y cuándo se acciona cada uno de ellos:

- **Load**. El evento *load* hace referencia a la carga de distintas partes de la página. Este se produce en el objeto `Window` cuando la página se ha cargado por completo. En el elemento `<img>` actúa cuando la imagen se ha cargado. En el elemento `<object>` se acciona al cargar el objeto completo. El manejador es `onload`.
- **Unload**. El objeto *unload* actúa sobre el objeto `Window` cuando la pagina ha desaparecido por completo (por ejemplo, si pulsamos el aspa cerrando la ventana del navegador). También se acciona en el elemento `<object>` cuando desaparece el objeto. El manejador es `onunload`.
- **Abort**. Este evento se produce cuando el usuario detiene la descarga de un elemento antes de que haya terminado, actúa sobre un elemento `<object>`. El manejador es `onabort`.
- **Error**. El evento *error* se produce en el objeto `Window` cuando se ha producido un error en JavaScritp. En el elemento `<img>` cuando la imagen no se ha podido cargar por completo y en el elemento `<object>` en el caso de que un elemento no se haya cargado correctamente. El manejador es `onerror`.
- **Select**. Se acciona cuando seleccionamos texto de los cuadros de textos `<input>` y `<textarea>`. El manejador es `onselect`.
- **Change**. Este evento se produce cuando los cuadros de texto `<input>` y `<textarea>` pierden el foco y el contenido que tenían ha variado. También se producen cuando un elemento `<select>` cambia de valor. El manejador es `onchange`.
- **Submit**. Este evento se produce cuando pulsamos sobre un botón de tipo `submit`. El manejador es `onsubmit`.
- **Reset**. Este evento se produce cuando pulsamos sobre un botón de tipo `reset`. El manejador es `onreset`.
- **Resize**. Este evento se produce cuando redimensionamos el navegador, actúa sobre el objeto `Window`. El manejador es `onresize`.
- **Scroll**. Se produce cuando varía la posición de la barra de desplazamiento (*scroll*) en cualquier elemento que la tenga. El manejador es `onscroll`.

- **Focus**. Este evento se produce cuando un elemento obtiene el foco. El manejador es `onfocus`.
- **Blur**. Este evento se produce cuando un elemento pierde el foco. El manejador es `onblur`.

Es importante indicar que el objeto *load* indica que la pagina se ha cargado entera. La especificación DOM requiere que la página y el árbol DOM se hayan cargado por completo. Por ello, este evento es muy importante y muy utilizado.

5.1.4 EVENTO DOM

Estos eventos hacen referencia a la especificación DOM. Se accionan cuando varía el árbol DOM. Estos eventos no están implementados en todos los navegadores, por tanto, habría que comprobar que el navegador soporta estos eventos, aun sabiendo que el navegador cumple con la especificación DOM.

- **DOMSubtreeModified**. Este evento se produce cuando añadimos o eliminamos nodos en el subárbol de un elemento o documento.
- **DOMNodeInserted**. Este evento se produce cuando añadimos un nodo hijo a un nodo padre.
- **DOMNodeRemoved**. Este evento se produce cuando eliminamos un nodo que tiene nodo padre.
- **DOMNodeRemovedFromDocument**. Este evento se produce cuando eliminamos un nodo del documento.
- **DOMNodeInsertedIntoDocument**. Este evento se produce cuando añadimos un nodo al documento.

5.2 UTILIZACIÓN DE FORMULARIOS DESDE CÓDIGO

Un formulario web es un utensilio que sirve para enviar, tratar y recuperar datos que son enviados y recibidos entre un cliente (navegador) y un servidor web. Cada elemento del formulario sirve para almacenar un tipo de dato o para accionar las distintas funcionalidades que tiene el formulario.

Los formularios disponen de una arquitectura, que habitualmente va ligada a otra más general. En nuestro caso los formularios están enmarcados en un lenguaje de marcado llamado HTML.

HTML es un lenguaje de marcado que estructura los contenidos de las páginas web. Los formularios están integrados en este lenguaje de programación y su función es, hacer que los usuarios interactúen con los datos de las páginas web, convirtiendo una página web en una aplicación web. A continuación vamos a ver cómo funciona un formulario web, y como se comunica el cliente con el servidor.

5.2.1 ESTRUCTURA DE UN FORMULARIO

Un formulario está compuesto por muchos elementos, pero para generar una estructura básica de un formulario, basta con utilizar dos de estos elementos. Al igual que HTML los formularios se definen con etiquetas. Dentro de cada etiqueta también podemos acceder a los atributo de la misma. La etiqueta principal de un formulario es `<form>`, esta

En el código anterior vemos como dentro del `body` de una página HTML introducimos una estructura `form`. En el `action` enviamos los datos del formulario (los datos corresponderían a los puntos suspensivos, los veremos en el siguiente punto) a la dirección, *www.web.es/formulario.php*, para que sean procesados en el servidor. En este caso los datos serán enviados de forma oculta puesto que hemos utilizado el método *POST*.

5.2.2 ELEMENTOS DE UN FORMULARIO

Según hemos visto en el punto anterior, un formulario sirve para enviar datos, pero... ¿cómo vamos a enviar esos datos? Y, además, ¿cómo vamos a indicar cuando queremos enviar esos datos? Para poder definir los datos que queremos enviar y cuando los vamos a enviar, tenemos los elementos del formulario. El elemento principal del formulario se denomina con la etiqueta `<input>`. Este elemento tiene una serie de atributos que modifican su funcionalidad. Los tipos de *input* según su funcionalidad se llaman *controles de formulario* y *campos de formulario*. Estos son los encargados de guardar los datos que vamos a enviar en el formulario. Los hay de varios tipos y a continuación vamos a definir cómo guardan y procesan los datos cada uno de ellos. También se encargan de accionar sucesos, por lo general son botones y al ser pulsados realizan la acción para la que están encomendados.

La etiqueta `input` tiene una lista de atributos muy extensa. A continuación vamos a definir cuáles son estos atributos, algunos atributos como el `type`, convierten al *input* en un elemento diferente generando ciertas peculiaridades con respecto a otros tipos de `input`.

5.2.3 ESTRUCTURA DE UNA ETIQUETA INPUT

El formato por defecto de una etiqueta `input` es el siguiente `<input ... />`, como es una etiqueta unaria se cierra a sí misma con la barra `( / )` al final. Esta etiqueta se transforma en distintos elementos que a su vez almacenan un tipo de dato. Para definir el tipo de elemento, la etiqueta `input` se sirve del atributo `type`. De esta forma si igualamos uno de los tipos validos al atributo `type`, modelamos el *input* al tipo asignado. A continuación vamos a ver el atributo `type` y el resto de atributos que puede tener una etiqueta `input`:

- **Type**. Es el que indica el tipo de elemento que vamos a definir. De él dependen el resto de parámetros puesto que el elemento cambia según sea de un tipo o de otro. Los valores posibles que acepta el atributo `type` son; `text` (cuadro de texto), `password` (cuadro de contraseña, los caracteres aparece ocultos tras asteriscos), `checkbox` (casilla de verificación), `radio` (opción de entre dos o más), `submit` (botón de envío del formulario), `reset` (botón de vaciado de campos), `file` (botón para buscar ficheros), `hidden` (campo oculto para, el usuario no lo visualiza en el formulario), `image` (botón de imagen en el formulario), `button` (botón del formulario). Debido a las peculiaridades de cada uno de los tipos procederemos a explicar detalladamente cada tipo en el punto siguiente, indagando en cómo afectan al resto de atributos.
- **Name**. El atributo `name` asigna un nombre al elemento. Si no le asignamos nombre a un elemento, el servidor no podrá identificarlo y, por tanto, no podrá tener acceso al elemento.
- **Value**. El atributo `value` inicializa el valor del elemento. Los valores dependerán del tipo de dato, en ocasiones los posibles valores a tomar serán verdadero o falso.
- **Size**. Este atributo asigna el tamaño inicial del elemento. El tamaño se indica en pixeles. En los campos `text` y `password` hace referencia al número de caracteres.

- **Maxlength**. Este atributo indica el número máximo de caracteres que pueden contener los elementos `text` y `password`. Es conveniente saber que el tamaño de los campos `text` y `password` es independiente del número de caracteres que acepta el campo.
- **Checked**. Este atributo es exclusivo de los elementos `checkbox` y `radio`. En el definimos que opción por defecto queremos seleccionar.
- **Disable**. Este atributo hace que el elemento aparezca deshabilitado. En este caso el dato no se envía al servidor.
- **Readonly**. Este atributo sirve para bloquear el contenido del control, por tanto, el valor del elemento no se podrá modificar.
- **Src**. Este atributo es exclusivo para asignar una URL a una imagen que ha sido establecida como botón del formulario.
- **Alt**. El atributo `alt` incluye una pequeña descripción del elemento. Habitualmente, y si no lo hemos desactivado cuando posicionamos el ratón (sin pulsar ningún botón) encima del elemento, podemos visualizar la descripción del mismo.

5.2.4 TIPOS DE INPUT

Como ya hemos visto en el punto anterior, la mutación que sufre cada tipo de *input* hace que sean elementos diferentes dentro del formulario. Por esta razón el estudio detallado de cada tipo, es necesario, para comprender las opciones que nos brindan los formularios. El objetivo de los tipos es adecuar los datos para facilitar la recopilación, manejo y envío de datos a través del formulario. También para la recuperación y comprensión de los mismos cuando el usuario recibe un formulario del servidor. A continuación vamos a detallar cada uno de estos tipos viendo un ejemplo de cada uno de ellos.

- **Cuadro de texto**. El cuadro de texto muestra un cuadro de texto vacío en el que el usuario puede introducir un texto. Este es uno de los elementos más usados. La forma de indicar que es un campo de texto es la siguiente `type="text"` La visualización en la página es la siguiente:

Figura 5.3. *Muestra de un cuadro de texto antecedido del texto nombre*

Para que el cuadro de texto pueda ser accedido deberá tener inicializado el atributo `name`, `name="nombreCuadro"`. El valor que indicamos en el atributo `name` es el que emplea el servidor para identificar este campo una vez se ha realizado el envío del formulario. El valor correspondiente a `name` deberá corresponder a un valor de variable como en cualquier otro lenguaje de programación. Respetando el no poner espacios en blanco, etc. El atributo `value` inicializará con un valor el cuadro de texto, `value="Javier"`. En el caso de que no mostremos el tamaño, el navegador mostrará el cuadro de texto con un tamaño predeterminado. De esta forma adaptaremos el tamaño del cuadro de texto al tipo de dato que va a incluir. Con el atributo `maxlength` podemos limitar el número de caracteres que podrá incluir el usuario, así, realizamos una primera limitación a la hora de validar el formulario. Es muy útil a la hora de indicar que un dato como, por ejemplo, un teléfono, no permita introducir más de 9 caracteres. El atributo `readonly` permite que en el formulario visualicemos el

cuadro de texto, pero no nos permite modificarlo. Este atributo es útil cuando realizamos una consulta de datos que residen en el servidor pero existen datos que no podemos modificar, por ejemplo el nombre de usuario a la hora de modificar el resto de datos. `Disabled` como el atributo anterior no permite modificar el campo de texto y además no envía los datos al servidor. A continuación vamos a ver cómo quedaría el código de una etiqueta `input` con los atributos anteriores:

```
<input t,gth="10" readonly />
```

En el código anterior vemos un *input* de tipo `text`, que se llama `nombre`, esta inicializado con la cadena de caracteres `"Javier"` y no se puede modificar por el usuario en el formulario.

- **Cuadro de contraseña**. El cuadro de contraseña es como el cuadro de texto, con la diferencia de que en el de contraseña, los caracteres que escribe el usuario no se ven en pantalla. En su lugar los navegadores muestran asteriscos o puntos con el fin de orientar al usuario del número de caracteres que va escribiendo. Su utilidad es escribir datos privados del usuario, como contraseñas u otros datos sensibles.

Figura 5.4. *Cuadro de contraseña en un formulario*

```
<input type="password" name="contrasenia" />
```

En el código anterior mostramos un *input* de tipo `password` que se llama `contrasenia`.

Los atributos indicados en el cuadro de texto pueden ser utilizados de igual forma en el cuadro de contraseña.

- **Casilla de verificación**. Estos elementos permiten al usuario activar o desactivar la selección de cada una de las casillas de forma individual. Su uso habitual es para indicar afirmación o negación sobre la casilla a la que se refiera.

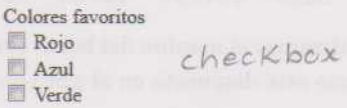


Figura 5.5. *Casilla de verificación de un formulario*

```
<p>Colores favoritos</p>
</br><input name="rojo" type="checkbox" value="ro"/> Rojo
</br><input name="azul" type="checkbox" value="az"/> Azul
</br><input name="verde" type="checkbox" value="ve"/> Verde
```

En el código anterior mostramos los 3 *input* de tipo `checkbox`, estos se llaman `rojo`, `azul` y `verde`, los valores que enviarán al servidor en caso de ser verificados son `ro`, `az`, y `ve`, respectivamente.

En el caso de querer mostrar que uno de los *input* de tipo `checkbox` este seleccionado por defecto lo tendríamos que hacer con el atributo `checked`. A continuación mostramos un ejemplo del código correspondiente:

```
<input name="rojo" type="checkbox" value="ro" checked /> Rojo
```

- **Opción de radio**. Este tipo de elemento agrupan una serie de opciones excluyentes entre sí. De esta forma el usuario solo puede coger una opción de entre todas las que tiene establecidas un grupo de botones radio. Al seleccionar una opción el usuario, automáticamente se deselecciona la que estaba seleccionada.

Figura 5.6. *Opción de radio de un formulario*

A continuación mostramos el código correspondiente la opción de radio que hemos mostrado anteriormente:

```
Género
</br><input type="radio" name="género" value="M"> Hombre
</br><input type="radio" name="género" value="F"> Mujer
```

En el código anterior observamos que el nombre de los *input* es el mismo. El nombre de los elementos que pertenezcan a un mismo grupo de opciones de radio deberá ser el mismo.

Es posible fijar un valor inicial con el atributo `checked`. En ese caso el *input* que deseemos que quede seleccionado por defecto deberá llevar dentro el atributo `checked`, `checked="checked"`.

- **Botón de envío**. Este elemento es el encargado de enviar los datos del formulario al servidor. En este caso el `type` toma el valor `submit`. El valor del atributo `value` se mostrará en este caso en el botón generado.

Figura 5.7. *Botón de envío de un formulario*

```
<input type="submit" name="enviar" value="Enviar">
```

En el código anterior inicializamos el nombre del botón con la cadena `"Enviar"`. El botón enviará los datos del formulario a la dirección que esté dispuesta en el `action`.

- **Botón de reset**. Este elemento es un botón que establece el formulario a su estado original:

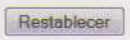


Figura 5.8. *Botón de restablecimiento de un formulario*

En este botón no hemos añadido el atributo `value`. Es el navegador el encargado de asignar un valor por defecto, el valor por defecto en el caso anterior es `"Restablecer"`.

- **Ficheros adjuntos**. Este tipo de *input* permite adjuntar ficheros adjuntos. Las limitaciones sobre cuántos ficheros y el tamaño no están definidas por el elemento. Por lo tanto, deberá controlarse desde otra parte del código para evitar desbordamientos o colapsos en el servidor. El elemento añade de forma automática un cuadro de texto que se dispondrá para almacenar la dirección del fichero adjunto seleccionado. Pulsando un botón

situado a la derecha del cuadro de texto navegaremos por el árbol de directorios en busca del fichero a adjuntar y una vez seleccionado, éste se incluirá en el cuadro que hemos indicado antes.

Figura 5.9. *Opción añadir ficheros adjuntos de un formulario*

```
Fichero adjunto
<input type="file" name="fichero"/>
```

Vemos como en el código anterior no es necesario incluir nada más que el tipo de *input* como `file` para que en la Figura 5.9 nos muestre el campo de texto y el botón de examinar.

Es importante saber que para adjuntar archivos debemos indicar el tipo de envío que vamos a realizar. Esto lo indicamos en la estructura del `form`, inicializando el atributo `enctype`. Habitualmente el valor que toma este atributo es `multipart/form-data`. Por lo tanto, el código en la cabecera del formulario quedaría de la siguiente manera.

```
<form action="…" method="post" enctype="multipart/form-data">
  ...
</form>
```

- **Campos ocultos**. Los campos ocultos no son visibles en el formulario por el usuario. Estos elementos son útiles para enviar información de forma oculta que no tenga que ser tratada por el usuario. Son usadas a menudo para que el servidor pueda tratar la información o hacer alguna acción determinada.

```
<input type="hidden" name="campoOculto" value="cambiar"/>
```

En el código anterior vemos como el campo con nombre `campoOculto`, toma el valor `cambiar`. Esto podría usarse para que el servidor realice un cambio con respecto a algo.

- **Botón de imagen**. Este elemento es una personalización de un botón, cambiando el aspecto por defecto que tienen los botones de un formulario por una imagen. El aspecto del botón sería el de la imagen a la que hayamos hecho referencia.

```
<input type="image" name="enviar" src="imagen_mundo.jpg"/>
```

El elemento botón de imagen se tipa igualando a `image` el tipo de *input* como vemos en el código anterior.

- **Botón**. Existe un elemento botón, al que podemos asociar diferentes funcionalidades. De esta forma no nos tenemos que ceñir los botones de *submit* o *reset* que nos ofrecen los formularios.

El botón podrá tener cualquier texto en su interior a través de la inicialización del atributo `value`.

```
<input type="button" name="opcion" value="Opcion validar"/>
```

Como vemos en el código anterior inicializamos el atributo `type` al valor `button`, de esta manera el elemento se convierte en un botón. Estos botones por sí mismos no tienen funcionalidad. Habrá que aplicar una funcionalidad por medio de JavaScript para que se accione alguna tarea.

A continuación vamos a mostrar el código de un formulario con varios componentes para ver cómo se integran los componentes del mismo entre sí.

```
<form action="pagina.php" method="post"
  enctype="multipart/form-data" /> <br/>

  Nombre: <input type="text" name= "nombre" value="" size="42
    maxlength="30" /> <br/>

  Apellidos: <input type="text" name="apellidos" value=""
    size="40" maxlength="80" /> <br/>

  DNI: <input type="text" name="dni" value="" size="10"
    maxlength="9" /> <br/>
  Sexo: <br/>
  <input type="radio" name="sexo" value="hombre"
    checked="checked" />
  Hombre <br/>

  <input type="radio" name="sexo" value="mujer" />
  Mujer  <br/>

  Incluir mi foto: <input type="file" name="foto" /> <br/>

  <input name="publicidad" type="checkbox" value="publicidad"
    checked="checked"/> Enviar publicidad <br/>

  <input type="submit" name="enviar" value="Guardar cambios"
    />
  <input type="reset" name="limpiar" value="Borrar los datos
    introducidos" />
</form>
```

Este sería un formulario combinando la estructura básica vista anteriormente con varios de los elementos que hemos visto en los puntos anteriores. El aspecto del formulario en el navegador es el siguiente:

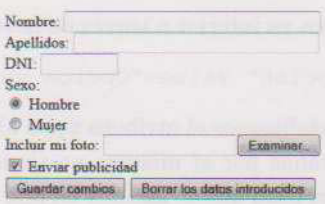


Figura 5.10. *Aspecto de un formulario dentro de un navegador*

5.3 MODIFICACIÓN DE APARIENCIA Y COMPORTAMIENTO

El lenguaje HTML no es muy flexible a la hora personalizar el aspecto de las páginas web. En cambio, habitualmente observamos que las páginas web muestran un aspecto de lo más variado, asemejando los colores, tonalidades y formas al objetivo de la página. Los formularios que hemos visto en los puntos anteriores, cumplen con las necesidades generales que exigen las aplicaciones para interactuar en la página web, pero a menudo surgen necesidades específicas para el desarrollo de una aplicación. Estas necesidades demandan una modificación del aspecto y para eso HTML hace uso de estilos. Las hojas de estilo modifican la apariencia de una página web. En el caso que nos atañe, las aplicaciones web, es especialmente importante poder modificar el aspecto, para conseguir que el usuario final tenga la sensación de manejar la aplicación igual que si lo hiciera en local.

En los siguientes puntos veremos que a través de estos estilos podemos mejorar notablemente el aspecto de los formularios, redondeando más sus formas, haciendo más uniformes los colores y estructurando las partes del formulario para que estos obtengan un aspecto mucho más agradable.

Los formularios tal y como los hemos visto en el punto anterior, pueden llegar a resultar un tanto estáticos. La diversidad de necesidades a la hora de desarrollar una aplicación web, nos hace pensar que necesitamos alguna herramienta que haga más flexible la utilización de formularios. De esta forma la proyección de un formulario en la aplicación web se puede disponer con una visualización u otra dependiendo de ciertas condiciones que vaya introduciendo el usuario.

En los puntos siguientes vamos a analizar cómo abordar el comportamiento de los formularios para que sus acciones tengan un funcionamiento para los que originalmente no estaban preparados. Para ello haremos uso de un lenguaje de *scripts* que nos permita personalizar las acciones y así conseguir los objetivos.

5.3.1 MODIFICACIÓN DE LA APARIENCIA DE UN FORMULARIO

Los formularios, por defecto, tienen unos estilos asignados. Estos estilos tienen asociados unos colores y unos bordes determinados. Desde los estilos por defecto, se pretende que estos se adapten en el entorno web en el que van a ser utilizados. Lamentablemente es difícil generalizar un estilo idóneo. Además, habitualmente, necesitamos personalizar tanto las formas como los colores para que estos se mimeticen con el resto de la aplicación web.

Para modificar el aspecto de un formulario, HTML utiliza estilos. Los estilos indican ciertas características de la apariencia de uno o más elementos del formulario. El lenguaje que maneja los estilos en HTML se llama *Cascading Style Sheets*. Estas siglas significan hojas de estilo en cascada. Habitualmente se denomina a este lenguaje CSS. Las modificaciones que podemos realizar con CSS son casi ilimitadas. A continuación vamos a ver cómo podemos modificar la apariencia de algunos elementos principales de un formulario.

- **Modificar el aspecto de un botón**. En ocasiones puede que necesitemos integrar un botón de un formulario en un texto o simplemente vemos la necesidad de que el enlace que envíe el formulario debería tener otro formato distinto al prefijado por el botón de formulario. A continuación vamos a modificar el aspecto de un botón para que este se muestre con una letra de un color diferente y tenga un borde más ancho de lo normal.

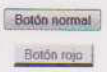


Figura 5.11. *Vista del mismo botón con aspecto diferente*

En la imagen vemos que el primer botón tiene el aspecto habitual para los formularios. En el segundo hemos modificado el color de la letra y el borde, agrandando el contorno del botón y cambiado el color de la línea inferior del mismo.

Para realizar esta modificación hemos aplicado un estilo al botón. Según el estilo aplicado el código del botón quedaría de la siguiente manera:

```
<input class="azul" type="button" value="Botón azul" />
```

Observamos como en el código hemos incluido un atributo llamado `class` al botón. De esta forma el botón buscara el estilo CSS correspondiente a ese nombre. El código CSS puede ser llamado de tres formas. Podemos aplicarlo directamente al elemento, práctica nada recomendable puesto que en el caso de querer modificar el estilo tendríamos que ir elemento por elemento. Otra de las formas es, generar un documento aparte que contenga los estilos, de manera que al comienzo de la página HTML se llame a este fichero y los elementos busquen su respectivo nombre de estilo en él. Y, por último, el modo que vamos a utilizar nosotros que es, incluir en la parte superior de la pagina HTML estos estilos. Los estilos CSS van incluidos dentro de la etiqueta `head`, que traducido al español significa cabecera. En el siguiente código mostramos el código CSS necesario para realizar la modificación del botón que hemos visto en la imagen anterior.

```
<html>
  <head>
    <title>Formulario</title>
    <style type="text/css">
      .azul{
        color: red;
        border-bottom: 4px solid blue;
      }
    </style>
  </head>
  <body>
    ...
  </body>
</html>
```

En el código podemos observar que el estilo `.azul` es el candidato que ha encontrado el atributo `class` del elemento botón para modificar sus colores y formas.

Imaginemos que queremos que el botón se muestre como un texto normal. Bastaría con que modificáramos el estilo indicando cómo queremos que se muestre éste. El código CSS sería el siguiente:

```
<style type="text/css">
  .azul{
    border: 0;
    background-color: transparent;
  }
</style>
```

Podemos observar en el código anterior como el borde del botón lo inicializamos a 0 y el color del botón como trasparente. De esta forma el botón se mostraría como un texto.

- **Suavizar el aspecto de un campo de texto**. Los campos de texto por defecto comienzan a escribir justo al principio del elemento. Al aparecer tan pegado la sensación que da al usuario al visualizarlo no es demasiado agradable. Desde CSS podemos modificar el punto a partir del que queremos que se escriban los caracteres. En la siguiente imagen mostramos la diferencia entre un campo de texto con el fondo amarillo, con el margen (`padding`) por defecto y debajo en el que se ha indicado un poco de margen en la parte izquierda.

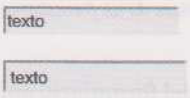


Figura 5.12. *Vista de un campo de entrada sin padding y con él*

El código correspondiente a los elementos *input* de tipo `text` anterior sería el siguiente:

```
<input class="col" type="text" value="texto" /></br></br>

<input class="color" type="text" value="texto" />
```

Para que los *input* recojan de las clases correspondientes a los estilos `col` y `color` necesitamos implementar el código CSS en la cabecera de la pagina HTML. El código es el siguiente:

```
.col{
  background-color: #FFFF00;
}

.color{
  padding: .2em;
  background-color: #FFFF00;
}
```

En el código se visualiza que el primer *input* pinta el fondo del campo de texto en amarillo y no tiene en cuenta el margen izquierdo. En el segundo caso de la llamada `color` se incluye un `padding` de 2 para que el aspecto visual quede más cuidado.

- **Organizar controles de un formulario**. Uno de los principales problemas que nos podemos encontrar a la hora de generar un formulario es cómo colocar los elementos para queden ordenados y bien guiados unos con otros. El aspecto de un formulario depende de que los elementos que lo integren estén bien organizados. A menudo se utilizan saltos de línea y tablas para integrar los elementos en sus celdas. Incluir CSS para guiar los elementos facilita la tarea, aumentando la capacidad de reutilización de código y modificaciones futuras. A continuación vamos a mostrar cómo organizar un formulario utilizando bloques. El código resultante mostraría un formulario con el siguiente aspecto.

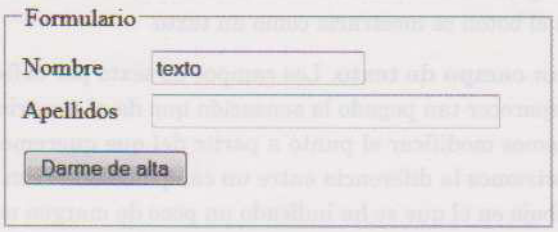


Figura 5.13. *Vista de un formulario en un bloque*

En la imagen hemos colocado los elementos del formulario aplicando bloques para que el guiado sea equivalente en elementos de todas las filas. El código a nivel de formulario sería el siguiente:

```
<fieldset>
  <legend>Formulario</legend>
  <div>
    <label for="nombre">Nombre</label>
    <input type="text" id="nombre" />
  </div>
  <div>
    <label for="apellidos">Apellidos</label>
    <input type="text" id="apellidos" size="35" />
  </div>
  <input class="btn" type="submit" value="Darme de alta" />
</fieldset>
```

En el código anterior no hemos introducido estilos. Solamente hemos estructurado los elementos dentro de un cuadro (`fieldset`). El *input* realiza una llamada a la clase `btn`. Veamos que a través de los estilos ordenamos los elementos:

```
<style type="text/css">
  div {
    margin: .4em 0;
  }

  div label {
    width: 25%;
    float: left;
  }

  .btn {
    display: block;
    margin: 1em 0;
  }
</style>
```

En el código anterior asignamos un margen superior de 4 y un margen izquierdo de 0 a todos los `div` que existan. En segundo lugar indicamos que los `label` ocuparan un 25% del `div`. El botón final lo agrupamos en forma de bloque, por lo que saldrá en una fila él solo y el margen superior que indicamos es de 1 para que no quede tan junto con la fila superior.

5.3.2 MODIFICACIÓN DEL COMPORTAMIENTO DE UN FORMULARIO

Los formularios tienen unas acciones predeterminadas por defecto. Estas acciones están asociadas a los elementos así como a la estructura principal del formulario. En el caso de que queramos darle otro comportamiento a un elemento del formulario, tenemos que modificar el elemento para que este no realice su función habitual. Imaginemos que tenemos la estructura principal de un formulario pero queremos que los datos se envíen a URL diferentes dependiendo de un dato que introduzca el usuario. En este caso deberíamos modificar el atributo `action` indicado en la estructura principal. Para realizar el envío a la URL podríamos accionar en un *script* la petición al servidor. A continuación mostramos un ejemplo de cómo se enviaría el formulario a una URL diferente:

```
<script language="javascript">
  function enviar(form){
    if (formulario.alta.checked == true){
      formulario.action = "paginas/alta.html";
    }

    if (formulario.alta.checked == false){
      formulario.action = "paginas/baja.html";
    }
    form.submit()
  }
</script>
```

En el código anterior comprobamos si la casilla `alta` ha sido chequeada. En caso de haber sido pulsada el `action` del `form` se inicia a un valor, si no ha sido pulsada se inicia a otro. Al final se envía el formulario por medio de un `submit`. Para que la función `enviar` sea ejecutada, debemos incluir un *input* de tipo `button` que contenga el evento `onclick`. En el manejador de este evento realizaremos la llamada de la siguiente forma: `Onclick="enviar (this.form)"`. El objeto `this` hace que podamos enviar el formulario a través de la función. De esta forma nunca llamamos al `action` de la estructura principal del formulario.

5.4 VALIDACIÓN Y ENVÍO

A la hora de de enviar un formulario, no podemos garantizar que el usuario haya insertado cada dato del formulario como esperamos. Supongamos que en un campo de texto el usuario tiene que incluir un código postal, pero el usuario en ese campo ha incluido una cadena de texto, por ejemplo, Madrid. Si el formulario se envía con ese dato se producirá un error. Para controlar si los campos de un formulario han sido rellenados correctamente haremos uso de las validaciones. Existen varios tipos de validaciones. Podemos realizar validaciones en el servidor, a nivel de servidor y también en la base de datos. Realizar validaciones a nivel de servidor supone que se recibe una petición, el servidor tiene que procesarla y emitir una respuesta. En el caso de que el usuario incluya muchos datos que no son correctos el servidor se puede ver sobrecargado. Para solucionar esta sobrecarga del servidor, realizamos un primer filtrado de los datos en el cliente. Estas validaciones son las que vamos a ver a continuación. En ellas analizamos con *scripts* en el navegador los datos que ha incluido el usuario una vez que pulsa el botón de enviar formulario. La petición no es enviada al servidor y se analiza dentro del navegador si existe algún dato que no se haya introducido correctamente. En el caso de que exista algún dato que no es correcto el *script* asignado muestra un mensaje al usuario indicando que debe realizar alguna modificación en el dato introducido. Si, por el contrario, se validan como correctos los datos, se envían los datos del formulario al servidor.

Este tipo de validaciones se suelen realizar llamando a una función que analice si el dato cumple con las restricciones que se han impuesto como, por ejemplo, en el caso anterior, donde el dato tenía que cumplir con los caracteres que puede tener un código postal. A continuación vamos a ver algunos de los ejemplos de validación más habituales, los tipos de validación que pueden existir son de lo más variado y basta con definirlas dentro de cada una de las funciones que valida un dato.

5.4.1 ESTRUCTURA DEL FORM PARA VALIDAR DATOS

Para que el formulario detecte que tiene que realizar una validación de los datos antes de enviar la petición al servidor, debemos indicárselo en su estructura. Para ello utilizaremos el evento `onsubmit` que ejecuta una acción cuando el `action` del formulario es llamado. Dentro del evento `onsubmit`, debemos llamar a una función, dependiendo de la respuesta de la función, el formulario enviará los datos al servidor (si los datos son correctos) o mostrará los diferentes mensajes de error, para los controles del formulario que se hayan realizado validaciones. El código correspondiente sería el siguiente:

```
<form action="URL" method="post" name="formValidado"
   onsubmit="return validar()">
   ...
</form>
```

En el código anterior observamos como la función `validar()` es llamada dentro del evento `onsubmit` y que dependiendo de la respuesta de validar el servidor envía o no los datos del formulario al servidor. Al realizarse un `submit` desde algún botón del formulario (petición de envío del formulario al servidor), siempre se realiza la llamada a la función `validar()`. Para que los datos del formulario sean validados, debemos tener implementada la función `validar()`. En esta función indicaremos cuáles son los datos del formulario que vamos a validar. A continuación mostramos algunos ejemplos de cómo validar datos de un formulario.

- **Validar un campo como obligatorio**. En ocasiones puede que nos interese que el usuario ingrese un campo como obligatorio en la aplicación web. En esta situación si el usuario no introduce el dato y antes de que el formulario sea enviado debemos comprobar si el campo está vacío. Habiendo llamado en la estructura del `form` a una función, la implementación de la función será la siguiente:

```
<script type="text/javascript">
  function validacion() {
    valor = document.getElementById("campo").value;

    if( valor == null || valor.length == 0 ){
      alert("El campo no puede ser vacío");
      return false;
    }
    return true;
  }
</script>
```

Este código deber ir integrado en la cabecera de la página HTML (`<head>`...código JavaScript...`</head>`). En primer lugar, recogemos el valor a través del objeto `document`, el nombre del *input* es `campo`. En el capítulo posterior veremos con detalle el árbol de estructura de una página web. En la estructura `if` comprobamos que el valor no sea nulo y que su longitud no sea cero. En caso de que el valor del campo `input` sea nulo o igual a cero, mostramos un mensaje de alerta y, posteriormente, devolvemos falso para que sea recuperado por el evento `onsubmit`. En ese caso no se enviaría el formulario.

- **Validar un campo de texto como numérico**. A menudo en los formularios nos encontramos con campos de texto que solo admiten números. Los teléfonos, código postal, DNI (sin letra) y otros muchos datos que obligatoriamente tienen una nomenclatura numérica. Para advertir al usuario de que el dato que ha introducido no es correcto utilizaremos el siguiente código implementado en una función:

```
<script type="text/javascript">
  function validaNum() {
    valor = document.getElementById("telefono").value;

    if( isNaN(valor) ) {
      alert("El campo tiene que ser numérico");
      return false;
    }
    return true;
  }
</script>
```

En el código anterior, tenemos una variable que se llama `valor`. Esta variable tomará el valor del `teléfono` o en su defecto el dato que haya introducido el usuario en la aplicación web. Con la función `isNaN()` comprobamos si el valor es numérico. En caso de que no lo sea devolverá un mensaje de alerta indicando que el dato tiene que ser numérico. Si por el contrario el dato es numérico la función devolverá verdadero.

- **Validar si una fecha es correcta**. Las fechas a menudo suelen ser datos en los formularios difíciles de comprobar, la razón es que hay infinidad de formas de expresarlas. En el ejemplo siguiente mostramos cómo validar de una forma sencilla una fecha, en la que se han ingresado el año, mes y día de forma independiente cada uno a través de tres campos diferentes:

```
<script type="text/javascript">
  function validaFecha() {
    var dia = document.getElementById("dia").value;
    var mes = document.getElementById("mes").value;
    var ano = document.getElementById("ano").value;

    fecha = new Date(ano, mes, dia);

    if( !isNaN(fecha) ) {
      return false;
    }
    return true;
  }
</script>
```

En el código anterior recogemos tres valores uno por el día otro por la fecha y el último por el año. Con la función `Date()` generamos una fecha introduciendo los valores. Posteriormente, comprobamos si la fecha que hemos creado es correcta, en caso de que el resultado sea que la fecha no es válida la función devuelve `false`.

- **Validar un checkbox**. Otra de las validaciones que podemos necesitar habitualmente, es comprobar si un *checkbox* ha sido seleccionado. A menudo en la aceptación de ciertas condiciones para acceder a servicios nos encontramos con esta comprobación, sin la que el formulario no tiene mucho sentido que continúe. En el siguiente código vamos a ver cómo se implementa la comprobación de si se ha activado un *checkbox*:

```
<script type="text/javascript">
  function validaCheck() {
    elemento = document.getElementById("campoCondiciones");

    if( !elemento.checked ) {
      return false;
    }
    return true;
  }
</script>
```

En el código anterior recuperamos el valor que ha tomado el elemento *checkbox* del formulario. En la siguiente condición comprobamos si éste ha sido chequeado, en caso de que no se haya pulsado la función devolverá falso. Si se ha chequeado la función devuelve verdadero.

Conviene tener en cuenta que si somos demasiado estrictos a la hora de validar o damos poca información, podemos bloquear al usuario de la aplicación web. Es posible que el usuario no sepa la manera de continuar en el proceso de manejo debido a un error en el tipo y validación de los datos.

5.5 EXPRESIONES REGULARES

Las expresiones regulares describen un conjunto de elementos que siguen un patrón. Dicho de otra forma, es una regla que identifica a una serie de elementos que tiene algo en común. Un ejemplo de expresión regular podrían ser todas las palabras que comienzan por la letra "a" minúscula. La expresión regular para este patrón debe asegurarse de que la palabra comienza por "a" minúscula, el resto de palabras que precedan a la cadena podrán ser cualquier carácter.

La mayoría de los lenguajes de programación incluyen la implementación de expresiones regulares. En el caso que nos atañe, JavaScript implementa una configuración para implementar expresiones regulares y así facilitar las comprobaciones de ciertos datos que deben seguir una estructura concreta. Este tipo de expresiones es muy útil a la hora de evaluar algunos tipos de datos que normalmente utilizamos en los formularios.

5.5.1 CARACTERES ESPECIALES DE LAS EXPRESIONES REGULARES

A continuación vamos describir algunos de los elementos que utiliza JavaScript para crear expresiones regulares. En el siguiente apartado vamos a representar la simbología que nos presta la expresión regular y, posteriormente, significado o interpretación:

- ✓ **^ Principio de entrada o línea**. Este carácter indica que las cadenas deberán comenzar por el siguiente carácter. Si éste fuera una "a" minúscula, como indicamos en el punto anterior, la expresión regular seria sería, `^a`.
- ✓ **$ Fin de entrada o línea**. Indica que la cadena debe terminar por el elemento precedido al dólar.
- ✓ *** El carácter anterior 0 o más veces**. El asterisco indica que el carácter anterior se puede repetir en la cadena 0 o más veces.
- ✓ **+ El carácter anterior 1 o más veces**. El símbolo más indica que el carácter anterior se puede repetir en la cadena una o más veces.
- ✓ **? El carácter anterior una vez como máximo**. El símbolo interrogación indica que el carácter anterior se puede repetir en la cadena cero o una vez.
- ✓ **. Cualquier carácter individual**. El símbolo punto indica que puede haber cualquier carácter individual salvo el de salto de línea.
- ✓ **x|y x ó y**. La barra vertical indica que puede ser el carácter *x* o el *y*.
- ✓ **{n} n veces el carácter anterior**. El carácter anterior a las llaves tiene que aparecer exactamente *n* veces.
- ✓ **{n,m} Entre n y m veces el carácter anterior**. El carácter anterior a las llaves tiene que aparecer como mínimo *n* y como máximo *m* veces.
- ✓ **[abc] Cualquier carácter de los corchetes**. En la cadena puede aparecer cualquier carácter que esté incluido en los corchetes. Además, podemos especificar rangos de caracteres que sigan un orden. Si se especifica el rango [a-z] equivaldría a incluir todas las letras minúsculas del abecedario.

- **[^abc] Un carácter que no esté en los corchetes**. En la cadena pueden aparecer todos los caracteres que no estén incluidos en los corchetes. También podemos especificar rangos de caracteres como en el punto anterior.
- **\b Fin de palabra**. El símbolo `\b` indica que tiene que haber un fin de palabra o retorno de carro.
- **\B No fin de palabra**. El símbolo `\B` indica cualquiera que no sea un límite de palabra.
- **\d Cualquier carácter dígito**. El símbolo `\d` indica que puede haber cualquier carácter numérico, de 0 a 9.
- **\D Carácter que no es dígito**. El símbolo `\D` indica que puede haber cualquier carácter siempre que no sea numérico.
- **\f Salto de página**. El símbolo `\f` indica que tiene que haber un salto de página.
- **\n Salto de línea**. El símbolo `\n` indica que tiene que haber un salto de línea.
- **\r Retorno de carro**. El símbolo `\r` indica que tiene que haber un retorno de carro.
- **\s Cualquier espacio en blanco**. El símbolo `\s` indica que tiene que haber un carácter individual de espacio en blanco: espacios, tabulaciones, saltos de página o saltos de línea.
- **\S Carácter que no sea blanco**. El símbolo `\S` indica que tiene que haber cualquier carácter individual que no sea un espacio en blanco.
- **\t Tabulación**. El símbolo `\t` indica que tiene que haber cualquier tabulación.
- **\w Carácter alfanumérico**. El símbolo `\w` indica que puede haber cualquier carácter alfanumérico, incluido el de subrayado.
- **\W Carácter que no sea alfanumérico**. El símbolo `\W` indica que puede haber cualquier carácter que no sea alfanumérico.

5.5.2 VALIDAR UN FORMULARIO CON EXPRESIONES REGULARES

Con la combinación de estas expresiones podemos abordar infinidad de patrones para validar datos en la mayoría de los formularios. Combinando cada una de las condiciones obtenemos patrones como pueden ser, una dirección de correo electrónico, un teléfono, un código postal, un DNI, etc. A continuación, vamos a ver cómo realizar algunas de las configuraciones de expresiones regulares más utilizadas para validad datos de un formulario.

- **Validar una dirección de correo electrónico**. En el caso de que el usuario de la aplicación web introduzca una dirección de correo electrónico, es casi imprescindible que esta sea correcta. Utilizando expresiones regulares vamos a comprobar que la estructura de la dirección sea correcta. En primer lugar, comprobaremos que comienza por una cadena de texto, que sigue por una @ y que termina por otra cadena de texto un punto y otra cadena de texto. De esta forma aseguraremos que, al menos, la estructura de la dirección es correcta. Combinando las funciones con las expresiones regulares, obtenemos un código que valida una dirección de correo electrónico.

  La llamada en la estructura principal del `form` sigue siendo la misma que en los casos anteriores y recogemos en el evento `onsubmit` la respuesta de la llamada a la función que valida, en nuestro caso la dirección de correo electrónico. El código correspondiente a la función JavaScript que valida un correo electrónico sería el siguiente:

```
<script type="text/javascript">
  function validaEmail() {
    valor = document.getElementById("campo").value;

    if(!(/\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-
      .]\w+)/.test(valor)) ) {
      return false;
    }
    return true;
  }
</script>
```

- **Validar un DNI**. Un dato muy habitual en los formularios es el documento nacional de identidad. A la hora de que el usuario interactúe con la aplicación web, es necesario comprobar que el DNI que introduce es correcto. No podemos asegurar que el DNI sea el de la persona que se está identificando, pero sí podemos asegurar que el DNI que introduce el usuario es correcto. Para validad un DNI introducido podemos utilizar el siguiente código.

```
<script type="text/javascript">
  function validaDNI(){
    valor = document.getElementById("dni").value;

    var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F',
      'P', 'D', 'X', 'B', 'N', 'J', 'Z', 'S', 'Q', 'V',
      'H', 'L', 'C', 'K', 'E', 'T'];

    if( !(/^\d{8}[A-Z]$/.test(valor)) ){
      return false;
    }

    if(valor.charAt(8) != letras[(valor.substring(0, 8))%23])
    {
      return false;
    }
    return true;
}
</script>
```

En el código anterior validamos un DNI en varios pasos. Primero recogemos en la variable `valor` el campo del DNI que ha introducido el usuario. Generamos un *array* con las letras validas para el DNI. En la primera condición comprobamos a través de expresiones regulares que el patrón del campo introducido tiene una longitud de 8 números y una letra. Si no es así devuelve falso la función.

En la última condición realizamos la división entre 23 y tomamos el resto. Comprobamos que la posición del *array* `letras`, perteneciente al resto de la operación, se corresponde con el valor de la letra del DNI introducido. En caso de que sea igual devuelve verdadero (al final de la función). Si no es igual la letra del valor introducido a la calculada, entra en la condición y devuelve falso.

- **Validar un número de teléfono**. Otro dato que es importante que sea correcto para interactuar en una aplicación web con el usuario son los números de teléfono. Al igual que en casos anteriores, no podemos comprobar que el numero pertenezca al usuarios que lo introduce, pero si podemos asegurar que el número de teléfono tiene un formato correcto. Las expresiones regulares son un recurso muy útil para validar un teléfono. A continuación vamos a ver cómo validaremos un número de teléfono:

```
function validaTelefono(){
  valor = document.getElementById("telefono").value;

  if( !(/^\d{9}$/.test(valor)) ) {
    return false;
  }
  return true;
}
```

En el código anterior estamos validando que el número que introdujo el usuario este formado por dígitos y tenga una longitud de 9. A la hora de validar un teléfono podemos ser más estrictos. Si queremos que el campo solo corresponda a un número de teléfono móvil, podemos especificar que el número solo pueda comenzar por 6, puesto que en España los teléfonos móviles comienzan todos por 6. La expresión regular integrada en la condición anterior para comprobar esto sería la siguiente:

```
if( !(/^[6]\d{8}$/.test(valor)) ) {
```

Esta expresión indica que el dato debe comenzar por el número 6 y que los siguientes 8 caracteres deben ser numéricos. En el caso de que queramos validar un número de teléfono fijo, el código sería el que mostramos a continuación:

```
if( !(/^[89]\d{8}$/.test(valor)) ) {
```

En el código los números solo pueden comenzar por 8 o por 9, que son los números actuales válidos en España.

En el caso de que la nomenclatura de teléfonos cambiara por alguna razón tecnológica, solamente hay que cambiar la función que valida el dato, para que ésta permita devolver verdadero en los casos de los teléfonos que cumplan con las expresiones regulares.

Existen otros muchos datos que podemos validar con expresiones regulares, como pueden ser números de cuentas bancarias, CIF de empresas y cualquier dato que siga un patrón determinado. Validar los datos para que el usuario no envíe el formulario si estos no son correctos es algo que aumenta el rendimiento del servidor, al disminuir las peticiones. También es importante mostrar una información amplia de como el usuario tiene que introducir los datos que se validen, de lo contrario, si el usuario no conoce cómo debe ingresar el dato puede quedarse bloqueado ante la aplicación. Ser demasiado estrictos en las validaciones a veces es contraproducente puesto que la curva de aprendizaje de la aplicación web puede aumentar considerablemente.

En los casos de validaciones estrictas tenemos que tener en cuenta que el servicio de atención al usuario debe tener medios para facilitar el manejo de la aplicación web al usuario, bien sea telefónico, presencial o por correo electrónico, dependiendo de la infraestructura en la que se enmarque nuestra aplicación.

5.6 UTILIZACIÓN DE COOKIES

Para poder hablar de las *cookies* antes es necesario conocer cómo funciona el protocolo de comunicaciones HTTP (*HyperText Transfer Protocol*), puesto que las *cookies* surgieron como necesidad ante algunas ausencias tecnológicas de este protocolo. El protocolo HTTP fue desarrollado por W3C (*Consorcio World Wide Web*), entre el año 1999 y 2000. Este protocolo sigue el esquema siguiente.

Petición cliente >>>>>> Respuesta del servidor

Este protocolo no tiene estado o conexión, lo que quiere decir que no guarda información de las conexiones anteriores. Realiza la petición el cliente, el servidor responde y ahí finaliza la comunicación. En los comienzos del uso de páginas web, apenas había necesidad de mantener una comunicación con el cliente en una página. La información se mostraba en una página. Para ir a otra página que enlazaba con la anterior, simplemente se pulsaba en el enlace y se mostraba la nueva página. Con la evolución de las páginas web se comenzó a ver la necesidad de identificar al usuario que realizaba la petición al servidor. En una página que muestre el correo electrónico identificamos al usuario mediante un nombre y una contraseña pero, y en la siguiente página... ¿cómo sabe el servidor que envía los datos al cliente que se *logueó*? Podría realizar una petición de esa URL otro navegador y visualizar sin ningún tipo de validación, la interfaz de entrada del correo electrónico. Para resolver este problema nacieron las *cookies*.

Una *cookie* es un fichero de texto que se almacena en el ordenador del cliente. Cada navegador especifica una ruta en la que guarda sus *cookies*. Por lo tanto, las *cookies* son un fichero propio de cada navegador. El servidor solicita al navegador que este guarde las *cookies*. En las sucesivas peticiones del cliente (navegador) al servidor, el navegador envía la petición junto con la *cookie*. De esta forma, el servidor sabe que quien le realiza la petición es el cliente anterior y no otro.

Las *cookies* surgieron con el fin de mantener la información de los carritos de la compra virtuales, para la compra a través de la Web. Por medio de las *cookies* el usuario podía navegar entre las páginas sin perder los elementos que había seleccionado para comprar, pudiendo modificar esta cesta virtual a través de la misma aplicación web. Debido a las carencias del protocolo HTTP en seguida se comenzaron a usar las *cookies* para otros fines de identificación de usuarios.

5.6.1 USOS DE LAS COOKIES

El principal objetivo de las *cookies* es que el servidor pueda conocer alguna características del usuario (navegador), para así poder tomar una decisión con respecto si envía una respuesta, cómo la envía y qué datos van asociados. Partiendo de este objetivo común los fines que puede tener un servidor para solicitar al usuario que almacene *cookies* en su ordenador son muy diversos. A continuación vamos a ver algunos de los principales usos que se da a las *cookies*. En los puntos siguientes vamos a ver que todos los usos de las *cookies* no son igual de lícitos y que si el navegador no está bien configurado puede ser un agujero de seguridad. Al navegar por páginas que no ofrezcan confianza podemos tener una vulnerabilidad en el sistema, el servidor de la página de dudosa confianza podría acceder al sistema accediendo al disco para fines a los que no le hemos autorizado. Para solucionar estos problemas podemos limitar o anular el uso de las *cookies* en nuestro navegador.

- **Mantener opciones de visualización**. Las *cookies* son utilizadas en ocasiones para mantener unas preferencias de visualización. Algunas páginas como Google, permiten que el usuario haga una configuración de su página de entrada en el buscador, a través de las *cookies* el servidor reconoce ciertos aspectos que el usuario configuró y conserva el aspecto.
- **Almacenar variables**. El servidor puede utilizar las *cookies* para almacenar variables que se necesiten utilizar en el navegador, puede almacenar los campos de un formulario que se va visualizando en páginas diferentes. Un ejemplo sería una página en la que nos solicitan unos datos, en la siguiente nos solicitan otros datos y así hasta la página final. Los datos de las páginas anteriores se irán almacenando en las *cookies* hasta que se finaliza el ciclo del formulario y el usuario envía los datos al servidor. Antes del envío al servidor se recuperarán todos los campos del formulario que están guardados en las *cookies*.
- **Realizar un seguimiento de la actividad de los usuarios**. En ocasiones los servidores hacen uso de las *cookies* para almacenar ciertas preferencias y hábitos que el usuario tiene a la hora de navegar. Con esta información, el servidor recoge información para poder personalizar sus servicios y publicidad orientándolo a cada cliente en particular. Estos fines no son del todo lícitos si la entidad que realiza estas actividades no avisa al usuario de que está realizando estas acciones.
- **Autenticación**. Es uno de los usos más habituales de las *cookies*, autenticar a los usuarios. A través de las *cookies*, el navegador guarda los datos del usuario, al realizar una petición al servidor, el navegador envía las *cookies* junto con la petición. De esta forma el servidor sabe que el usuario que realiza la petición es el que se autentico la primera vez. Las *cookies* tiene una caducidad, cuando pasa un período de tiempo establecido, estas desaparecen junto con el fichero de texto que guarda el navegador. En el caso de la autenticación las *cookies* suelen caducar cuando se finaliza la navegación por la página, cerrando la ventana. Habitualmente, como mecanismo de seguridad, las aplicaciones web aplican un tiempo máximo de inactividad, por ejemplo de 15 minutos, tras el cual las *cookies* caducan si no se produjo movimiento en la navegación de la aplicación web.

5.6.2 LECTURA Y ESCRITURA DE LAS COOKIES

Hasta ahora hemos hablado de que son y para qué sirven las *cookies* pero, ¿cómo se implementan? Al principio las *cookies* solo podían ser almacenadas por la petición de un CGI (*Common Gateway Interface*) desde el servidor. Un CGI es, según su traducción, una interfaz de entrada común. Es una tecnología que permite a un cliente (navegador) solicitar datos de un programa que se ejecuta en un servidor.

La compañía de software Netscape implementó en su lenguaje JavaScript la capacidad de introducir las *cookies* directamente desde el cliente (navegador), sin necesidad de CGI. Al principio se produjeron ciertas vulnerabilidades de seguridad, por lo que es muy importante configurar convenientemente el navegador, puesto que las *cookies* pueden ser creadas, borradas, aceptadas o bloqueadas directamente en el navegador. A continuación vamos a ver cómo se escribe y se lee una *cookie* del navegador.

- **Implementación de lectura y escritura de una cookie**. Los dos procesos de implementación principales de una *cookie* son la escritura y la lectura de la misma. A continuación vamos a ver el código de cómo se escribe una *cookie* y cómo se lee:

```
<html>
 <head>
  <script type="text/javascript">
   function getCookie(c_name){
    var i,x,y,ARRcookies=document.cookie.split(";");
    for (i=0;i<ARRcookies.length;i++){
     x=ARRcookies[i].substr(0,ARRcookies[i].indexOf("="));
     y=ARRcookies[i].substr(ARRcookies[i].indexOf("=")+1);
     x=x.replace(/^\s+|\s+$/g,"");
     if (x==c_name){
      return unescape(y);
     }
    }
   }

   function setCookie(c_name,value,exdays){
    var exdate=new Date();
    exdate.setDate(exdate.getDate() + exdays);
    var c_value=escape(value) + ((exdays==null) ? "" : ";
    expires="+exdate.toUTCString());
    document.cookie=c_name + "=" + c_value;
   }

   function checkCookie(){
    var username=getCookie("username");
    if (username!=null && username!=""){
     alert("Bienvenido " + username);
    }else{
     username=prompt("Por favor, Introduzca su usuario:","");
     if (username!=null && username!=""){
      setCookie("username",username,365);
     }
    }
   }
  </script>
 </head>
 <body>
  <input type="button" name="chequeaCookie" value="Chequear
   las cookies" onclick="checkCookie();">
 </body>
</html>
```

En el código anterior tenemos tres funciones JavaScript. La primera función devuelve el valor de una *cookie* de la cual hemos pasado el nombre a la función. Para introducir el valor de la *cookie* accedemos a través del árbol del documento (páginas HTML), llamado `document`.

En la segunda función escribimos una *cookie* pasando el nombre de la variable, el valor y la fecha de caducidad. Se comprueba la fecha actual y se le suman los días pasados por parámetro.

En la tercera función comprobamos si existe un valor para la *cookie*, en el caso de que no exista un valor se llama la función `setCookie()`. Ésta se encarga de solicitar al usuario el nombre e introducirlo en la *cookie*. La próxima vez que chequeemos la *cookie* el navegador contendrá ya el valor del nombre, por tanto, mostrará el nombre que introdujimos antes. Para volver a repetir la operación de insertar un nombre que guardemos en la *cookie*, debemos borrar las *cookies* del navegador.

Podemos probar que si desactivamos las *cookies* del navegador, este código que hemos visto anteriormente no funciona. Por lo tanto, las aplicaciones web que hagan uso de las *cookies* requieren que éstas estén activadas en el navegador. De lo contrario, por ejemplo, si se usan las *cookies* para logarse en la aplicación, el usuario no podría acceder a la misma.

ACTIVIDADES 5.1

- Se propone al alumno realizar una plantilla con los eventos que existen y dónde puede utilizar cada uno de ellos.
- La arquitectura de una aplicación web está compuesta por diferentes tecnologías (HTML, CSS, JavaScript). Defina en un esquema sobre cómo se unen y cuáles son los elementos principales que tiene cada una de ellas.
- En el tema hemos visto varios tipos de validaciones. Si tenemos que abordar la validación de un número de cuenta bancaria, un CIF, que un campo de entrada tenga como mínimo 6 caracteres y que un campo para poder ser rellenado dependa de si uno anterior se rellenó, ¿cómo podríamos implementar estas validaciones?
- Realice un estudio de cuáles pueden ser los problemas que generan las *cookies* en una aplicación web. Busque en Internet qué alternativas existen para abordar la funcionalidad de las *cookies*. Concluya si la tendencia es utilizar las *cookies* o si son otras tecnologías las que se están tendiendo a utilizar.
- Busque en Internet qué es el protocolo HTTPS. Comente qué aporta este protocolo con respecto al uso de las *cookies*.

RESUMEN DEL CAPÍTULO

Los eventos ofrecen mecanismos para que cuando hay movimientos en una aplicación web se desencadenen acciones implementadas por el programador. Los eventos están asociados al teclado, al ratón, a la página del navegador o al árbol DOM.

Los formularios se utilizan para crear, modificar y manipular información. También sirven para ayudar a emular los comportamientos de cualquier aplicación en local que queramos transformar en la aplicación web.

Los estilos CSS modifican la apariencia de las paginas HTML y también de los formularios. De esta forma adaptamos la aplicación web a fin destinado. Se puede asemejar el aspecto de una aplicación web a la misma aplicación en local tanto que nos costaría indicar cuál es cada una.

La validación en el cliente sirve para disminuir las peticiones al servidor, dar más rapidez en la respuesta al cliente y comprobar los datos antes de ser enviados. Las expresiones regulares nos ayudan a validar los datos que tienen patrones de comportamiento.

Las *cookies* nos ayudan a mantener el estado entre las páginas web puesto que éstas carecen de esa característica. Además, nos permiten guardar información del usuario en el caso de que éste visite de nuevo la página para poder atenderle de forma más personalizada.

EJERCICIOS PROPUESTOS

1. Realice una pequeña aplicación en la que exista un formulario con los distintos tipos vistos en el tema. Valide cada campo para que el usuario no pueda introducir un tipo de datos diferente al requerido. Valide también para que no pueda dejar campos en blanco. Cuando el usuario introduzca los datos correctamente en el formulario introduzca estos datos en *cookies* y proporcione al formulario un botón para que estos puedan ser leídos.

TEST DE CONOCIMIENTOS

1 Los eventos son mecanismos que se accionan:
a) En el servidor.
b) En el cliente.
c) En el servidor y en el cliente.
d) Todas las anteriores.

2 Los eventos, según su origen, se dividen en:
a) Eventos de ratón y teclado.
b) Eventos de acción y reacción.
c) Eventos HTML y DOM.
d) La respuesta *b* no es correcta.

3 El orden de ejecución de los eventos:
a) Se ejecutan en el orden en el que se pusieron.
b) Tienen un orden de ejecución en cada caso.
c) Se ejecutan todos a la vez si se refieren al mismo elemento.
d) Hay que asignar la prioridad de ejecución a cada evento si queremos que cambie el orden de ejecución.

4 Los eventos HTML:
a) Son todos los eventos que existen.
b) Son los eventos que actúan cuando hay cambios en la ventana del navegador.
c) Estos eventos no se producen en el objeto `Window`.
d) `Submit` no es un evento HTML.

5 Para poder utilizar un formulario:
a) Es necesario tener un atributo `action`.
b) Es necesario tener un atributo `action` y uno `method`.
c) Es necesario tener un atributo `action`, un `method` y un `name`.
d) Es necesario tener un atributo `action` y uno `name`.

6 En un formulario:
a) Tenemos el atributo `type` que indica el tipo de elemento.
b) El atributo `name` identifica de forma única cada elemento.
c) El atributo `checked` está definido para todos los elementos.
d) Las respuestas *a* y *b* son correctas.

7 El tipo de elemento *contraseña*, que se define con `password`:
a) Cifra los datos introducidos en el campo para que al enviarlos estos no puedan ser interceptados.
b) Sustituye los campos por asteriscos o círculos para que no los vean las miradas indiscretas.
c) Modula la contraseña para que ésta no sea interceptada al enviarse.
d) No se usa casi nunca porque es muy poco seguro.

8 Para modificar el aspecto de un elemento del formulario:
a) Utilizamos hojas de estilos en la cabecera u otro fichero.
b) Aplicamos los estilos al elemento directamente con CSS, es lo más recomendable.
c) Aplicamos atributos de HTML para modificar el aspecto.
d) Usamos JavaScript para modificar el aspecto de los elementos.

9 Los datos se validan en el navegador para:
a) Dar seguridad a la aplicación y evitar fallos.
b) Evitar demasiadas peticiones al servidor, mejorar la experiencia del usuario y asegurar que la información enviada es correcta.
c) No tener que validar los datos en el servidor puesto que se sobrecarga más.
d) No tener que comprobar en la base de datos que los datos son correctos.

10 Las *cookies* son:
a) Almacenadas en el servidor.
b) Almacenadas en un directorio del ordenador cliente.
c) Almacenadas entre en servidor y el navegador.
d) Todas las anteriores son correctas.

6 Utilización del Modelo de Objetos del Documento (*DOM-Document Object Model*)

OBJETIVOS DEL CAPÍTULO

- ✓ Reconocer el modelo de objetos del documento de una páginas Web, identificando sus objetos, propiedades y métodos.
- ✓ Generar y verificar código que acceda a la estructura del documento y crear nuevos elementos de la estructura.
- ✓ Asociar acciones a los eventos del modelo.
- ✓ Identificar diferencias del modelo en distintos navegadores. Programar aplicaciones para que funcionen en los distintos navegadores.
- ✓ Separar las facetas, contenido, aspecto y comportamiento en aplicaciones Web.

En el capítulo anterior hemos visto que accedíamos a través de un objeto que se llamaba `document` a algunos elementos de la página web. El objeto `document` es el objeto de más alto nivel dentro del modelo de objetos del documento. ¿Qué es el modelo de objetos del documento? Es un estándar de W3C que define cómo acceder a los documentos, como por ejemplo HTML y XML. A este estándar se le llama *Document Object Model*, o DOM, que traducido significa *Modelo de Objetos del Documento*. El DOM es una interfaz de programación de aplicaciones (API) de la plataforma de W3C y cuenta con un lenguaje neutro que permite a los programas y *scripts* acceder y actualizar dinámicamente su contenido, estructura y estilo de documento.

6.1 EL MODELO DE OBJETOS DEL DOCUMENTO (DOM)

El DOM fue utilizado por primera vez con el navegador Netscape Navigator (versión 2.0 del navegador). A esta primera versión de DOM, se la considera *modelo básico* o *DOM nivel 0*. El primer navegador de Microsoft que comenzó a utilizar el modelo básico fue Internet Explorer 3.0. En la versión 3.0 de Netscape y la 4.0 de Internet Explorer se comenzaron a utilizar *rollovers*. *Rollover* es un efecto que hace cambiar una imagen, por ejemplo, cuando pasamos el ratón por encima. A partir de aquí comenzaron a incluir en algunos navegadores la capacidad de detectar eventos de ratón y de teclado. Debido a las diferencias entre los navegadores W3C se emitió una especificación a finales de 1998 que se llamó *DOM nivel 1*. En esta especificación ya se consideraban las características y manipulación de todos los elementos existentes en los archivos HTML y también de XML. A finales del año 2000, W3C emitió la especificación *DOM nivel 2*. En esta especificación se incluyó el manejo de eventos en el navegador y la capacidad de interactuar con hojas de estilo CSS. En 2004 se emitió la especificación *DOM nivel 3*, la cual utiliza la definición de tipos de documento (DTD) y la validación de documentos.

6.1.1 TIPOS DE MODELOS DOM

Actualmente DOM se divide en tres partes diferentes según la W3C. A estas partes también se les llama niveles. A continuación vamos a ver cuáles son estos tres niveles y en qué consiste cada uno de ellos:

- **Núcleo del DOM**. Este es el modelo estándar para cualquier documento estructurado. En este modelo se especifican a nivel general las pautas para definir los objetos y propiedades de cualquier documento estructurado, así como los métodos para acceder a ellos.
- **XML DOM**. Este es el modelo estándar para los documentos XML. Este modelo define los objetos y propiedades de todos los elementos XML, así como los métodos para acceder a ellos.
- **HTML DOM**. Modelo estándar para los documentos HTML. Este es el modelo en el que nos vamos a centrar. El DOM HTML es un modelo de datos estándar para HTML. Una interfaz de programación estándar para HTML independiente de la plataforma y el lenguaje. Es un estándar de la W3C. El DOM HTML define los objetos y las propiedades de los elementos HTML. También define los métodos para acceder a ellos. Por lo tanto, DOM HTML es un estándar sobre la forma de obtener, modificar, añadir o eliminar elementos HTML.

En el punto siguiente vamos a ver cómo se estructura el modelo DOM para facilitar la organización de un documento de aplicación web.

6.1.2 ESTRUCTURA DEL ÁRBOL DOM

El modelo DOM HTML se define a través de una estructura de árbol. Para poder utilizar las utilidades DOM, el navegador web transforma automáticamente todas las páginas web en una estructura de árbol. Las páginas web son una sucesión de caracteres, por lo que resultaría excesivamente complicado manipular los elementos si no fuera por esta conversión. La estructura de cada página web, se organiza de forma que se pueda acceder a los elementos a través de la estructura de árbol. DOM transforma los documentos HTML en elementos. Estos elementos se llaman nodos. A su vez los nodos están interconectados y muestran el contenido de la página web y la relación entre nodos. Por lo tanto, al conjunto de nodos se le llama, árbol de nodos. A continuación mostramos un ejemplo de una estructura de nodos generada por DOM a partir de una página HTML.

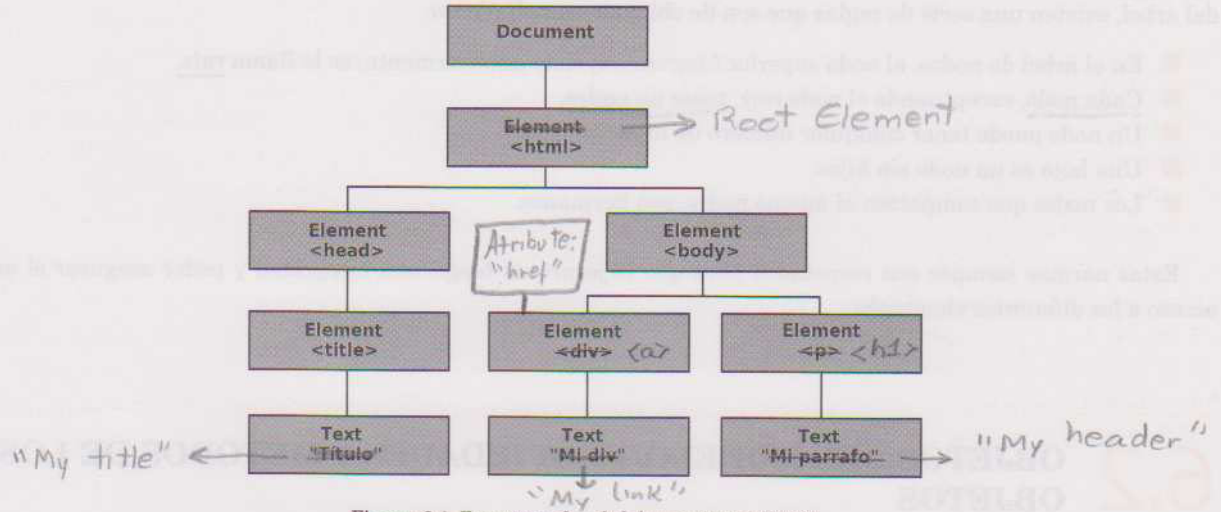


Figura 6.1. Estructura de arbol de una página HTML

Según vemos en la figura anterior, DOM crea una estructura por cada uno de los elementos de la página HTML. A estos elementos que se sitúan en el árbol se les llama nodos. Por lo tanto, el árbol de la figura anterior tendría once nodos.

En la estructura anterior el nodo `<html>` sería el nodo raíz, por tanto, todo está contenido dentro de él. La estructura en código HTML de la imagen de la Figura 6.1 sería la siguiente:

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="">My link</a>
    <h1>My header</h1>
  </body>
</html>
```

El navegador convierte cada página de la aplicación web en una estructura de árbol. Como podemos observar los datos de cada elemento se almacenan en un nodo aparte, que cuelga del nodo elemento. Existe un nodo especial llamado documento, está en el nivel superior como hemos visto en la Figura 6.1. A partir de este nodo raíz, las etiquetas HTML se transforman en distintos tipos de nodo.

El modelo DOM define la forma en que los objetos y elementos se relacionan entre sí, tanto en el navegador como en el documento. Cada objeto tiene un nombre único. Cuando existe más de un objeto de un tipo, estos objetos se organizan en un vector.

El primer objeto que nos encontramos es el objeto `document`, este objeto es la raíz del árbol de nodos. Por lo tanto, es un objeto nodo. Este nodo al ser el primero, no tiene nodos padre, tampoco tiene nodos a su mismo nivel. Solo tiene nodos hijo. De acuerdo con el DOM, todo lo que existe en un documento HTML es un nodo. Para ordenar la estructura del árbol, existen una serie de reglas que son de obligado cumplimiento:

- En el árbol de nodos, al nodo superior (`document`, visto anteriormente) se le llama raíz.
- Cada nodo, exceptuando el nodo raíz, tiene un padre.
- Un nodo puede tener cualquier número de hijos.
- Una hoja es un nodo sin hijos.
- Los nodos que comparten el mismo padre, son hermanos.

Estas normas siempre son respetadas para que la jerarquía tenga una integridad y poder asegurar el orden y acceso a los diferentes elementos.

6.2 OBJETOS DEL MODELO. PROPIEDADES Y METODOS DE LOS OBJETOS

Una vez que DOM ha creado de forma automática el árbol completo, tenemos los nodos del árbol DOM. El estándar considera un nodo a cada una de las partes del árbol, por tanto, en una de las propiedades del nodo se especifica el tipo de nodo. Esta propiedad es `nodeType`. La veremos en los puntos anteriores junto con el resto de propiedades. Existen doce tipos de nodos posibles, cada uno tiene unas propiedades asociadas. A continuación vamos a estudiar cuáles son los tipos de nodos que pueden existir en un árbol DOM. Los tipos de nodos indicados son específicos del lenguaje XML, no obstante estos pueden ser aplicados a los lenguajes basados en XML como, por ejemplo, XHTML. En las páginas HTML, el navegador hace como si HTML estuviera basado en XML y lo trata de la misma forma.

6.2.1 OBJETOS DEL MODELO

Según los tipos de nodos, como hemos visto en el punto anterior, las características de estos van a variar considerablemente. A continuación vamos a definir cada uno de los tipos de nodos que existen en el DOM HTML. Según la W3C, estos son los más importantes.

Tipos de nodos:

- **Document**. Es el nodo raíz del documento HTML. Todos los elementos del árbol cuelgan de él.
- **DocumentType**. Este nodo indica la representación del DTD de la página. Un DTD es una definición de tipo de documento. Define la estructura y sintaxis de un documento XML. El *DOCTYPE* es el encargado de indicar el `DocumentType`.
- **Element**. Este nodo representa el contenido de una pareja de etiquetas de apertura y cierre (`<etiqueta>...</etiqueta>`). También puede representar una etiqueta abreviada que se cierra a si misma (`<br />`). Este es el único nodo que puede tener tantos nodos hijos como atributos.
- **Attr**. Este nodo representa el nombre del atributo o valor.
- **Text**. Este nodo almacena la información que es contenida en el tipo de nodo `Element`.
- **CDataSection**. Este nodo representa una secuencia de código del tipo `<![CDATA[ ]]>`. Este texto solo será analizado por un programa de análisis.
- **Coment**. Este nodo representa un comentario XML.

Viendo la relación que genera el estándar entre las partes de una aplicación web a nivel de código y los tipos de nodos del árbol, queda estructurado el árbol para poder manipular de una forma fácil cada uno de los elementos. En el punto siguiente vamos a definir cómo se accede al documento y a los tipos de nodos desde DOM.

6.2.2 LA INTERFAZ NODE

Para poder manipular la información de los nodos, JavaScript crea un objeto, llamado `Node`. En este objeto se definen las propiedades y los métodos para procesar y manipular los documentos. El objeto `Node` define una serie de constantes que identifican los tipos de nodo. En la siguiente tabla vemos una relación entre las constantes y un número que se asigna a cada uno de los tipos de nodo.

Tabla 6.1 Asociación de constantes valor de tipo de nodo

Tipo de nodo=Valor
`Node.ELEMENT_NODE = 1`
`Node.ATTRIBUTE_NODE = 2`
`Node.TEXT_NODE = 3`
`Node.CDATA_SECTION_NODE = 4`
`Node.ENTITY_REFERENCE_NODE = 5`
`Node.ENTITY_NODE = 6`
`Node.PROCESSING_INSTRUCTION_NODE = 7`
`Node.COMMENT_NODE = 8`
`Node.DOCUMENT_NODE = 9`
`Node.DOCUMENT_TYPE_NODE = 10`
`Node.DOCUMENT_FRAGMENT_NODE = 11`
`Node.NOTATION_NODE = 12`

Además de estas constantes el objeto `Node` proporciona una serie de propiedades y métodos para poder acceder a la jerarquía de elementos del árbol. Estos los mostramos en la tabla a continuación:

Tabla 6.2 Métodos y propiedades de *Node*

Propiedad/Método	Valor devuelto	Descripción
`nodeName`	`String`	Nombre del nodo (no está definido para algunos tipos de nodo).
`nodeValue`	`String`	Valor del nodo (no está definido para algunos tipos de nodo).
`nodeType`	`Number`	Una de las 12 constantes definidas anteriormente.
`ownerDocument`	`Document`	Referencia del documento al que pertenece el nodo.
`firstChild`	`Node`	Referencia al primer nodo de la lista `childNodes`.
`lastChild`	`Node`	Referencia al último nodo de la lista `childNodes`.
`childNodes`	`NodeList`	Lista de todos los nodos hijo del nodo actual.
`parentNode`	`Node`	Referencia al padre del nodo hijo.
`previousSibling`	`Node`	Referencia al nodo hermano anterior o `null` si este nodo es el primer hermano.
`nextSibling`	`Node`	Referencia al nodo hermano siguiente o `null` si este nodo es el último hermano.
`hasChildNodes()`	`Boolean`	Devuelve `true` si el nodo actual tiene uno o más nodos hijo.
`attributes`	`NamedNodeMap`	Lo empleamos con nodos de tipo `Element`. Contiene objetos de tipo `Attr` que definen todos los atributos del elemento.
`appendChild(nodo)`	`Node`	Añade un nuevo nodo al final de la lista `childNodes`.
`removeChild(nodo)`	`Node`	Elimina un nodo de la lista `childNodes`.
`replaceChild(nuevoNodo, anteriorNodo)`	`Node`	Reemplaza el nodo `anteriorNodo` por el nodo `nuevoNodo`.
`insertBefore(nuevoNodo, anteriorNodo)`	`Node`	Inserta el nodo `nuevoNodo` antes de la posición del nodo `anteriorNodo` dentro de la lista `childNodes`.

Con estos métodos, podemos acceder a los diferentes nodos del árbol y también a la jerarquía. Así como crear, modificar y eliminar nodos.

6.3 ACCESO AL DOCUMENTO DESDE CÓDIGO

Cuando el árbol de nodos DOM ha sido construido por el navegador de forma automática, y se ha cargado completamente, podemos acceder a cualquier nodo. Si existe más de un tipo de un elemento, estos se van almacenando en un vector. A continuación vamos a ver la estructura de acceso a través del árbol DOM en una página HTML. Supongamos que tenemos una paginas HTML con la siguiente estructura:

```
<html>
  <head>
    <title>Titulo DOM</title>
  </head>
  <body>
    <p>Parrafo DOM</p>
    <p>Parrafo DOM segundo</p>
    <p>Parrafo DOM tres</p>
  </body>
</html>
```

Cuando la página ha sido cargada, podemos acceder a los elementos. Según la estructura de DOM, el primer paso es recuperar el objeto que representa el elemento raíz de la página. En realidad el objeto que se define como raíz es *HTMLDocument*. El objeto `document` es parte del BOM (*Browser Object Model*), pero se considera equivalente al `document` de DOM, por lo que `document` también hace referencia al nodo raíz las páginas HTML.

```
var obj_html = document.documentElement;
```

De esta forma cargamos en el objeto `obj_html` un objeto de tipo *HTMLElement* que representa al elemento `html` de nuestra estructura. De este elemento derivan siempre según la estructura DOM `<head>` y `<body>`, al saber que son dos, podríamos acceder a ellos recuperando el primer y último hijo del nodo `<html>` utilizando dos de los métodos de la tabla anterior.

```
var obj_head = obj_html.firstChild;
var obj_body = obj_html.lastChild;
```

En el caso de que existan más de dos nodos y queramos acceder a ellos, podríamos hacerlo a través del índice, el modo de acceso es a través de un vector que se genera con los objetos que están al mismo nivel. La forma de acceder es la siguiente:

```
var obj_head = obj_html.childNodes[0];
var obj_body = obj_html.childNodes[1];
```

En el caso de que no sepamos el número de nodos hijo, podemos acceder a este valor utilizando la propiedad `length` sobre el método `childNodes`.

```
var numeroHijos = obj_html.childNodes.length;
```

Otra forma de acceder a un nodo en el árbol, es a través de su hijo. Para recuperar el elemento deberíamos utilizar el método `parentNode` de la siguiente manera:

```
var obj_html = obj_body.parentNode;
```

Como dijimos en uno de los puntos anteriores, el nombre de los `element` coincide con el nombre de su etiqueta. Para acceder a los elementos, DOM permite acceder a través del nombre del elemento. Esto asigna el elemento al objeto que pretendamos. En el código que mostramos a continuación, asignamos el elemento `body` (llamado en la página HTML, `<body>...</body>`), al objeto `obj_body`.

```
var obj_body = document.body;
```

6.3.1 ACCESO A LOS TIPOS DE NODO

Los objetos del modelo, se diferencian por su tipo. Existen distintos tipos de nodo. La Tabla 6.1, muestra la relación entre la constante relacionada con el tipo de nodo y el número que se asigna al tipo. La forma de acceder al tipo de nodo, es a través de la propiedad `nodeType`. A través de la interfaz `Node`, JavaScript accede al tipo de nodo. Por lo tanto, la forma de acceder a un tipo de nodo es la siguiente:

```
obj_tipo_document = document.nodeType; // 9
obj_tipo_elemento = document.documentElement.nodeType; // 1
```

Las llamadas a los tipos de nodo anteriores, devuelven en el primer caso, "9" el valor asociado en la constante `Node.DOCUMENT_NODE`, asociado en la Tabla 6.1. En el segundo caso "1", valor asociado en la constante `Node.ELEMENT_NODE` según la Tabla 6.1 de las constantes y los números asociados a los tipos. El uso de los tipos puede ser accedido a través de las constantes, que vienen definidas por defecto en la mayoría de los navegadores. Por lo tanto, podemos usar sentencias del tipo:

```
if(document.nodeType == Node.DOCUMENT_NODE) then{
  alert("Verdadero")
} // true

if(document.documentElement.nodeType == Node.ELEMENT_NODE)
  then{
    alert("Verdadero")
  } // true
```

Las condiciones devolverán verdadero, puesto que las constantes definidas en la Tabla 6.1, tienen los valores por defecto. En el caso de tener un navegador que no tenga las constantes definidas por defecto, habría que definirlas de forma explícita. Esto ocurre con la versión 7 y anteriores del navegador Internet Explorer.

6.3.2 ACCESO DIRECTO A LOS NODOS

Los métodos que hemos visto anteriormente, nos permiten acceder a los nodos a través de la jerarquía del árbol. Esta forma de acceso hace necesario acceder al nodo raíz de la página para llegar a cualquier otro. Al acceder a un nodo en una página HTML real, el árbol tiene infinidad de nodos. Acceder a estos nodos resulta una tarea tediosa. En el caso de que modifiquemos la estructura del árbol añadiendo, modificando o quitando nodos, podemos ocasionar problemas en el acceso a los nodos. Por esta razón DOM añade una serie de métodos, que nos permiten acceder de forma directa a los nodos. Los métodos son `getElementByTagName()`, `getElementsByName()` y `getElementById()`. Los nombres son largos, pero tiene la ventaja de que el nombre del método es autodescriptivo sobre su función. A continuación vamos a describir en qué consiste cada uno y cuál es la forma de invocarlos en el código.

El método `getElementByTagName()` recupera los elementos de la página HTML de la etiqueta que hayamos pasado como parámetro. Un ejemplo de cómo devuelve las etiquetas `div` es:

```
var divs = document.getElementsByTagName("div");
```

La función devuelve un vector con los nodos de tipo `div`. En realidad el vector es una lista de nodos (`nodeList`). La forma de acceder a los `div`, es:

```
var primerDiv = divs[0];
var segundDiv = divs[1];
```

Podemos recuperar todos los `div` existentes en la página con una estructura repetitiva como es el `for`. En el caso anterior la forma de recuperar los `div` de la página sería:

```
var divs = document.getElementsByTagName("div");
for(var i=0; i<div.length; i++) {
  var div = div[i];
}
```

Esta función también la podemos aplicar para recuperar otra estructura dentro de una de un tipo ya recuperado. Siguiendo con el ejemplo anterior, mostramos cómo recuperar todos los enlaces del primer `div` del `nodeList`.

```
var divs = document.getElementsByTagName("div");
var primerdiv = div[0];
var enlaces = primerdiv.getElementsByTagName("a");
```

En el código anterior recuperamos todos los enlaces contenidos en el primer `div` de la página.

El método `getElementByName()` recupera todos los elementos de la página HTML en los que el atributo `name` coincide con el parámetro pasado a través de la función:

```
var divPrimero = document.getElementsByName("primero");

<div name="primero">...</div>
<dif name="segundo">...</div>
<div>...</div>
```

En el código anterior recuperamos el `div` que tiene por `name primero`. Habitualmente el `name` es único para cada elemento. En el caso de que existan varios elementos con el mismo `name`, recuperaríamos todos. En el caso de los *input* de tipo `radiobutton`, todos los elementos relacionados tienen el mismo `name`, en este caso la función recupera la colección de elementos. Esta función podríamos utilizarla para aplicar algo a un grupo de elementos que tengan el mismo `name`.

El método `getElementById()` recupera el elemento HTML cuyo *id* coincida con el pasado a través de la función. Esta función accede directamente al nodo a través del *id*, que se le pasa a la función. Como el *id* de cada elemento tiene que ser único, es la función más utilizada para leer y modificar sus propiedades.

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

<div id="pie">
  <a href="URL" id="imagen">...</a>
</div>
```

6.3.3 ACCESO A LOS ATRIBUTOS DE UN NODO TIPO ELEMENT

En los puntos anteriores hemos accedido a las etiquetas (nodo de tipo `element`) del árbol, pero DOM permite acceder directamente a todos los atributos de una etiqueta. Para que los atributos de una etiqueta, puedan ser accedidos, estos contienen la propiedad `attributes`. Esta propiedad nos permite acceder a todos los atributos de un nodo de tipo `element`. Para ello hace uso de los siguientes métodos:

- **getNameItem(nomAttr)**. Devuelve el nodo de tipo `attr` (atributo), cuya propiedad `nodeName` (nombre del nodo) contenga el valor `nomAttr`.
- **removeNameItem(nomAttr)**. Elimina el nodo de tipo `attr` (atributo) en el que la propiedad `nodeName` (Nombre del nodo) coincida con el valor `nomAttr`.
- **setNameItem(nodo)**. Este método añade el nodo `attr` (atributo) a la lista de atributos del nodo `element`. Lo indexa según la propiedad `nodeName` (del atributo).
- **item(pos)**. Devuelve el nodo correspondiente a la posición indicada por el valor numérico `pos`.

En los métodos indicados anteriormente, el tipo de nodo es `attr`. Este tipo de nodo corresponde con los atributos que están integrados dentro de un nodo de tipo `element` (etiqueta). El nodo de tipo `attr` no devuelve directamente el valor del atributo, si no el nombre del atributo. A continuación mostramos cómo podemos utilizar los métodos anteriores:

```
<p id="parraf" style="color:blue">Prueba de texto</p>

var p = document.getElementById("parraf");

// El valor de valorId es parraf
var valorId = p.attributes.getNamedItem("id").nodeValue;
var valorId = p.attributes.item(0).nodeValue;

// El valor de valorId es parraf
p.attributes.getNamedItem("id").nodeValue= "parrafModifica";
```

El párrafo anterior ha sido modificado, en el *id* y hemos añadido un atributo nuevo de tipo `name`, por tanto queda como lo mostramos a continuación:

```
<p id="parrafModifica" style="color:blue">Prueba de texto</p>
```

Estos métodos pueden resultar algo tediosos a la hora de acceder a los tributos de un nodo, por ello DOM proporciona unos métodos que permiten acceso directo a la modificación, inserción y borrado de los atributos de una etiqueta.

- **getAttribute(nomAtributo).**
  Este método equivale a: `attributes.getNameItem(nombAtributo).`
- **setAttribute(nomAtributo, valorAtributo).**
  Este método equivale a la estructura: `attributes.getNamedItem(nomAtributo).value = valor.`
- **removeAttribute(nomAtributo).**
  Este método equivale a la estructura: `attributes.removeNameItem(nomAtributo).`

Con los siguientes métodos, podemos hacer las acciones del código anterior de la siguiente manera:

```
<p id="parraf" style="color: blue">Prueba de texto</p>

var p = document.getElementById("parraf");

// El valorId es parraf
var valorId = p.getAttribute("id");

// Se añade al atributo id el valor parraTexto
p.setAttribute("id", " parraTexto ");
```

Con estos métodos podemos manejar todos los atributos que tiene una etiqueta HTML.

6.3.4 CREACIÓN Y ELIMINACIÓN DE NODOS

En los puntos anteriores hemos visto como acceder a los nodos y a las propiedades de los mismos, pero DOM nos permite la creación de nodos en el árbol de forma dinámica. Para crear nodos DOM proporciona una serie de métodos. Como los tipos de nodos son diferentes, existe un método para la creación de cada tipo de nodo. A continuación vamos a ver cuáles son los métodos que nos permiten crear nodos cuando el árbol ya ha sido creado.

- **createAttribute(nomAtributo)**. Este método crea un nodo de tipo atributo con el nombre pasado a la función.
- **createCDataSection(textoPasado)**. Este método crea una sección de tipo `CDATA` con un nodo hijo de tipo texto con el valor `textoPasado`.
- **createComent(textoPasado)**. Este método crea un nodo de tipo `coment` (comentario), con el contenido de `textoPasado`.
- **createDocumentFragment()**. Este método crea un nodo de tipo `DocumentFragment`.

- **createElement(nomEtiqueta).**
  Este método crea un elemento del tipo etiqueta, del tipo del parámetro pasado como `nomEtiqueta`.
- **createEntityReference(nomNodo).**
  Este método crea un nodo de tipo `EntityReference`.
- **createProcessingInstruction(objetivo,dato).**
  Este método crea un nodo de tipo `ProcessingInstruction`.
- **createTextNode(textoPasado).**
  Este método crea un nodo de tipo texto con el valor del parámetro pasado, `textoPasado`.

No todos los navegadores soportan estos métodos, Internet Explorer, por ejemplo, no soporta los métodos `createCDataSection()`, `createEntityReference()` y `createProcessingInstruction()`. En realidad los métodos más utilizados en la creación de nodos son `createElement()`, `createTextNode()` y `createAttribute()`. Con estos métodos podemos crear los nodos que aparecen mayoritariamente en una página HTML.

Crear un nodo en el árbol y añadirle un valor no es algo directo. Este proceso requiere de una serie de pasos. La razón de que existan varios pasos para crear y dar valor a un nodo es que existe un nodo para el elemento y, en el caso de que este elemento tenga un valor asociado, este valor es otro nodo hijo. Por lo tanto, un nodo de tipo `element`, que tiene contenido tendrá su nodo hijo de tipo texto asociado. De esta manera, es el nodo hijo el que tiene el contenido. Supongamos que partimos de una estructura HTML como la siguiente:

```
<html>
  <head><title>Ejemplo creación de un nodo</title></head>
  <body></body>
</html>
```

A continuación vamos a mostrar cómo crear un nodo de tipo `element` en nuestra página HTML. Los pasos a seguir son:

1 Creamos un nodo nuevo de tipo elemento utilizando la función vista anteriormente. Pasamos a la función el parámetro `div` para que nuestra etiqueta sea una de tipo `div`:

```
var div = document.createElement("div");
```

2 A continuación creamos un nodo de tipo texto. A la función la hemos pasado como parámetro el texto, `Hola mundo`, que será el contenido del nodo. La etiqueta la llamaremos `texto`.

```
var texto = document.createTextNode("Hola mundo");
```

3 Una vez que tenemos creados los dos nodos, asociamos al nodo de tipo `element` el de tipo texto para que éste sea su hijo.

```
div.appendChild(texto);
```

4 En este paso tenemos los dos nodos creados, y a la etiqueta `div` asociada a un nodo hijo de tipo texto. Esta etiqueta contiene la cadena `Hola mundo`. El elemento está creado pero no tiene una posición en el árbol, por tanto, si introdujéramos estos pasos en una página HTML, la etiqueta `div` no sería visualizada por el usuario. El último paso que tenemos que realizar es asociar al árbol de estructura de la página, nuestro nuevo nodo elemento.

```
document.body.appendChild(div);
```

En el código anterior hemos introducido la etiqueta `div` como hija del elemento `body`. Una vez realizados los pasos de creación e inserción de un nodo `div` en la página HTML, la estructura de la página anterior queda de la siguiente forma:

```
<html>
  <head><title> Ejemplo creación de un nodo </head>
  <body>
    <div>Hola mundo</div>
  </body>
</html>
```

Es importante que la creación de nodos la realicemos cuando el árbol esté construido. El navegador construye el árbol de nodos DOM cuando la página se ha cargado por completo. Si el árbol no está generado por completo, no podemos utilizar las funciones de creación de nodos. Como veremos en los puntos siguientes, existen mecanismos para detectar si la página se ha cargado completamente, el evento `onload()`, visto en el tema anterior, nos permitía, por ejemplo, comprobar si la página se había cargado por completo.

Eliminar un nodo del árbol es una tarea que requiere identificar previamente el nodo a eliminar. Podemos identificar un nodo a través de su posición en el vector de tipos de etiquetas. A continuación vamos a seleccionar el nodo tipo `div` que creamos anteriormente para eliminarlo:

```
var div = document.getElementsByTagName("div")[0];
```

Para eliminar un nodo debemos hacerlo a través de su nodo padre. Para ello, utilizaremos el método `removeChild()` que nos encontrábamos en la Tabla 6.2 de la interfaz de `Node`.

```
document.body.removeChild(div);
```

Vemos como accedemos al documento, luego al `body`, en la llamada al método, pasamos como parámetro el valor que nos ha devuelto antes la posición primera del vector.

Como en casos anteriores, las páginas reales tienen un árbol excesivamente complejo. Por esta razón es inviable detectar la posición en el árbol de un elemento. Las páginas, como hemos visto, las podemos modificar dinámicamente, lo que altera la estructura del árbol. En estos casos podrían producirse errores si realizáramos llamadas desde el primer nivel de la jerarquía. Para solucionar este problema podemos utilizar el método `parentNode()`, de esta forma no tenemos que acceder al nodo padre desde el primer nivel de la jerarquía del árbol. El código anterior para eliminar el nodo `div` quedaría de la siguiente manera:

```
div.parentNode.removeChild(div);
```

La inicialización del elemento `div` en una variable es igual. En la llamada al método hemos realizado la llamada sobre el nodo padre, pero subiendo un nivel en la jerarquía en vez de bajar hasta el padre desde el elemento de más alto nivel del árbol.

Modificar un nodo del árbol es una tarea que podemos utilizar para alternar o modificar la estructura de un árbol en la página HTML. Existen casos en los que puede ser conveniente remplazar un nodo por otro para conseguir una visualización diferente. En este caso utilizaremos el método `remplaceChild(paramNuevo, paramOriginal)`. Para

remplazar un nodo existente por otro tendremos que realizar la siguiente implementación. Según el ejemplo anterior, si queremos remplazar un `div` por el `div` que teníamos sería:

```
var nuevoDiv = document.createElement("div");
var nuevoTexto = document.createTextNode("Cadena que
  sustituye a Hola mundo");

nuevoDiv.appendChild(nuevoTexto);

var originalDiv = document.getElementsByTagName("div")[0];
originalDiv.parentNode.replaceChild(nuevoDiv, originalDiv);
```

En el código anterior, creamos un nuevo elemento `div` con su texto correspondiente, asociamos el `div` al hijo de tipo texto. Introducimos en la variable `originalDiv` el valor actual. En la última línea, llamamos al método remplazar hijo, invocando desde el método padre. Los parámetros que pasamos al método son, en primer lugar, el nodo nuevo y, en segundo lugar, el nodo que va a ser remplazado. La estructura de la página quedaría de la siguiente manera:

```
<html>
  <head><title> Ejemplo creación de un nodo </head>
  <body>
    <div> Cadena que sustituye a Hola mundo </div>
  </body>
</html>
```

Para poder situar en el árbol, donde queremos incluir un nuevo nodo, DOM, nos permite añadir un nodo antes de otro ya existente. Esto aumenta mucho el rendimiento de la programación. Imaginemos que queremos incluir una etiqueta `div`, que indique que hemos introducido mal un teléfono. Pretendemos que este mensaje se visualice justo antes de la estructura tipo `input` que nos permite introducir el teléfono. Bastaría con añadir la etiqueta `div` en el caso de que el valor del teléfono que se ha introducido no coincida con el patrón establecido. A continuación vamos a ver como añadiríamos una etiqueta:

```
var divAntes = document.createElement("div");
var textoAnt = document.createTextNode("div antes que
  original");

divAntes.appendChild(textoAnt);

var divOriginal = document.getElementsByTagName("div")[0];
document.body.insertBefore(divAntes, divOriginal);
```

En el código anterior creamos un nuevo `div`. Inicializamos el ya existente a una variable. Una vez en variables los dos `div`, utilizamos el método insertar, pasando como parámetros, primero la variable que queremos que se posicione antes y después la original. De esta forma la estructura de la página quedaría de la siguiente manera:

```
<html>
  <head><title> Ejemplo creación de un nodo </head>
  <body>
    <div> div antes que original </div>
    <div> Cadena que sustituye a Hola mundo </div>
  </body>
</html>
```

A través de estos métodos, podemos modificar la estructura de la página, para conseguir un dinamismo total en el lado del cliente, consiguiendo una limpieza y estructura de código limpia y ordenada.

6.4 PROGRAMACIÓN DE EVENTOS

A lo largo de los puntos anteriores, hemos visto multitud de acciones asociadas al modelo de objetos del documento (DOM). Hemos visto como podíamos modificar la estructura del árbol, añadir nodos, eliminarlos. En una aplicación web, es el usuario el que interactúa, además de desencadenantes como el tiempo u otras razones externas a la aplicación web. Por esta razón necesitamos algo que relacione la interacción del usuario con las acciones de DOM vistas anteriormente. Para relacionar esto, utilizamos los eventos. En el capítulo anterior veíamos como un evento era capaz de capturar una acción determinada. En función de la acción, por ejemplo hacer clic con el ratón, podíamos tomar una decisión programada. El lenguaje de programación de *script* que vamos a utilizar es JavaScript.

6.4.1 CARGA DE LA PÁGINA HTML

Una condición indispensable para que se genere la estructura de árbol, es que la página se haya cargado completamente. En el caso de que la página no se haya cargado, no podemos acceder a la estructura de árbol, puesto que esta no se ha generado, o al menos no se ha generado completamente. Por lo tanto, necesitamos conocer si la página se ha cargado, antes de realizar cualquier actuación sobre la estructura del árbol. Para comprobar si la página se ha cargado completamente, disponemos del evento `onload`. Este evento está definido para el nodo de tipo `element`, `<body>`. A continuación vamos a invocar al evento en una página HTML. Nos mostrará el mensaje (`"Página cargada completamente"`) cuando ésta se cargue completamente.

```
<html>
  <head><title>Titulo DOM</title></head>
  <body onload="alert('Página cargada completamente');">
    <p>Primer parrafo</p>
  </body>
</html>
```

El código anterior muestra el mensaje `"Página cargada completamente"` cuando la pagina se carga. De esta forma podemos asegurarnos de que el árbol esta completado.

6.4.2 COMPROBAR SI EL ÁRBOL DOM ESTÁ CARGADO

En DOM podemos modificar la estructura original de la página como hemos visto en los puntos anteriores. Por lo tanto, necesitamos poder saber si una vez hemos realizada una modificación en el árbol, la página se ha cargado de nuevo por completo.

A continuación vamos a comprobar a través de una función si se ha cargado la página. En el caso de que se haya cargado, al hacer clic sobre el párrafo, mostraremos el mensaje: "Se cargo la pagina correctamente".

```
<html>
  <head><title>Titulo DOM</title>
    <script>
      function cargada() {
        window.onload = "true";
        if(window.onload){
          return true;
        }
        return false;
      }

      function pulsar(){
        if(cargada()){
          alert("Se cargo la página correctamente");
        }
      }
    </script>
  </head>
  <body>
    <p onclick="pulsar();">Primer párrafo</p>
  </body>
</html>
```

En el código anterior vemos que la función `cargada()`, comprueba sobre el objeto `Window` (objeto de más alto nivel, la ventana), si ocurrió el evento `onload` (el evento `onload` ocurre, cuando la página se ha cargado completamente). Cuando pulsemos con un clic sobre "Primer párrafo" se comprueba si la función `cargada()` es verdadera. En caso de ser verdadera, nos muestra el mensaje "Se cargo la página correctamente". Si no, no muestra nada.

6.4.3 ACTUAR SOBRE EL DOM AL DESENCADENARSE EVENTOS

Una de las grandes ventajas que nos aporta DOM es poder actuar sobre la estructura del árbol, de forma inmediata. Los eventos por otro lado nos permiten definir el momento en el que queremos que se realice una acción. Estas dos características combinadas, convierten la arquitectura web en el lado del cliente, en algo dinámico, rápido y versátil, para conseguir cualquier objetivo.

A continuación vamos a ver cómo podemos actuar sobre la estructura del árbol, cuando se desencadena un evento. Una etiqueta `div`, tiene un valor por defecto. Cuando pasamos el ratón por encima cambia su valor, cuando el ratón no está por encima, el `div` toma otro valor diferente.

```
<html>
  <head><title>Titulo DOM</title>
    <script>

      function ratonEncima(){
        document.getElementsByTagName("div")
          [0].childNodes[0].nodeValue="EL RATON ESTA ENCIMA";
      }

      function ratonFuera(){
        document.getElementsByTagName("div")
          [0].childNodes[0].nodeVale="NO ESTA EL RATON
          ENCIMA";
      }
    </script>
  </head>
  <body>
    <div onmouseover="ratonEncima();
    "onmouseout="ratonFuera();">
    VALOR POR DEFECTO
    </div>
  </body>
</html>
```

En el código anterior hay dos funciones que modifican el texto que mostramos en la etiqueta `div`. Estas funciones actúan directamente a través de la estructura DOM, sobre el valor de la etiqueta `div`. En la etiqueta `div` se llama a dos eventos, el primero, para que se accione cuando el ratón esta encima de la etiqueta `div`. El segundo, para que se accione cuando el ratón no se sitúa encima del texto de la etiqueta `div`. Cuando la página se carga por primera vez, el texto que aparece, es el texto por defecto.

6.5 DIFERENCIAS EN LAS IMPLEMENTACIONES DEL MODELO

Una de las principales dificultades que nos encontramos a la hora de utilizar DOM, es que no todos los navegadores hacen una misma interpretación. W3C hace unas recomendaciones y crea unos estándares, que los navegadores van adoptando, con arreglo a la capacidad que tienen en el desarrollo del navegador. Los navegadores han de respetar los estándares que existían antes y adoptar los nuevos que van saliendo. Solo de esta forma es posible que la gran cantidad de páginas web que existen y no están adaptadas a las recomendaciones W3C puedan sobrevivir entre las que si cumplen los nuevos estándares. Las altas expectativas en generar los estándares, los caminos paralelos, los intereses de las empresas que están detrás de los diferentes navegadores, hacen que esta tarea sea lenta y exija un consenso entre los principales navegadores y la asociación estandarizadora W3C.

La guerra entre los navegadores a la hora de generar sus propios estándares, ha generado muchos problemas a los programadores de páginas web. Todos los navegadores utilizan JavaScript como uno de los lenguajes de programación en el entorno cliente, pero los objetos y eventos no se comportan de la misma forma en todos ellos. Esto obliga a generar diferente código dependiendo del navegador. Otra opción es limitar el uso de una aplicación web a uno o dos navegadores concretos. Los principales navegadores Internet Explorer, Firefox, Google Crome, Opera, Safari o Nescape Navigator implementan sus navegadores intentando adaptar la interpretación del código a los estándares. Además, dentro de un mismo navegador existen distintas versiones, de tal forma que unas soportan unos mecanismos y otras no.

6.5.1 ADAPTACIONES DE CÓDIGO PARA DIFERENTES NAVEGADORES

A la hora de abordar configuraciones de código que se adapten a varios navegadores, tenemos que tener en cuenta, que esto complica la estructura de la programación. Se pueden ocasionar problemas de interpretación, complicando la actualización futura de la página y aceptando que los navegadores que no soportan los estándares tienen intención de soportarlos en el futuro y es probable que el código implementado se tenga que retocar. Aun así, por exigencias nos podemos ver forzados a realizar adaptaciones para poder dar un servicio web en diferentes navegadores. W3C con el fin de estandarizar la Web, creo un modelo de objetos único, que es el que hemos visto en los puntos anteriores. Aun así, algunos navegadores, como Internet Explorer de Microsoft, han añadido su propia extensión de DOM, por lo que se generan problemas de interoperabilidad entre los navegadores web. A continuación vamos a ver algunas de las adaptaciones que podemos realizar para que nuestra aplicación web sea compatible con posibles carencias tecnológicas del navegador.

- **Crear de forma explícita las constantes predefinidas**. En uno de los puntos anteriores hablábamos de las constantes predefinidas que tenía la interfaz `Node` del DOM. El navegador que respeta este estándar genera de forma automática una serie de constantes, asociadas a los tipos de nodo. Estas constantes asocian una cadena de texto, que describe mejor el tipo de nodo al que nos estamos refiriendo.

  De esta forma no es necesario aprenderse a qué numero corresponde cada tipo de nodo. La programación así se facilita y aumenta la legibilidad del código. Un ejemplo de lectura de una constante predefinida por el navegador sería:

```
alert(Node.DOCUMENT_NODE); // Devolvería 9
alert(Node.ELEMENT_NODE); // Devolvería 1
alert(Node.ATTRIBUTE_NODE); // Devolvería 2
```

En los ejemplos anteriores vemos que las constantes que incluyen los tipos de nodos están definidas. En el caso de que estas constantes no estén definidas, podemos crearlas de forma explícita dentro del navegador.

```
if(typeof Node == "undefined") {
  var Node = {
    ELEMENT_NODE: 1,
    ATTRIBUTE_NODE: 2,
    TEXT_NODE: 3,
    CDATA_SECTION_NODE: 4,
    ENTITY_REFERENCE_NODE: 5,
    ENTITY_NODE: 6,
    PROCESSING_INSTRUCTION_NODE: 7,
    COMMENT_NODE: 8,
    DOCUMENT_NODE: 9,
    DOCUMENT_TYPE_NODE: 10,
    DOCUMENT_FRAGMENT_NODE: 11,
    NOTATION_NODE: 12
  };
}
```

En el código anterior comprobamos si el navegador ha definido el objeto `Node`. En caso de no estar definido, creamos de forma explícita las constantes asociadas a los tipos de nodo. El navegador Internet Explorer 7, no soporta las constantes predefinidas por lo que, si vamos a programar una aplicación web, y usamos estas constantes, debemos definirlas antes.

La realidad en la actualidad, es que los cambios en las versiones de los navegadores son constantes. Por esta razón, no podemos abarcar las múltiples opciones que surgen a la hora de adaptar el código a todos los navegadores y versiones de los navegadores de una compañía. El programador es sometido a la dualidad de programar siguiendo los estándares que propone W3C, que aparentemente sería lo correcto, o seguir ciertas reglas independientes que pueden tener otros navegadores como Internet Explorer (tiene su propia especificación de DOM, aunque ha colaborado con el estándar W3C), puesto que estos navegadores son de uso mayoritario. En el primer caso puede conseguir un efecto rechazo de las páginas que sigan el estándar (al no ser soportadas por el navegador de uso mayoritario). En el segundo caso se aleja del estándar, comprometiendo su código a la voluntad de una entidad determinada.

6.6 USO DE LIBRERÍAS DE TERCEROS

Como hemos visto en el punto anterior, el desarrollo de aplicaciones web en diferentes navegadores, complica sobremanera la implementación del código. Para solucionar este problema nace *cross-browser*. *Cross-browser* es un concepto que nace con la intención de visualizar una página o aplicación web exactamente igual en todos los navegadores. Aplicando este concepto el programador web se ve obligado a unificar y asociar las utilidades que tienen comportamientos diferentes en los navegadores. Para conseguir este objetivo, surgen utilidades que nos permiten unificar los eventos y sus propiedades.

Es importante no confundir el concepto *cross-browser* con el de multinavegador. Este concepto tiene que ver con la posibilidad de que el programador pueda visualizar en diferentes navegadores una aplicación web. De esta manera puede adaptar la aplicación web al mayor número de navegadores posibles. Cabe indicar que existen varias formas de conseguir este objetivo. A continuación vamos a definir algunas de ellas.

- **Renderizar a través de una Web**. Existen páginas web que nos permiten introducir una dirección de una página web y elegir la versión del navegador con el que queremos visualizarlo. Por ejemplo, *netrenderer* nos permite visualizar una página web en las distintas versiones de Internet Explorer. Normalmente este tipo de páginas solo nos muestran una imagen del resultado, por lo que puede resultar complicado en ocasiones solucionar algunos problemas.
- **Programas para renderizar**. Existen programas que nos permiten instalar varias versiones del mismo navegador, como *multi IE* o *Internet Explorer Collection*. Estos programas dan problemas de compatibilidad de versiones con los últimos navegadores.
- **Instalar los navegadores en máquinas virtuales**. Otra opción es instalar las versiones de navegadores en máquinas virtuales, que además estén acorde con los sistemas operativos para los que hay instalables de la versión de navegador.

Para el uso de las utilidades *cross-browser* es necesario implementar funciones que habitualmente vienen definidas en librerías. A continuación vamos a mostrar cómo sería la estructura básica condicional, que comprueba el navegador utilizado y toma una decisión u otra en función de la respuesta.

```
if (window.XMLHttpRequest){
   // codigo para IE7+, Firefox, Chrome, Opera, Safari
   Xmlhttp = new XMLHttpRequest();
}else{
   // codigo para IE6, IE5
   Xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
```

En el código anterior vemos que si la respuesta del `window.XMLHttpRequest` es verdadero, crea un objeto de un tipo. En el caso de que no sea verdadero, crea otro objeto, compatible con Internet Explorer en su versión 5 y 6.

Los pasos para generar las librerías son: crear los objetos que agrupan los objetos y métodos relacionados con los eventos. Crear un método para poder obtener el nuevo objeto evento. Este objeto agrupará los métodos relacionados. En el caso de que exista algún navegador, como Internet Explorer, que le faltan eventos, deberemos añadir al objeto evento de Internet Explorer, los nuevos eventos, para así estandarizarlo. De esta forma iremos comprobando el navegador en el que estamos. Según sea, cada caso de navegador, iremos creando objetos del tipo de navegador, para así poder dar respuesta a los eventos según el comportamiento de la versión y tipo de navegador.

ACTIVIDADES 6.1

- Realice un *script* que recorra y muestre con un `alert` de JavaScript, cada tipo de nodo del árbol de una página HTML, que previamente haya creado.
- Realice dos tablas, una con las constantes de la interfaz `node` y otra con los métodos, indicando los valores que reciben y los que devuelven.
- Haga un pequeño esquema sobre cuáles son las distintas formas de acceder a un nodo en la estructura de árbol.
- Describa en forma de puntos, cuáles son los pasos para crear y eliminar un nodo tipo `element`, que tiene a su vez un valor de texto.
- Enumere los pasos para realizar la inserción de un nuevo nodo en el árbol. Para ello hay que tener en cuenta que la página está cargada completamente.

RESUMEN DEL CAPÍTULO

El modelo de objetos del documento (DOM) es un estándar de la entidad W3C que define cómo acceder a la estructura de documentos HTML y XML. Para ello cuenta con programas de *script*, que permiten acceder y actualizar los datos de forma dinámica en el miso navegador.

El DOM crea una estructura en árbol formada por nodos que permite organizar de una forma ordenada la complejidad de una página HTML. El acceso a la estructura se puede realizar, recorriendo el árbol o recuperando nodos de forma independiente, a partir de su identificador. Podemos recuperar vectores de un tipo de nodo y acceder a ellos a través de un índice.

Existen varios tipos de nodos que representan cada una de las partes que nos vamos a encontrar en una página HTML. Para facilitar el acceder, insertar, modificar o borrar nodos, DOM hace uso de métodos.

Los nodos pueden contener atributos, que también pueden ser creados, modificados y eliminados a través de métodos del modelo DOM.

Para actuar sobre la estructura de árbol de DOM, la página HTML tiene que estar cargada completamente.

Las adaptaciones de código para los diferentes navegadores son complejas. No todos los navegadores hacen la misma interpretación del código. Esto requiere que tengamos en cuenta en que navegador se ejecuta el código, si queremos que sea compatible con diferentes navegadores del mercado.

El uso de los navegadores por parte de los usuarios es vital, para que en el futuro los estándares sigan un camino o cambien de rumbo.

EJERCICIOS PROPUESTOS

1. Se propone realizar una calculadora. El funcionamiento de la misma será el siguiente. Dispondrá de números del 0 al 9 y de los signos, +, -, *, /, =. Para contener cada uno de estos número y signos usaremos una etiqueta `div` (un `div` para cada uno). Existirá otra etiqueta `div` para mostrar el resultado. El funcionamiento de la calculadora será el mismo que el de la calculadora de Windows. Al pulsar en la etiqueta `div` que contenga cada número, el `div` de resultado añadirá el valor a la izquierda del que existía antes, y así sucesivamente, hasta pulsar un signo. Cuando pulsemos el signo se mostrará en la pantalla de resultado. En el siguiente paso volveremos a introducir otro número como antes, hasta que pulsemos la tecla *igual*. Al haber pulsado el *igual* se mostrará el resultado en el `div` de resultado. Se deberá incluir un `div` que nos permite inicializar la calculadora.

TEST DE CONOCIMIENTOS

1 DOM fue utilizado por primera vez por Internet Explorer.
- a) No, la primera vez lo utilizó el navegador Netscape Navigator.
- b) Sí, Internet Explorer colabora habitualmente con la W3C.
- c) El primer navegador de Microsoft que lo utilizó fue la versión 3.0.
- d) Las respuestas *a* y *c* son correctas.

2 Los nodos de tipo `element`:
- a) Tienen todo lo necesario para mostrar una etiqueta con su texto.
- b) Necesitan un nodo hijo de tipo texto para mostrar algún valor.
- c) No son nodos de tipo etiqueta.
- d) No tienen nodos hijo.

3 En una estructura de árbol de nodos:
- a) Cada nodo tiene un padre, excepto el nodo raíz.
- b) Un nodo no puede tener cualquier número de hijos.
- c) Los nodos que no comparten padre pueden ser hermanos.
- d) Las respuestas *a* y *b* son correctas.

4 El nodo raíz de un documento HMTL:
- a) Se llama `DocumentType`.
- b) Se llama `Text`.
- c) Se llama `Document`.
- d) Se llama `DOM`.

5 La propiedad `nodeType` devuelve:
- a) El nombre de la constante definida por la interfaz `Node`.
- b) El número asociado al tipo de nodo.
- c) El tipo de `element` al que corresponde el nodo.
- d) El valor del nodo en ese momento.

6 Si queremos acceder al primer `div` de la estructura del árbol de una página HTML:
- a) Tenemos que recorrer el árbol desde el nodo raíz obligatoriamente.
- b) Podemos acceder a través de un vector que se genera con las etiquetas `div`.
- c) Podemos acceder a través de su identificador.
- d) Las respuestas *b* y *c* son correctas.

7 Los atributos de un nodo:
- a) Son solo atributos de ese nodo.
- b) Pueden ser creados y modificados una vez se carga la página.
- c) No pueden ser accedidos si no se identifica a su nodo.
- d) Todas las respuestas anteriores son correctas.

8 La estandarización por parte de W3C:
- a) Pretende que todos los navegadores sigan las normas del estándar.
- b) Desvincula los intereses particulares de los estándares a seguir.
- c) El éxito de sus estándares, depende en gran medida de los navegadores usados por los usuarios finales.
- d) Todas las anteriores son correctas.

9 Un multinavegador permite al programador:

a) Programar sin preocuparse de que el código se vea bien en todos los navegadores. F

b) Programar permitiendo ver en los distintos navegadores los errores que se puedan ocasionar. F

c) Desvincularse de la programación para varios navegadores.

– d) Conseguir que un navegador interprete el código de una manera correcta.

– Sitios web funcionen en varios navegadores

10 *Cross-Browser* en el mundo de las aplicaciones web:

a) Es un concepto.

b) Hace posible visualizar una aplicación web en distintos navegadores.

c) Unifica y asocia los comportamientos de los eventos.

– d) Todas las anteriores son correctas.

– S.ho web funcione en cualquier navegador y cualquiera de sus versiones

7 Utilización de mecanismos de comunicación asíncrona

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer los mecanismos de comunicación asíncrona en las aplicaciones web.
- ✓ Conocer las tecnologías asociadas con la técnica AJAX y su utilización en el desarrollo de aplicaciones interactivas.
- ✓ Profundizar en los formatos de envío y recepción de información asíncrona.
- ✓ Conocer en detalle la realización de llamadas asíncronas.
- ✓ Describir las librerías de actualización dinámicas actuales.

En este capítulo presentamos los conceptos necesarios para comprender los mecanismos de comunicación asíncrona y las tecnologías asociadas con la interactividad de las páginas web actuales. Así mismo, vamos a profundizar en AJAX (*Asynchronous JavaScript And XML*), la cual es una técnica de programación que tiene como objetivo intercambiar pequeñas cantidades de datos entre el cliente y el servidor en segundo plano recargando partes de la página web sin la necesidad de recargar todo el contenido de la misma.

7.1 MECANISMOS DE COMUNICACIÓN ASÍNCRONA

Actualmente muchas de las páginas web son interactivas, permitiendo la comunicación con el usuario y produciendo un cambio en el recurso recibido. Las aplicaciones web interactivas, como hemos mencionado anteriormente, se basan en que dicha interacción genere un diálogo entre el cliente y el servidor (Connolly, 2000). Desde el punto de vista del modelo de programación, la lógica asociada al inicio y gestión de esta comunicación puede ser ejecutada tanto en el cliente como en el servidor (e incluso en ambos).

En un proceso habitual el cliente es el que inicia el intercambio de información solicitando datos al servidor que responde enviando uno o más flujos de datos al cliente.

Así, si el usuario ingresa a una página web introduciendo una URL en el navegador esperará la respuesta del servidor hasta que el código HTML llegue por completo y se dibuje la página solicitada. En ese caso se está utilizando un mecanismo de comunicación síncrona: el cliente ha enviado una petición y permanece bloqueado esperando la respuesta del receptor.

Ahora imaginemos una página web de compra de billetes de tren. Si solo existiera la comunicación síncrona, una vez realizada la elección del origen, la página debería recargarse completamente para obtener los destinos disponibles para el origen seleccionado.

El mecanismo de comunicación asíncrona permite recargar en segundo plano una parte de la página web, dejando desbloqueado el resto. El cliente que envía una petición no permanece bloqueado esperando la respuesta del servidor. Esto ayuda a que las aplicaciones web tengan una interactividad similar a las aplicaciones de escritorio y es en parte lo que hace algunos años se denomina Web 2.0.

La necesidad de que las aplicaciones web tengan la interactividad y la usabilidad de las aplicaciones de escritorio ha llevado a dar un nuevo uso a tecnologías como XML, CSS o DOM. El 18 de febrero de 2005 J.J. Garrett en (Garret, 2005) habla por primera vez sobre AJAX (*Asynchronous JavaScript And XML*), siglas que en español significan: JavaScript asíncrono y XML. AJAX es una técnica de desarrollo de aplicaciones web que permite la creación de aplicaciones interactivas. Una de las ventajas de este tipo de aplicaciones es que se minimizan las comunicaciones entre el cliente y el servidor, realizándose de manera asíncrona. Por lo tanto, las páginas web suprimen los efectos secundarios de las recargas, como la pérdida del contexto, la ubicación del *scroll* o las respuestas más lentas.

En ese sentido, el mismo Garrett ponía como ejemplo un sitio web de consulta de mapas. Al realizar un acercamiento en el mapa, el usuario no esperaba a que la página se recargara completamente. Por el contrario, la experiencia de acercar el mapa era casi instantánea. Esto es lo mismo que se percibiría en una aplicación de escritorio.

7.1.1 DEFINICIÓN DE AJAX

AJAX indica una técnica y el uso de un conjunto de tecnologías. Tiene por objetivo intercambiar pequeñas cantidades de datos entre el cliente y el servidor recargando partes de la página web sin la necesidad de recargar todo el contenido de la misma. Pero AJAX no es un lenguaje de programación. De acuerdo con Garret, AJAX incluye las siguientes tecnologías (Garrett, 2005):

- ✓ XHTML y CSS para una presentación basada en estándares.
- ✓ DOM para la interacción y la visualización dinámica de datos.
- ✓ XML y XSLT para el intercambio y transformación de datos.
- ✓ XMLHttpRequest para la recuperación asíncrona de los datos.
- ✓ JavaScript como elemento de unión.

Las tecnologías utilizadas en AJAX estaban presentes desde el año 1998, entre sus antecedentes se encuentra el *Scripting Remoto* o los *iframes* desarrollados por Microsoft (Clinick, 1999). Aun así, ha alcanzado popularidad debido a su utilización, principalmente por aplicaciones relacionadas con Google.

7.1.2 ELECCIÓN DE AJAX

Paradójicamente, uno de los problemas más frecuentes actualmente es la programación excesiva con AJAX. Entre las principales precauciones al utilizar AJAX y que pueden recapitularse de (Holdener III, 2008) y (Powell, 2008) encontramos las siguientes:

- En las páginas con AJAX intervienen más tecnologías y, por tanto, son más complejas de desarrollar que las páginas estáticas.
- Al no realizarse la petición de una página completa las peticiones realizadas con AJAX no son registradas de la misma manera en el historial del navegador. Por lo tanto, el comportamiento considerado lógico por el usuario al utilizar la funcionalidad de "volver a la página anterior" no es reproducido de la misma manera.
- Los recursos. Las aplicaciones web o sitios web con AJAX utilizan más recursos del servidor. En este sentido, no todas las peticiones deben realizarse con AJAX, sino solo las necesarias.
- El uso de las tecnologías asociadas con AJAX no están presentes por defecto en cualquier tipo de dispositivos, por ejemplo, dispositivos móviles o PDA.
- Aunque cada vez menos, todavía existen incompatibilidades entre navegadores.

7.1.3 REPASO A LAS TECNOLOGÍAS INVOLUCRADAS

En este apartado repasaremos brevemente las diferentes tecnologías involucradas en AJAX, las características que las hacen indispensables para construir la técnica de programación asíncrona y los aspectos más importantes a tener en cuenta para su correcto uso.

XHTML y CSS

La principal diferencia entre el lenguaje de Internet de las páginas web, el *HyperText Markup Language* (HTML) y su sucesora *eXtensible HyperText Markup Language* (XHTML) es su grado de especificación. Por lo tanto, XHTML (W3C, 2002) es un HTML estándar especificado mediante un documento XML. Siempre se ha asociado a HTML solamente como un lenguaje de etiquetas con una amplia flexibilidad. Si el programador se olvidaba de cerrar una etiqueta, la página web se mostraba igual (o similar). En cambio XHTML es más riguroso con su estructura (justamente, como un fichero XML).

Aquí dejamos algunos ejemplos de la diferencia entre HTML y XHTML:

- Los valores de los atributos, siempre entre comillas:
  - Incorrecto: `<td colspan=2>`.
  - Correcto: `<td colspan="2">`.
- Los nombres de elementos y atributos deben ir en minúsculas:
  - Incorrecto: `<IMG SRC="dibujo.png" /IMG>`.
  - Correcto: `<img src="dibujo.png" /img>`.
- No está permitida la minimización de atributos (se usa el nombre del atributo como valor):
  - Incorrecto: `<textarea readonly>`.
  - Correcto: `<textarea readonly="readonly">`.

Para utilizar AJAX es indispensable la división en partes de la página web y para hacerlo debe estar correctamente construido el código de la página web.

Las hojas de estilo en cascada, o *Cascading Style Sheets* (CSS), indican el diseño y la presentación del texto y los datos de las páginas web. De esta manera, un cambio en el CSS es reflejado instantáneamente en todas las páginas que utilizan algún elemento de presentación identificado en dicha hoja de estilo. Si a esto le sumamos la capacidad de poder ser leído y modificado mediante DOM, vemos la gran importancia que tiene en las aplicaciones con AJAX.

DOM

El modelo de objetos para la representación de documentos, o *Document Object Model* (DOM), es una representación de la página web en una estructura de jerarquía de árbol (W3C, 2003). Así mismo, es esencialmente una interfaz de programación de aplicaciones (también llamado API) que proporciona un conjunto estándar de objetos para representar documentos XHTML, un modelo estándar sobre cómo pueden combinarse dichos objetos y una interfaz estándar para acceder a ellos y manipularlos. Lo importante es que todas las partes de la página están accesibles desde el cliente y no es necesario utilizar tecnologías del lado del servidor. Por ejemplo, un sencillo código HTML como el que vamos a ver a continuación tiene una representación DOM que podemos ver en la Figura 7.1.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Página de Inicio</title>
  </head>
  <body>
    <p>Contenido de la Página</p>
  </body>
</html>
```

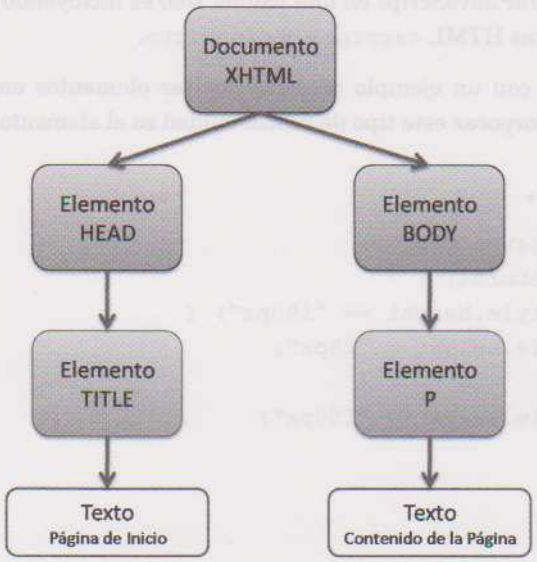


Figura 7.1. *Ejemplo de árbol DOM*

Ahora bien, la fortaleza de esta representación jerárquica está en su grado de estandarización y en que todos los navegadores lo utilicen. El DOM es un estándar del *World Wide Web Consortium*, también conocido como W3C (*www.w3.org*) y, por tanto, se considera un estándar que deben seguir todos los navegadores para representar las páginas web. Esto garantiza que, modificando el color del borde de un botón mientras pasamos el cursor sobre el elemento, funcione de la misma manera tanto en Internet Explorer como en Safari, Firefox, etc. Por lo tanto, mediante DOM y un lenguaje de *script* del lado del cliente (el más utilizado es JavaScript) podemos agregar o modificar elementos del árbol DOM con la particularidad de que aparecerán inmediatamente en el navegador.

Es necesario comprobar la compatibilidad de las páginas web mostradas. Aunque cada vez se encuentran más estandarizadas este tipo de tecnologías, todavía son comunes las visualizaciones diferentes entre navegadores. En ese sentido, existen herramientas para determinar los problemas de compatibilidades y comparar visualizaciones. Algunas de las más conocidas son: *Adobe® BrowserLab*, *Browser Shots* o *Cross Browser Testing*.

JavaScript

El lenguaje de *scripting* JavaScript es, como hemos dicho anteriormente, el más utilizado actualmente en los navegadores. Lo podemos catalogar como un lenguaje orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico (Zakas, 2009).

Su utilización principal es en el lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario, páginas web dinámicas y accediendo al árbol DOM para realizar modificaciones instantáneas del código fuente. Por lo tanto, el código JavaScript se ejecuta en el propio navegador y va literalmente formando parte del propio código HTML de dicha página. Por lo tanto, mientras se escribe la página web HTML, también se escribe el código JavaScript.

La forma más usual de encontrar JavaScript en una página web es incluyendo las sentencias y funciones dentro de la página utilizando las etiquetas HTML `<script>` y `</script>`.

A continuación trabajaremos con un ejemplo de cómo ocultar elementos en una página HTML. Uno de los elementos más utilizados para incorporar este tipo de funcionalidad es el elemento HTML `div`.

```
<html>
  <head>
    <script type="text/javascript">
      function cambiarAltura() {
        if (novedades.style.height == "150px") {
          novedades.style.height = "25px";
        }else{
          novedades.style.height = "150px";
        }
      }
    </script>
  </head>
  <body>
    <div id="novedades" style="height:150px; border:1px;">
      <a href="#" onClick="cambiarAltura()">X</a>
      <p>Alerta: La línea de bus 25 cambia su recorrido</p>
    </div>
  </body>
</html>
```

Analizando el código anterior, tenemos una función JavaScript que al ser ejecutada realiza el cambio de tamaño del elemento `div`. Comprobamos mediante una sentencia condicional `if` el tamaño actual del elemento `div` cuyo identificador es `"novedades"`. De hecho, modificamos directamente al atributo altura `(height)` del elemento novedades.

```
if (novedades.style.height == "150px"){
  novedades.style.height = "25px";
} else{
  novedades.style.height = "150px";
}
```

La sentencia `novedades.style.height` hace referencia al elemento `div` que tiene el atributo `id` con el valor `"novedades"`, este `div` también tiene un atributo `style` y dentro de este atributo la propiedad `height` de CSS. Lo que hacemos aquí es cambiar esta propiedad CSS.

Ahora bien, si analizamos el resto del código fuente podemos ver el mecanismo de llamada de la función JavaScript. Dentro del elemento `div` tenemos un enlace, que dispara la función JavaScript utilizando el atributo `onclick`.

```
<a href="#" onClick="cambiarAltura()">X</a>
```

XML y XSLT

El lenguaje XML es utilizado para describir y estructurar datos. Alrededor de XML también encontramos otras tecnologías, por ejemplo para realizar búsquedas dentro del documento XML o para realizar transformaciones (W3C, 2011). En ese sentido, navegadores como Internet Explorer o Firefox contienen funcionalidades internas para trabajar con documentos XML.

Entre las tecnologías asociadas con XML encontramos el *EXtensible Stylesheet Language Transformations* (XSLT), el cual es un lenguaje para transformar un documento XML en otro documento XML, texto plano, etc. Por lo tanto, con XSLT podrías recibir un documento XML representando una tabla y transformarlo en una tabla XHTML (ver Figura 7.2). Como siguiente paso, podríamos incluir la tabla en el árbol DOM mediante JavaScript.

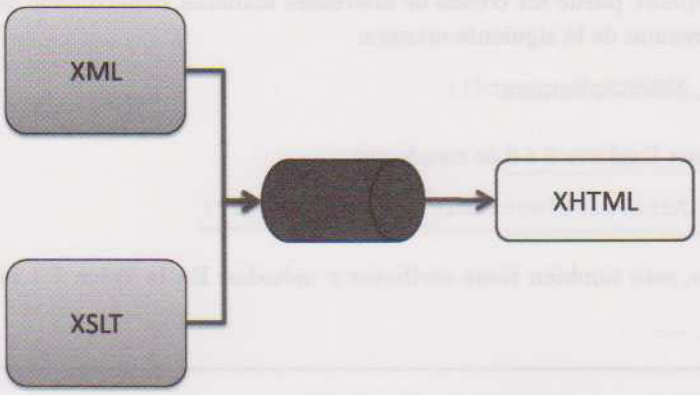


Figura 7.2. *Esquema simple de procesamiento XSLT*

XSLT utiliza otro lenguaje, el XPath, para realizar consultas en el documento XML cuando se aplican las transformaciones. En este caso, XPath busca elementos dentro del XML de entrada y XSLT los procesa.

¿SABÍAS QUE...?

XML y HTML son aplicaciones de un lenguaje genérico de etiquetado denominado SGML (*Standard Generalized Mark-up Language*).

El objeto XMLHTTPRequest

EL objeto `XMLHttpRequest` aparece a partir de Internet Explorer 5 en la forma de un control ActiveX llamado XMLHttp. Estos elementos (llamado en la actualidad solo "objeto") se fueron transformando en un estándar de facto en navegadores como Firefox, Safari y Opera. Aun así, uno de los problemas más importantes es la estandarización total de este objeto.

Actualmente el objeto `XMLHttpRequest` se encuentra descrito por el *World Wide Web Consortium* (W3C, 2010) y sirve como una interfaz con la que se realizan peticiones a servidores web. Usualmente las librerías que implementan

esta interfaz proporcionan una clase a ser instanciada por la aplicación cliente. A continuación describimos un ejemplo de instanciación del objeto `XMLHttpRequest` en JavaScript:

```
var httpRequest;
if (window.XMLHttpRequest){
  httpRequest = new XMLHttpRequest();
}else if (window.ActiveXObject){
  httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Un objeto `XMLHttpRequest` puede ser creado de diferentes maneras, dependiendo del tipo de navegador. En los navegadores actuales lo creamos de la siguiente manera:

```
httpRequest = new XMLHttpRequest();
```

Mientras que en Internet Explorer 5 ó 6 es creado así:

```
httpRequest = new ActiveXObject("Microsoft.XMLHTTP")
```

Como cualquier objeto, este también tiene atributos y métodos. En la Tabla 7.1 presentamos los principales atributos y su definición.

Tabla 7.1 Atributos del objeto XMLHttpRequest

Atributo	Descripción
`readyState`	Devuelve el estado del objeto como sigue: 0 = sin inicializar, 1 = abierto, 2 = cabeceras recibidas, 3 = cargando y 4 = completado.
`responseBody`	Devuelve la respuesta como un *array* de bytes.
`responseText`	Devuelve la respuesta como una cadena.
`responseXML`	Devuelve la respuesta como XML. Esta propiedad devuelve un objeto documento XML, que puede ser examinado usando las propiedades y métodos del árbol del *Document_Object_Model*.
`status`	Devuelve el estado como un número (p. ej. 404 para "Not Found").
`statusText`	Devuelve el estado como una cadena (p. ej. "Not Found").

Así mismo, en la Tabla 7.2 y la Tabla 7.3 podemos ver los principales métodos y propiedades respectivamente. La secuencia de utilización normal consiste en abrir un canal de comunicación y especificar la petición con el método `open()`, enviar la petición con el método `send()`, agregar la función que se disparará en el evento `onreadystatechange` y obtener la respuesta en los diferentes atributos del objeto `XMLHttpRequest` (como una cadena, como un documento XML o como un *array* de bytes).

Tabla 7.2 Métodos del objeto XMLHttpRequest

Métodos	Descripción
`abort()`	Cancela la petición en curso.
`getAllResponseHeaders()`	Devuelve el conjunto de cabeceras HTTP como una cadena.
`getResponseHeader(cabecera)`	Devuelve el valor de la cabecera HTTP especificada.
`open( método, URL [, asíncrono[, nombreUsuario [, clave]]] )`	Especifica el método, URL y otros atributos opcionales de una petición. El parámetro de método puede tomar los valores `"GET"`, `"POST"`, o `"PUT"` (`"GET"` y `"POST"` son dos formas para solicitar datos, con `"GET"` los parámetros de la petición se codifican en la URL y con `"POST"` en las cabeceras de HTTP). El parámetro URL puede ser una URL relativa o completa. El parámetro asíncrono especifica si la petición será gestionada asíncronamente o no. Un valor `true` indica que el proceso del script continúa después del método `send()`, sin esperar a la respuesta, y `false` indica que el *script* se detiene hasta que se complete la operación, tras lo cual se reanuda la ejecución. En el caso asíncrono se especifican manejadores de eventos, que se ejecutan ante cada cambio de estado y permiten tratar los resultados de la consulta una vez que se reciben, o bien gestionar eventuales errores.
`send([datos])`	Envía la petición.

El manejo de eventos en este caso es muy importante. Hay que recordar que estamos trabajando en segundo plano, por lo que la página web no quedará bloqueada a la espera de una respuesta. En cambio, es necesario dar respuesta a los diferentes eventos que pueden ocurrir. El principal es el cambio de estado de la petición.

Tabla 7.3 Propiedades del objeto XMLHttpRequest

Propiedades	Descripción
`onreadystatechange`	Evento que se dispara con cada cambio de estado.
`onabort`	Evento que se dispara al abortar la operación.
`onload`	Evento que se dispara al completar la carga.
`onloadstart`	Evento que se dispara al comenzar la carga.
`onprogress`	Evento que se dispara periódicamente con información de estado.

La contribución del objeto XMLHttpRequest es clara, permite el envío y recepción de información en segundo plano. Por lo tanto, permite la comunicación asíncrona entre el cliente y el servidor web.

7.1.4 PERSPECTIVA GLOBAL DE UN DESARROLLO AJAX

La tecnología clave para que AJAX funcione es el lenguaje de scripts JavaScript debido principalmente a que es soportado por la mayoría de los navegadores. A partir de JavaScript podríamos obtener los datos desde el servidor de forma asíncrona, ya sea en formato XML o en texto plano. La perspectiva global del funcionamiento de AJAX, representada en la Figura 7.3, es el siguiente:

- La página recibida por el cliente contiene código JavaScript, el cual puede obtener datos del servidor. El navegador, al recibir la página web crea el DOM asociado, el cual puede ser accedido y modificado por el código JavaScript.
- Cuando la página necesita información del servidor (por ejemplo, cuando hemos elegido el origen de un viaje y es necesario el listado de destinos posibles), desde el lenguaje JavaScript se utiliza un elemento especial: el objeto `XMLHttpRequest`. A través de este elemento se envía una petición al servidor sin provocar que la página sea recargada. Esta es la clave, la ejecución de JavaScript en el cliente dispara una comunicación asíncrona, no bloquea la página hasta la recepción de los datos (en el caso del ejemplo, los destinos posibles para el origen elegido) y el usuario puede seguir trabajando en la página.
- Los datos obtenidos del servidor son, por lo general, objetos XML o texto plano que será leído por el código JavaScript.

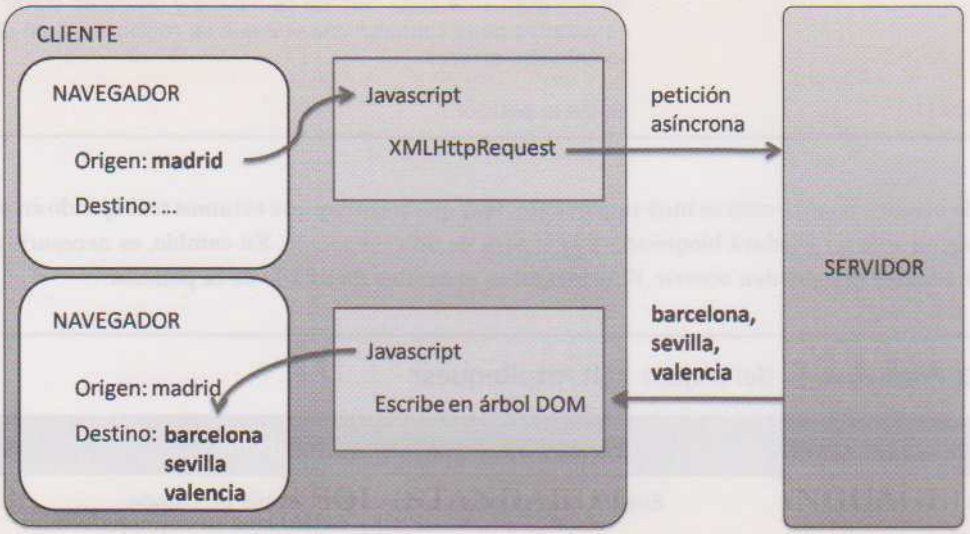


Figura 7.3. Funcionamiento de AJAX

Por lo tanto, al desarrollar una página web con AJAX tendremos elementos adicionales al HTML (ver Figura 7.3). Por un lado, el agregado de código JavaScript para obtener los datos del servidor. Por otro lado, código del lado del servidor que responderá a las peticiones asíncronas JavaScript (por ejemplo PHP, ASP .Net o JSP).

ACTIVIDADES 7.1

- Identifique ejemplos de utilización de AJAX en las páginas web actuales.
- XSLT permite realizar transformaciones a partir de documentos XML. Investigue el tipo de declaración utilizada por el lenguaje XSLT y en qué se diferencia con un lenguaje imperativo como C, Java o VB .Net.
- Uno de los principales problemas al programar de manera asíncrona es la compatibilidad entre los diferentes navegadores. Describa brevemente esos problemas.

7.2 FORMATOS PARA EL ENVÍO Y RECEPCIÓN DE INFORMACIÓN. XML Y JSON

JSON, en el acrónimo de *JavaScript Object Notation* y podemos verlo desde dos puntos de vista. Por un lado es un formato ligero para el intercambio de datos y, por otro lado, es una manera de almacenar información. El creador de este tipo de notación es Douglas Crockford y fue pensado en un primer momento como una alternativa a XML (Crockford, 2006).

Para el intercambio de datos, XML no es lo suficientemente eficiente. Los documentos XML tienen una gran cantidad de información extra, asociada a su estructura. La simplicidad de JSON ha dado lugar a la generalización de su uso, especialmente como alternativa a XML en AJAX. JSON está constituido por dos estructuras:

- ✓ Una colección de pares de nombre/valor.
- ✓ Una lista ordenada de valores.

Estas son estructuras universales y ampliamente soportadas. Por ello, es razonable que un formato de intercambio de datos independiente del lenguaje de programación se base en estas estructuras. A continuación describiremos la sintaxis de JSON.

7.2.1 SINTAXIS DE JSON

La sintaxis de JSON es relativamente sencilla. Antes de dar algunos ejemplos de JSON y compararlo con XML describiremos los elementos más importantes.

El elemento base de la sintaxis es el `object`. Está conformado por un conjunto desordenado de pares nombre/valor. Un objeto comienza con una llave de apertura y finaliza con una llave de cierre. Cada nombre es seguido por dos puntos, estando los pares nombre/valor separados por una coma (ver Figura 7.4).

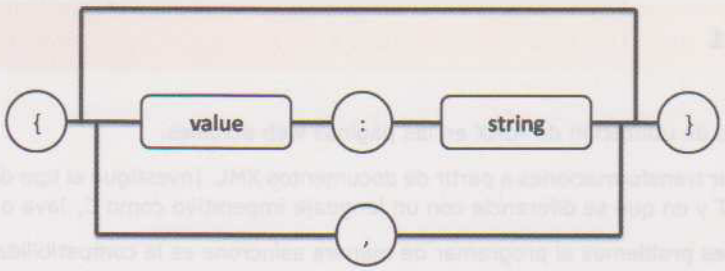


Figura 7.4. Sintaxis de un elemento `object` en JSON

Como siguiente elemento están los *array*. Un *array* es una colección de elementos *values*. Comienza por un corchete izquierdo y termina con un corchete derecho. Los elementos *value* se separan por una coma (ver Figura 7.5).

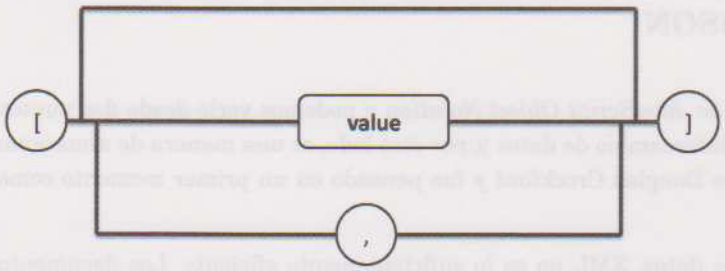


Figura 7.5. Sintaxis de un elemento array en JSON

A su vez un elemento *value* puede ser una cadena de caracteres o *string* con comillas dobles, un *number*, los valores booleanos *true* o *false*, *null*, un *object* o un *array*. Estas estructuras pueden anidarse (ver Figura 7.6).

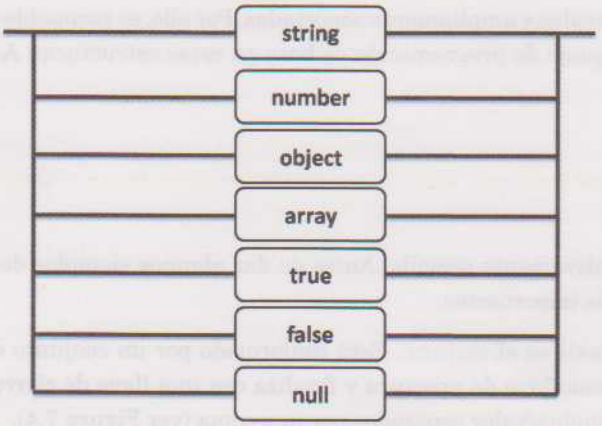


Figura 7.6. Sintaxis de un elemento value en JSON

Un *string* es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Una cadena de caracteres es parecida a una cadena de caracteres en C o Java. Un elemento *number* es similar a un número en C o Java, excepto que no se usan los formatos octales y hexadecimales. Por último, los espacios en blanco pueden insertarse entre cualquier par de símbolos.

7.2.2 EJEMPLOS DE INTERCAMBIO DE DATOS CON JSON Y XML

A continuación veremos dos ejemplos de datos estructurados en JSON y XML, para ver sus diferencias.

Estructura de datos en JSON:

```
{
  'nombre':'pepe',
  'edad':34,
  'domiclio':'calle alcalá 1',
  'estudios':['primario','secundario','universitario']
}
Estructura de datos en XML:
<ciudadano>
  <nombre>pepe</nombre>
  <edad>34</edad>
  <domicilio>calle alcalá 1</domicilio>
  <estudios>
    <estudio>primario</estudio>
    <estudio>secundario</estudio>
    <estudio>universitario</estudio>
  </estudios>
</ciudadano>
```

Como hemos podido observar la diferencia de peso es importante, sobre todo para el uso con AJAX. Además, si vemos cómo se usa cada uno encontraremos más ventajas. La utilización de JSON es recomendable por su facilidad y en el caso de serializar una estructura de datos simple. Así mismo, en caso de utilizar los datos directamente, JSON lo permite de una manera más sencilla.

La utilización de XML es recomendable para los casos más genéricos. Recordemos que tenemos la opción de utilizar otras tecnologías asociadas a XML como el XSTL. En caso de que la utilización no sea tan directa y necesitemos un amplio procesamiento de los datos obtenidos, XML puede ser la mejor opción. Así mismo, XML actualmente tiene mayor soporte y herramientas de desarrollo. En el caso de JSON es necesario incluir analizadores en el lado del servidor.

Aunque hemos comparado y dado pautas para seleccionar JSON o XML, también es frecuente el uso de JSON y XML unidos. Por ejemplo, una aplicación de cliente que integra datos de *Google Maps* con datos meteorológicos en SOAP hacen necesario soportar ambos formatos.

¿SABÍAS QUE...?

JSON está basado en un subconjunto del lenguaje de programación JavaScript

ACTIVIDADES 7.2

- Una vez obtenida información estructurada del lado del servidor (XML o JSON), es necesario realizar su tratamiento y utilización posterior. Investigue cuáles son los pasos necesarios para cargar una lista desplegable con elementos obtenidos de XML y de JSON.

7.3 EJEMPLO DE COMUNICACIÓN ASÍNCRONA

En esta sección presentamos un ejemplo práctico de cómo utilizar AJAX en una página web. A partir del ejemplo, analizaremos los diferentes pasos a seguir, la utilización de estructuras de datos como XML o JSON, las notificaciones recibidas por el objeto `XMLHttpRequest` y la interacción con la página web.

```
<html>
 <head>
  <title>Ejemplo de AJAX</title>
  <script language = "javascript">
    var objetoXHR = false;

    if (window.XMLHttpRequest) {
     objetoXHR = new XMLHttpRequest();
    }else if (window.ActiveXObject) {
     objetoXHR = new ActiveXObject("Microsoft.XMLHTTP");
    }

    function obtenerDatosServidor(origen, elemento) {
     if(objetoXHR) {
      var objeto_destino = document.getElementById(elemento);
      objetoXHR.open("GET", origen);
      objetoXHR.onreadystatechange = respuesta();
      objetoXHR.send(null);
     }
    }

    function respuesta(){
     if (objetoXHR.readyState == 4 &&
      objetoXHR.status == 200) {
      objeto_destino.innerHTML = objetoXHR.responseText;
     }
    }
```

```
  </script>
 </head>
 <body>
  <H1>Ejemplo de AJAX</H1>
  <form>
   <input type = "button" value = "Buscar información"
     onclick = "obtenerDatosServidor('http://Web/datos.txt',
     'elemento_destino')">
  </form>
  <div id="elemento_destino">
   <p>La información aparecerá aquí</p>
  </div>
 </body>
</html>
```

El funcionamiento de esta aplicación es sencillo. La página web muestra un botón el cual, cuando hacemos clic sobre él, muestra un mensaje en un elemento `div` cambiando el texto que se encontraba anteriormente. Por último, la información que se muestra la obtenemos desde el servidor web de manera asíncrona.

Ahora pasemos a identificar las diferentes partes del ejemplo. El cuerpo de la página web contiene un elemento `div` cuyo identificador es `"elemento_destino"` (elegimos este nombre por ser representativo de la función con la cual utilizaremos este elemento).

```
<div id="elemento_destino">
  <p>La información aparecerá aquí</p>
</div>
```

También existe un botón en esta página, de tal manera que cuando el usuario hace clic en el botón llamará a una función JavaScript denominada `"obtenerDatosServidor"`.

```
<form>
  <input type = "button" value = "Buscar información"
    onclick = "obtenerDatosServidor('http://Web/datos.txt',
    'elemento_destino')">
</form>
```

La función `"obtenerDatosServidor"` contiene dos parámetros. Por un lado la URL de un texto (`datos.txt`) y por otro lado el nombre del elemento `div`. El fichero `datos.txt` contiene el texto plano que vamos a mostrar en la página web, que proviene del servidor web y que es descargado en segundo plano.

Analizando el código JavaScript de nuestro ejemplo podemos ver la declaración de la variable `objetoXHR` en la primera línea (donde XHR son las siglas por las que se conoce al objeto `XMLHttpRequest`). La variable la inicializamos a valor booleano falso para poder verificar posteriormente si el objeto ha sido creado.

Así mismo, incorporamos un código inicial para comprobar el tipo de navegador y, por tanto, instanciar el objeto `XMLHttpRequest` de la manera correcta. Para ello tenemos en cuenta que, dependiendo del navegador, el objeto

`XMLHttRequest` pertenecerá a la implementación del mismo navegador y más precisamente formará parte de su elemento `window`. Por lo tanto, preguntamos por la existencia del elemento `window.XMLHttpRequest` y en caso afirmativo instanciamos un objeto. En otro caso, estaremos ante un navegador Internet Explorer 5 ó 6 por lo que deberemos instanciar el objeto `XMLHttpRequest` a partir de un objeto ActiveX. Como resultado de este código fuente tendremos un objeto `XMLHttpRequest` instanciado en la variable `objetoXHR`.

```
var objetoXHR = false;

if (window.XMLHttpRequest){
  objetoXHR = new XMLHttpRequest();
} else if (window.ActiveXObject){
  objetoXHR = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Esta porción del código fuente nos intenta mostrar la complejidad existente al utilizar AJAX de manera nativa, sin la ayuda de librerías. Y cabe resaltar que ha sido solamente un ejemplo simple. Ahora bien, el objetivo del código sigue siendo refrescar el contenido de la página con texto del servidor y sin necesidad de refrescar toda la página web. Todo esto sucede en la función `"obtenerDatosServidor"`. La función empieza verificando que la variable `objetoXHR` contiene un objeto `XMLHttpRequest` válido (distinto del valor booleano falso dado al principio). El siguiente paso es utilizar el método `open()`, con la información pasada en los parámetros de la llamada a la función JavaScript `"obtenerDatosServidor"`.

```
function obtenerDatosServidor(origen, elemento) {
  if(objetoXHR) {
    var objeto_destino = document.getElementById(elemento);

    objetoXHR.open("GET", origen);
    objetoXHR.onreadystatechange = respuesta();
    objetoXHR.send(null);
  }
}
```

El primer parámetro pasado, denominado `"origen"` es la URL desde donde obtendremos el texto a ser mostrado. Para enviar datos desde el cliente al servidor podemos utilizar los mismos métodos utilizados por los formularios de la página web: *GET*, *POST* o *PUT* (Zakas, 2009). Usualmente vamos a utilizar *GET* para enviar poca información y *POST* para el envío de gran cantidad de información desde el cliente al servidor. En este caso se utilizará *GET*. El método `open()` configurará el objeto `XMLHttpRequest` con la URL pasada pero no realizará todavía la conexión. Esta la realizamos con el método `send()`.

El segundo parámetro es el identificador del elemento destino. En este caso lo hemos denominado `"elemento"` y contiene el valor dado al llamar a la función a partir del clic en el botón. Este valor es: `"elemento_destino"`, que se corresponde con el identificador del elemento `div` a ser modificado.

Por defecto, la conexión a la URL se realizará de forma asíncrona, lo que significa que esta sentencia de código no esperará hasta que la conexión se realice y los datos sean obtenidos. Ahora bien, en toda conexión asíncrona la pregunta principal es: ¿Cómo se notifica la finalización de la comunicación? En este caso, cuando los datos sean

obtenidos completamente. Revisando nuevamente las propiedades del objeto `XMLHttpRequest`, existe una llamada `onreadystatechange` a partir de la cual se manejan las operaciones asíncronas. Si asignamos el nombre de una función JavaScript en esta propiedad, esta función será llamada cada vez que el estado del objeto `XMLHttpRequest` cambie. En este caso, a la función la denominamos `"respuesta"`.

Por lo tanto, esta función será llamada también una vez los datos terminen de cargarse. En este momento, tenemos que tener en cuenta dos propiedades del objeto `XMLHttpRequest`: `readyState` y `status` (ver Figura 7.1). La propiedad `readyState` nos indicará cuando se han terminado de recibir los datos, tomando el valor `"4"`. La propiedad `status` nos indica si la comunicación con la URL ha sido correcta, o por el contrario existe algún problema. En caso de recibir el valor `"200"`, quiere decir que la comunicación es correcta. A continuación mostramos el código dentro de la función `"respuesta"` donde verificamos el estado del objeto `XMLHttpRequest` mediante el cual se ha realizado la comunicación asíncrona.

```
if (objetoXHR.readyState == 4 && objetoXHR.status == 200)
```

Ahora como último paso recuperamos los datos del servidor y se los asignamos al contenido del elemento `div` del cual ya conocemos su identificador. En el objeto `XMLHttpRequest` es posible leer los datos obtenidos del servidor a partir de una de sus propiedades. En este caso, para obtener texto plano, preguntamos por la propiedad `responseText`.

```
objeto_destino.innerHTML = objetoXHR.responseText;
```

7.3.1 COMUNICACIÓN CON XML

En esta sección introduciremos aspectos de XML, recordando que la letra "X" de las siglas AJAX se refiere a XML. El formato XML puede utilizarse tanto para la recepción como para la envío de datos al servidor. En cuanto a la recepción de datos XML será necesario que el mismo servidor conteste en dicho formato y que al momento de leer los datos obtenidos hagamos uso de la propiedad `responseXML` del objeto `XMLHttpRequest`.

```
if (objetoXHR.readyState == 4 && objetoXHR.status == 200) {
  var xmlDocument = objetoXHR.responseXML;
```

Para el envío de datos hacia el servidor en formato XML necesitamos dos pasos. Primero, es necesario indicar el formato de envío a partir del método `setRequestHeader()` del objeto `XMLHttpRequest`.

```
objetoXHR.setRequestHeader("Content-Type", "text/xml")
```

Una vez que hemos realizado la modificación, solo es necesario enviar el texto XML al servidor utilizando el método `send()`.

```
objeto.send("<documento><xml>contenido</xml><documento>");
```

Ahora bien, en caso de trabajar con JSON, debido a que es una tecnología relacionada con la notación de JavaScript, los datos enviados por el servidor pueden ser tratados de una forma nativa. Por ejemplo, para convertir un texto JSON en un objeto JavaScript tenemos que utilizar la función `eval()` que invoca al compilador de JavaScript. El único detalle es que el texto debe estar entre paréntesis.

```
var objeto = eval('(' + texto_JSON + ')');
```

Como podemos ver, la utilización de esta función conlleva una grave amenaza a la seguridad. El uso de `eval()` está indicado solo en el caso de páginas web y servidores confiables. En otro caso es recomendable utilizar un parser JSON. Un parser JSON solo reconocerá texto JSON y en el caso de que el navegador lo implemente de manera nativa, será incluso más rápido que la función `eval()`.

```
var objeto = JSON.parse(texto_JSON);
```

También existe la posibilidad de serializar estructuras de datos JavaScript en texto JSON utilizando el método `stringify()`.

```
var texto_JSON = JSON.stringify(objeto);
```

Ambos métodos aceptan parámetros opcionales para realizar las transformaciones de los valores con un mayor nivel de detalle.

ACTIVIDADES 7.3

- Hemos visto la manera en la que el servidor web envía información en texto plano invariante. Uno de los puntos fuertes de esta tecnología es personalizar la página web cliente dependiendo de las interacciones con la misma. Es por ello que la información obtenida del servidor debe ser personalizada de acuerdo con información enviada en la petición. Investigue cómo realizar este tipo de personalización en el lado del Servidor, por ejemplo, a partir de lenguajes del lado del servidor: PHP, JSP, ASP, etc.

7.4 LIBRERÍAS DE ACTUALIZACIÓN DINÁMICA

Como hemos podido ver en las secciones anteriores, es necesario tener en cuenta una gran cantidad de variables a la hora de construir páginas web con características interactivas. Es por ello que junto con la tecnología AJAX están las librerías que implementan una gran cantidad de funciones y controles a ser utilizados por los desarrolladores.

A la hora de seleccionar las herramientas que un desarrollador tiene a su disposición es necesario hacer una primera clasificación de los instrumentos involucrados en función de sus capacidades. Entre las características recomendables que debemos tener en cuenta a la hora de elegir una librería en el lado del cliente se encuentran las siguientes (Holdener III, 2008) (Powell, 2008) (Zakas, 2009):

- Independiente de la tecnología del servidor (por ejemplo, PHP, JSP, ASP, etc.).
- Manejar de manera transparente las incompatibilidades de los diferentes navegadores.
- Manejar la comunicación asíncrona, sin necesidad de realizar la gestión de las operaciones de bajo nivel, como por ejemplo el manejo de estados y de tipos de errores.
- Acceso sencillo al árbol DOM.
- Información de errores para facilitar su utilización al desarrollador.
- Proporcionar controles y objetos gráficos configurables, como por ejemplo: botones, calendarios, campos de texto, etc.

En ese sentido, las librerías que actualmente pueden encontrarse nos ayudarán en el día a día de la programación. De hecho, muchas de ellas son soportadas por los entornos de desarrollo y están respaldadas por comunidades de desarrolladores y de usuarios que comparten su experiencia, código fuente y respuestas.

Entre las librerías más utilizadas para el desarrollo web están los denominados *frameworks JavaScript*. Como hemos visto en los ejemplos anteriores uno de los retos consiste en facilitar la programación del lado del cliente, por lo que podemos encontrar ejemplos genéricos o asociados con tecnologías propietarias.

Por ejemplo, podemos encontrar la librería ASP.Net AJAX la cual es una extensión de controles ASP.Net que implementa funcionalidades AJAX o el Spry framework, una librería desarrollada por Adobe para la construcción de aplicaciones web. También existen otras librerías como: MooTools, DOJO Toolkit, Ext, YUI o Prevel. A continuación detallamos dos de las más utilizadas hasta el momento. JQUERY, PROTOTYPE

7.4.1 JQUERY

Es una librería JavaScript del año 2006 con licencia MIT y GPL, por lo que podemos utilizarla tanto en entornos libres como de software comercial (Bibeault, 2010). A partir de la librería podemos acceder al árbol DOM de una manera más amigable, así como el manejo del contenido y eventos de la página, crear efectos visuales o modificar el CSS. De igual forma, puede trabajar con JSON y contiene componentes visuales como cuadros de diálogo, paneles colapsables o calendarios.

La forma básica de interactuar es mediante la función `$()` que recibe como parámetro el identificador de un elemento HTML o el nombre de una etiqueta HTML. Por ejemplo, a continuación podemos ver el código fuente jQuery para agregar efectos visuales a un elemento con identificador "`boton_1`". El efecto visual está representado por la función `fadeOut()` la cual mostrará el elemento HTML con difuminación.

```
$('#boton_1').fadeOut();
```

Otra de las ventajas de esta librería es la gran cantidad de complementos (como plugins) que podemos encontrar. Uno de ellos es el jsTree, una librería Javascript para crear listas jerárquicas (también llamado *treeview*) desde diferentes fuentes de datos, como por ejemplo XML o JSON.

Entre las características de este complemento de jQuery encontramos:

- ✓ Soporta efectos complejos como los de arrastrar y soltar.
- ✓ Soporte de temas y de manejo directo con el teclado.
- ✓ Funciones de búsqueda.
- ✓ Animaciones.
- ✓ Edición en el mismo control.

A continuación describiremos un ejemplo de utilización de esta librería. Para este ejemplo haremos uso de un texto plano con formato JSON llamado por el control *treeview*. El fichero de texto en formato JSON (el cual podemos obtenerlo como una respuesta del objeto `XMLHttpRequest`) lo detallamos a continuación:

```
{
  attributes: {id: "1",rel: "HD"},
  data: "C:",
  icons: "imagenes/disco.png",
  state: "open",
  children:[
    {
      attributes:{id: "1.1",rel: "carpeta" },
      data: "Mis Documentos",
      icons: "imagenes/carpeta.png"
    },
    {
      attributes:{id: "1.2",rel: "carpeta" },
      data: "Mi Música",
      icons: "imagenes/carpeta.png"
    },
    {
      attributes:{id: "1.3",rel: "carpeta" },
      data: "Mis Vídeos",
      icons: "imagenes/carpeta.png"
    }
  ]
}
```

El fichero JSON está compuesto por un conjunto de atributos anidados (*array* `children`) con la información de los nombres de cada elemento (valor `data`) y el icono asociado (valor `icons`). Como paso final utilizamos la funcionalidad dada por el componente jsTree para cargar el control *treeview*. Instanciamos un nuevo componente, en este caso llamado "`arbol`" e inicializamos el objeto con el método `init()`, cuyos parámetros son el identificador del elemento HTML donde se cargará el control y los datos a ser mostrados (en este caso se carga un fichero con la estructura del árbol a ser mostrado, por ejemplo, un fichero llamado "`árbol.txt`").

```
arbol = new tree_component();

arbol.init($("#identificador"),
  {data:{type: "json", url: "arbol.txt"}}
);
```

7.4.2 PROTOTYPE

Esta librería también se encuentra orientada al desarrollo dinámico de aplicaciones web, implementando técnicas de AJAX (Orchard et. al, 2010). En este caso, sus características son similares a jQuery, con la función básica `$()` o también llamada "función dólar".

Así mismo, define otro tipo de funciones que pueden ser tomadas como atajos a la programación de JavaScript. En cuanto a la tecnología AJAX, también define una serie de funciones. A continuación pondremos dos ejemplos.

Por un lado está la función `AjaxRequest()` para realizar y procesar peticiones AJAX. La función `AjaxRequest()` puede recibir dos parámetros, el primero es la URL donde realizar la solicitud y el segundo parámetro es un *array* de opciones para especificar los valores de la solicitud.

```
new Ajax.Request('URL', {
  method: 'get',
  asynchronous: true,
  onSuccess: funcion_respuesta,
  onFailure: funcion_error
});
```

Por lo tanto, si no existe ningún problema en la llamada asíncrona se ejecutará la función "`funcion_respuesta`". En este caso, el primer parámetro que pasamos a la función es el objeto `XMLHttpRequest` (que como hemos visto en el ejemplo anterior no se instancia explícitamente).

```
function funcion_respuesta(objeto_respuesta) {
  alert(objeto_respuesta.responseText);
}
```

A partir de esta función construimos otras como `Ajax.Updater()` la cual tiene como parámetro adicional el identificador de un elemento HTML. En este caso, se actualizará el contenido del elemento HTML con la respuesta enviada por el servidor. Así mismo, existe también la función `Ajax.PeriodicalUpdater()`, la cual es una versión especializada de `Ajax.Updater()`, que se emplea para ejecutar de forma repetitiva una llamada a `Ajax.Updater()`. Esta función puede sernos útil para ofrecer información en tiempo real, como por ejemplo un canal de noticias.

ACTIVIDADES 7.4

- Investigue librerías gráficas que hagan uso de AJAX para mostrar imágenes.

RESUMEN DEL CAPÍTULO

En este capítulo hemos presentado los conceptos necesarios para comprender los mecanismos de comunicación asíncrona y las tecnologías asociadas con la interactividad de las páginas web actuales, profundizando en el concepto de AJAX. Hemos visto las siguientes tecnologías y su relación en la realización de la comunicación asíncrona y las aplicaciones web interactivas.

- XHTML y CSS se utilizan para una presentación basada en estándares y que asegure su correcta estructuración y recorrido.
- DOM se utiliza para la interacción, la visualización dinámica de datos y la estructuración de las páginas XHTML de tal manera que pueden ser modificadas con velocidad.
- XML y JSON se utilizan para el intercambio y tratamiento de datos entre el cliente y el servidor.
- XMLHttpRequest para el manejo y la recuperación asíncrona de los datos entre cliente y servidor.
- JavaScript como elemento de unión respecto del resto de las tecnologías y como marco de trabajo del lado del cliente.

Las librerías JavaScript que implementan características de la técnica AJAX y permiten realizar desarrollos disminuyendo la complejidad de la gestión de la comunicación asíncrona.

EJERCICIOS PROPUESTOS

1. Realice una página web de selección origen/destino para la compra de billetes de autobús.

Orígenes y destinos posibles:

- ✓ Madrid: Barcelona, Valencia, Sevilla.
- ✓ Barcelona: Madrid, Zaragoza.
- ✓ Valencia: Madrid.

La página web estará conformada por:

- ✓ Una lista desplegable donde se encuentre el listado de orígenes posibles.
- ✓ Una vez seleccionado uno de los orígenes, se llamará a la función JavaScript con la ciudad de origen seleccionada.
- ✓ Desde el servidor se retornará (de manera estructurada) el listado de ciudades destino.
- ✓ Desde JavaScript se cargarán las ciudades en la lista desplegable destino.

Consideraciones:

- ✓ Elegir la tecnología de comunicación y procesamiento de datos: XML o JSON y justificar.
- ✓ El código fuente del lado del cliente debe ser completamente correcto.
- ✓ No es necesario realizar el código fuente del lado del servidor.

TEST DE CONOCIMIENTOS

1 ¿La comunicación asíncrona aparece en el año 2005?
a) Verdadero.
b) Falso.

2 ¿AJAX es un lenguaje de programación?
a) Verdadero.
b) Falso.

3 Las tecnologías incluidas en AJAX son:
a) XHTML, CSS, XML, DOM, Javascript, XSLT XMLHttpRequest.
b) HTML, PHP, Javascript, XMLHttpRequest. F
c) HTMLHttpRequest, XHTML, CSS, DOM, Javascript.
d) AJAX es una tecnología en sí misma. F

4 Un desarrollador busca obtener datos estructurados desde el servidor que va a transformar en diferentes formatos (fichero PDF, HTML, etc.). ¿Cuál sería su recomendación?
a) Utilizar XML como formato de intercambio de datos.
b) Utilizar JSON como formato de intercambio de datos.
c) Ninguna opción es correcta.

5 El objeto `XMLHttpRequest` permite comunicación síncrona:
a) Falso, es una tecnología para realizar solamente comunicación asíncrona.
b) Falso, aunque mediante modificaciones no habituales puede tener un comportamiento síncrono.
c) Verdadero, por defecto la comunicación será síncrona. F
d) Verdadero. F

6 La utilización de la función `eval()` está recomendaba para:
a) Solamente para compilar código JSON, debido a que es el único que acepta. F
b) Solamente documentos XML, debido a que son los únicos que lo acepta. F
c) Es una manera fácil y segura de interpretar código JavaScript.
d) Evaluar código JavaScript de una fuente segura. ✓

7 La utilización de librerías al programar con AJAX:
a) Está completamente prohibido. F
b) No es necesario, aunque en ocasiones puntuales puede ser recomendable. F
c) Es aconsejable, aunque actualmente no existen librerías confiables. F
d) El resto de opciones son incorrectas.

3) XHTML
CSS
DOM
XML, XSLT
XML HttpRequest
JavaScript

8 Almacenamiento de datos en el lado cliente

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer los mecanismos de almacenamiento web del lado del cliente.
- ✓ Comentar la especificación web Storage de la W3C.
- ✓ Repasar y comparar las diferentes tecnologías y sus implantaciones: los objetos de almacenamiento web de HTML 5 e IndexedDB.
- ✓ Profundizar en los conceptos genéricos de las bases de datos del lado del cliente.

En este capítulo presentamos los conceptos básicos del almacenamiento de datos e información del lado del cliente. Estudiaremos las desventajas de las *cookies*, como una de las tecnologías más utilizadas en el pasado y que actualmente sigue manteniendo su vigencia. Así mismo, profundizaremos en los nuevos mecanismos introducidos por HTML 5 y su implantación en los navegadores actuales. Por último, comentaremos las tendencias actuales con las bases de datos del lado del cliente y el almacenamiento orientado a objetos JavaScript.

8.1 ALMACENAMIENTO WEB

Desde 1989 las páginas web han surgido como herramientas para visualizar e intercambiar información. Con el paso del tiempo han evolucionado en aplicaciones web, permitiendo interactividad y personalización (Berners-Lee, 1999).

Uno de los pilares de la personalización se encuentra en el concepto de sesión y en la habilidad de almacenar datos del usuario que utiliza el sitio web en el mismo navegador cliente con el fin de mejorar la experiencia del usuario. Una de las premisas en ese sentido es la siguiente: si la información pertenece al usuario debe estar en la localización del usuario (es decir en el navegador cliente). A continuación veremos las diferentes opciones de almacenamiento de datos en los navegadores clientes. La opción inicial son las *cookies* (Barth, 2011).

8.1.1 LAS COOKIES

El protocolo de transferencia de hipertexto o *Hypertext Transfer Protocol* (HTTP) es el mecanismo utilizado para el intercambio de información en Internet. Pero HTTP es un protocolo sin estado, por tanto, el servidor web administrará cada nueva petición HTTP de manera independiente y sin retener ningún tipo de información, como, por ejemplo, los valores de las variables y campos que se han utilizado en cada solicitud. Cuando un cliente solicita una página web, el servidor entrega la página y, a continuación, olvida todo sobre la petición y el cliente.

Por supuesto, esto disminuye la capacidad de las aplicaciones web por lo que la construcción de aplicaciones que necesiten mantener el estado entre peticiones es una tarea más complicada. Es por ello que, teniendo en cuenta esta necesidad, se implementan mecanismos para lograr la persistencia de los datos, especialmente a partir de las sesiones.

Una sesión es una técnica para vincular datos de un usuario identificado a lo largo de los distintos accesos a una aplicación web. Entre los diferentes usos están el de mejorar la usabilidad y la experiencia del usuario. Un ejemplo muy común es el de mantener al usuario registrado en una página que le ha pedido identificación (*login* y contraseña).

Uno de los mecanismos más utilizados para mantener información en el cliente son las *cookies* (Barth, 2011): una pequeña cantidad de datos almacenada en el cliente en forma de pares clave/valor, con una fecha de expiración y relacionada con un dominio determinado (en realidad contiene más atributos pero de forma genérica estos son los más importantes).

La fecha de expiración es un parámetro opcional que indica el tiempo que se conservará la *cookie*. Si no especificamos la fecha de expiración la *cookie* se eliminará cuando el usuario cierre todas las instancias del navegador. Así mismo, el nombre de dominio total o parcial funcionará como filtro. El navegador enviará la *cookie* al servidor cuyo nombre de dominio sea igual a ese atributo. Por ejemplo, si especificamos el nombre de dominio de la forma ".google.es" el

navegador incorporará la *cookie* a *www.google.es*, *maps.google.es*, *scholar.google.es*, etc. En caso de no indicar este atributo realizamos el filtrado más restrictivo posible y la *cookie* será devuelta solo al servidor que la ha originado.

La información contenida en la *cookie* será enviada junto con cada petición HTTP al servidor. Esto tiene como ventaja evidente el acceso inmediato por parte del servidor a los datos del cliente. Tiene como principal desventaja la disminución del rendimiento y las limitaciones de la cantidad de datos a ser almacenado. Así mismo, las *cookies* son almacenadas en un fichero, usualmente en el espacio destinado al navegador en el disco duro del ordenador en texto plano. Este fichero se leerá al abrir nuevamente el navegador y las *cookies* se gestionarán en memoria hasta que el navegador se cierre. En ese momento se grabarán aquellas *cookies* que no hayan expirado. También existen otras limitaciones, como la cantidad de *cookies* asociadas a un dominio determinado o el tamaño máximo (el cual suele ser de 4096 bytes).

Respecto a la seguridad, las *cookies* usualmente persisten en texto plano por lo que es muy difícil asegurar que los datos no sean corruptos o evitar que información sensible del usuario pueda hacerse pública.

8.1.2 PROBLEMAS CON LAS COOKIES

Aunque es una de las tecnologías más extendidas, las *cookies* tienen una serie de inconvenientes (Hope, 2008). El principal inconveniente tiene que ver con que los usuarios las relacionan con aspectos maliciosos. En realidad, una *cookie* es solo información en texto plano administrable por el mismo usuario y en ningún caso es código fuente interpretable.

Parte de la sencillez de las *cookies* genera inconvenientes:

- Cada navegador tendrá sus propias *cookies*.
- Las *cookies* no diferencian entre usuarios que utilicen el mismo navegador en una misma sesión del sistema operativo. Muchas veces queda almacenada la información de nuestra tarjeta de crédito al realizar una transferencia bancaria.
- Son vulnerables a los *sniffer* (programas que pueden leer el contenido de peticiones y respuestas HTTP) debido a que estas se realizan en texto plano.
- Las *cookies* pueden ser modificadas en el cliente, lo cual podría aprovechar vulnerabilidades del servidor. Si, por ejemplo, una *cookie* contiene información sobre una compra vía web, cambiando el precio de compra el servidor podría permitir al atacante pagar menos. Esto es fácil de resolver, indicando en la *cookie* solo identificadores de sesiones en el servidor y no directamente información sensible.

8.1.3 LAS COOKIES DE FLASH

Uno de los avances más significativos en las páginas web ha sido la inclusión de Flash. Esta tecnología es utilizada para crear y manipular gráficos vectoriales logrando animaciones y sitios web interactivos (Keefe, 2008). Podemos encontrar aplicaciones web realizadas en un gran porcentaje con Flash e información sobre sus ventajas respecto de HTML. Aun así, actualmente en sitios que no son comerciales se utiliza la tecnología Flash solo en publicidad y en visores de vídeos.

La tecnología Flash también utiliza almacenamiento del lado del cliente, con los denominados *Local Shared Object* (Objeto Local Compartido) o como comúnmente se los llama: *cookie flash*.

Son utilizados con el mismo propósito que las *cookies*: para almacenar en el cliente pequeñas cantidades de información que serán accedidas en sesiones posteriores. La diferencia inicial se encuentra en su gestión, ya que los navegadores por lo general no implementan un sistema para su modificación y borrado. Otra diferencia es su peso (hasta 100 kilobytes, en lugar de los 4 kilobytes de las *cookies* normales).

Adobe ofrece desde su página un programa en Flash que nos permite gestionar las *cookies flash*, visualizarlas y borrarlas. Así mismo, en el panel de configuración global de almacenamiento podemos controlar cuánto espacio de disco utilizan los sitios web para almacenar información o no permitir el almacenamiento de información en el equipo.

8.1.4 LA ESPECIFICACIÓN WEB STORAGE DE LA W3C

El almacenamiento web ha cobrado más fuerza con la nueva especificación de HTML 5. Ha sido estandarizado por el *World Wide Web Consortium* (W3C) y actualmente se encuentra en una especificación por separado siendo implementado por Internet Explorer 8 y Firefox entre otros navegadores.

HTML 5 incluye dos nuevos objetos para el almacenamiento de datos en el cliente: los `sessionStorage` y los `localStorage` (W3C, 2011a). Más adelante detallaremos el uso de estos objetos. En la Tabla 8.1 podemos ver una lista de navegadores que soportan esta tecnología.

Tabla 8.1 Listado de navegadores que implementan Web Storage

Caraterística	Internet Explorer	Firefox	Chrome	Opera	Safari
`localStorage`	8	3.5	4	10.50	4
`sessionStorage`	8	2	5	10.50	4

Esta especificación introduce dos mecanismos relacionados para obtener la persistencia de datos de manera estructurada del lado del cliente, similares al mecanismo de las *cookies*. Una de las principales diferencias está en que el contenido de las *cookies* es enviada al servidor en cada petición. En HTML 5 no, de hecho la información solo podrá ser accedida desde el lado del cliente por lo que es posible almacenar gran cantidad de información sin afectar el rendimiento de la aplicación web. Por último indicar que se utiliza JavaScript para realizar el almacenamiento y acceso a los datos.

Algunos escenarios de implementación y uso son los siguientes:

- **Almacenamiento de datos**. Los datos son almacenados en el cliente y pueden ser pasados al servidor por intervalos, en lugar de en tiempo real. Aclaración: utilizando `localStorage` los datos también estarán disponibles entre peticiones y sesiones del navegador.
- **Utilización fuera de línea**. Como consecuencia de que la relación entre los datos almacenados y el navegador no se pierden entre sesiones (para el `localStorage`).

- **Mejora de rendimiento**. Podemos almacenar datos estáticos (por ejemplo imágenes en codificación Base64) que no serán nuevamente obtenidos mediante peticiones al servidor.
- En el caso del `sessionStorage` no existe relación entre lo almacenado en diferentes pestañas o ventanas de un mismo navegador.
- Con el objeto `sessionStorage` existe la seguridad de que los datos serán borrados una vez termine la sesión de la ventana que lo ha utilizado.

SessionStorage

Actualmente podemos encontrarnos con un problema para realizar múltiples transacciones en diferentes ventanas o pestañas de un navegador si lo hacemos al mismo tiempo, en caso de que la aplicación utilice *cookies* para mantener el estado. Las diferentes instancias del navegador (ya sea en diferentes pestañas de una misma ventana o en diferentes ventanas) obtendrán información de una misma *cookie* debido a que todas las ventanas están asociadas a una misma sesión (ver Figura 8.1).

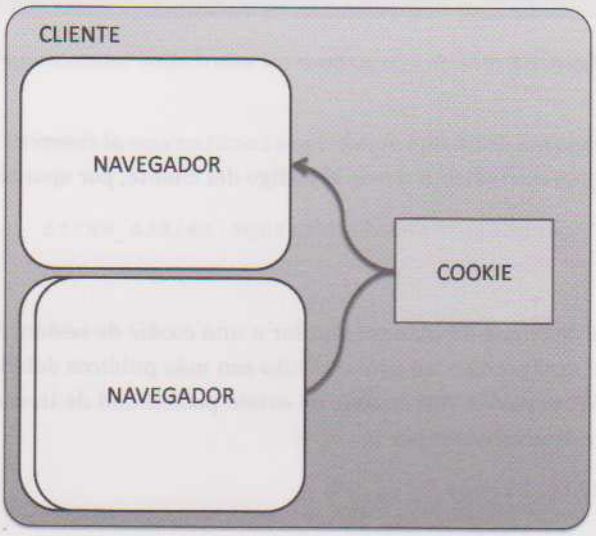


Figura 8.1. *Relación entre ventanas que usan cookies*

La especificación actual de *Web Storage* incluye un nuevo objeto llamado `sessionStorage`. Las aplicaciones web pueden agregar información a este atributo el cual estará accesible durante toda la sesión. El objeto `sessionStorage` se instancia por sesión y ventana, por lo que dos pestañas del navegador abiertas al mismo tiempo y para un mismo sitio web pueden tener información distinta (ver Figura 8.2). Al cerrar la sesión se pierde la información.

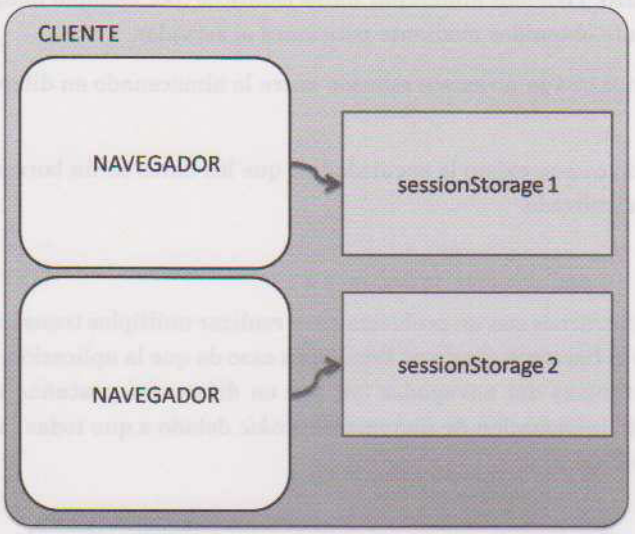


Figura 8.2. *Relación entre ventanas que usan el objeto "sessionStorage"*

A continuación veremos un ejemplo de uso del objeto `sessionStorage` al reservar un vuelo y poder elegir maletas extras. La utilización de este objeto se realizará desde el código del cliente, por ejemplo:

```
<input type="checkbox" onchange="sessionStorage.maleta_extra
  = checked ? 'true' : ''">
```

Al parecer este tipo de objeto funciona de manera similar a una *cookie* de sesión. Aun así, existen diferencias. En cuanto a la seguridad, los datos almacenados en este atributo son más públicos debido a que no existe la posibilidad de esconder información entre los usuarios. Así mismo, no existe posibilidad de iterar entre los ítems y es necesario pedir el valor del par clave/valor directamente por la clave.

```
if (sessionStorage.maleta_extra) { ... }
```

En cuanto al tiempo de expiración, no es posible indicarlo. Esto podría parecer una desventaja en relación con las *cookies* de sesión. En otro sentido, una de las mayores ventajas teóricas de este objeto es su persistencia entre caídas del cliente.

A continuación indicamos los métodos del objeto `sessionStorage`:

- **setItem()**. Para agregar un nuevo par clave/valor, por ejemplo una maleta adicional al comprar un billete de avión utilizamos el método `setItem()` con la clave "`maleta`" y el valor "`1`":

  ```
  sessionStorage.setItem("maleta","1");
  ```

- **getItem()**. Para obtener el dato almacenado utilizamos el método `getItem()` con el nombre de la clave:

  ```
  var item = sessionStorage.getItem("maleta");
  ```

- **removeItem()**. Para eliminar el par clave/valor existe el método `removeItem()` que deberá ser llamado con la clave o con la posición del elemento a eliminar:

```
var item = sessionStorage.removeItem("maleta");
var item = sessionStorage.removeItem(1);
```

- **clear()**. Este método borra todos los elementos de la lista `sessionStorage`.

```
sessionStorage.clear();
```

- Así mismo, podemos utilizar otros atributos, por ejemplo el atributo `length` para conocer la cantidad de elementos clave/valor almacenados:

```
var cantidad_elementos = sessionStorage.length;
```

Como ejemplo de utilización podemos ver un código JavaScript para mantener la información de un campo de texto de búsqueda, después de que el navegador sea refrescado accidentalmente.

El objetivo del código que veremos a continuación es mantener el contenido del elemento "`busqueda`" en el objeto `sessionStorage`. Para utilizamos la función `setInterval()` de JavaScript, la cual tiene como primer parámetro el nombre de una función que se ejecutará cada cierto intervalo. Como segundo parámetro pasamos el intervalo de ejecución en milisegundos. Por ejemplo, a continuación se ejecutará cada segundo la función "`refrescar`".

```
setInterval(refrescar,1000);
```

Volviendo con el ejemplo, lo primero que necesitamos es obtener el elemento que queremos persistir:

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

A continuación detallamos la función "`refrescar`", la cual guarda el valor de la variable "`búsqueda`". Recordando que esta función se ejecutará cada segundo, en todo momento la tendremos en el objeto `sessionStorage` el elemento de nombre "`autoguardado`" conteniendo el valor de la variable "`búsqueda`".

```
refrescar(){
  sessionStorage.setItem("autoguardado", busqueda.value);
}
```

Recapitulando el ejemplo, la primera vez que se interpreta el código de la página web se carga la variable "`búsqueda`" y cada segundo se refresca el elemento "`autoguardado`" con el valor de la variable.

En caso de perder el contenido de la variable "`busqueda`" y si existe el elemento "`autoguardado`" podremos recuperar su valor:

```
if ( sessionStorage.getItem("autosave")){
  search.value = sessionStorage.getItem("guardado");
}
```

El código completo es el siguiente:

```
var busqueda = document.getElementById("búsqueda");

if ( sessionStorage.getItem("autoguardado")) {
  busqueda.value = sessionStorage.getItem("guardado");
}

setInterval(refrescar,1000);

refrescar(){
  sessionStorage.setItem("autoguardado", busqueda.value);
}
```

Para terminar, este código fuente lo podemos asociar a un evento del campo de texto "`busqueda`", para que se ejecute cada vez que el contenido del texto cambie.

LocalStorage

El segundo mecanismo de almacenamiento está diseñado para los datos que se extienden a lo largo de múltiples ventanas y múltiples sesiones. Está orientado a las aplicaciones que necesitan mantener gran cantidad de información del usuario (en el orden de los megabytes), principalmente por motivos de rendimiento.

Para ello, la W3C ha definido el objeto `localStorage`. De esta manera podemos acceder al área de almacenamiento local del dominio. Puede ser accedido cada vez que se visita el dominio (los subdominios no son válidos) y todas las sesiones abiertas sobre la misma web ven la misma información (ver Figura 8.3).

Figura 8.3. *Persistencia del objeto localStorage*

En la especificación oficial se indica, además, que cualquier tipo de cambio en el almacén debe disparar un evento de tipo `StorageEvent`, de forma que cualquier ventana con acceso al almacén pueda responder al mismo. Esto ocurrirá ante cambios como: agregado o modificación de un elemento, borrado de un elemento o de todos ellos.

Las cuatro propiedades del evento las podemos ver en la Tabla 8.2.

Tabla 8.2 Propiedades del evento storage

Propiedades	Internet Explorer
`url`	El dominio asociado con el objeto que ha cambiado.
`storageArea`	Representa el objeto `localStorage o sessionStorage afectado.`
`key`	La clave del par clave/valor que ha sido agregado, modificado o borrado.
`newValue`	El nuevo valor asociado con la clave. Será `null` si trata de una eliminación.
`oldValue`	El antiguo valor.

A continuación explicamos el ejemplo clásico de contar la cantidad de veces que se ha visitado una página. Debido a que con el objeto `localStorage` no se pierde entre sesiones de un mismo dominio podemos realizar un contador de la siguiente manera:

```
<p>
  Ud. ha visto esta página:
  <span id="cuenta"> muchas </span>
  veces.
</p>

<script>
  if(!localStorage.cuenta)
    localStorage.cuenta = 0;

  localStorage.cuenta = parseInt(localStorage.cuenta) + 1;
  document.getElementById("cuenta").textContent =
  localStorage.cuenta;
</script>
```

En este caso inicializamos el elemento "`cuenta`" con el valor 0. Como veremos en el siguiente código, algunos navegadores permiten realizar el acceso directo a las claves del objeto como si se tratasen de una variable. Es decir, en lugar de incluir la sentencia:

```
localStorage.setItem("cuenta","0");
```

Realizamos lo siguiente:

```
localStorage.cuenta = 0;
```

Otro aspecto importante aquí es que JavaScript realiza la conversión automática de tipo de dato numérico a cadena de caracteres. Por ello, al recuperar el valor almacenado es necesario transformarlo en tipo numérico para que sea tratado como un contador y sumarle la nueva entrada:

```
localStorage.cuenta = parseInt(localStorage.cuenta) + 1;
```

La persistencia de los datos

Una de las características importantes de este tipo de objetos es el tener una capacidad mayor de almacenamiento. Dependiendo el navegador, existirán diferentes estrategias de implantación en cuanto la forma de escritura de los datos. Firefox, por ejemplo, realiza el almacenamiento de manera síncrona, es decir que persiste inmediatamente los datos que se almacenan mediante código. En cambio en Internet Explorer los datos se escriben de manera asíncrona y, por tanto, puede existir una diferencia (mayor cuanto mayor sean los datos) entre el tiempo de asignación y el tiempo de escritura. La principal ventaja de ello es la rapidez en la ejecución del código, aunque existe mayor peligro de perder los datos.

En Internet Explorer 8 podemos forzar la persistencia mediante los métodos `begin()` y `commit()` (MSDN, 2009). Las asignaciones de datos que se encuentren encerrados entre la llamada de ambos métodos se realizarán de manera síncrona. Por ejemplo:

```
sessionStorage.begin();

sessionStorage.setItem("figura1","R0lGODlhNQAkAKIAAHJycvwHB
  wAAqwCjC+3PDv///wAAAAAAACwAAAAANQAkAA");

sessionStorage.commit();
```

ACTIVIDADES 8.1

- Investigue el uso del objeto `globaStorage` y su relación con los objetos `localStorage` y `sessionStorage`.

8.2 BASES DE DATOS SQL (STANDARD QUERY LANGUAGE) EN ENTORNO CLIENTE

En HTML 5 se especifican las características de un sistema de base de datos de objeto en el lado del cliente. A continuación explicaremos los conceptos centrales de esta nueva funcionalidad.

Por un lado es necesario aclarar lo que se considera una base de datos. En ese sentido lo podríamos definir como un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso. Esta misma definición puede dársele a las *cookies* o los objetos `sessionStorage` o `localStorage` (comúnmente llamados

DOM Storage). La diferencia principal entre estos últimos y una base de datos es la estructuración de los datos, lo que permite mantener una mayor eficiencia de los mismos y la aplicación de búsquedas directas.

Por otro lado, es necesario explicar la ventaja de tener una base de datos local en lugar de utilizar una base de datos en el servidor. Las bases de datos locales permiten almacenar datos en el cliente teniendo acceso a esta información sin necesidad de estar en línea.

Dentro de las bases de datos más utilizadas encontramos dos: las bases de datos relacionales y las bases de datos orientadas a objeto. La idea fundamental en la base de datos relacional es el uso de las relaciones entre conjunto de tuplas (comúnmente representadas por tablas).

Aunque las bases de datos relacionales son actualmente las más utilizadas, tienen una desventaja respecto de los paradigmas de programación actuales. Las aplicaciones se encuentran mayormente desarrolladas a partir de la orientación a objetos, donde se programan objetos compuestos por datos, comportamiento y relacionados con otros objetos. Sin embargo, las bases de datos actuales no están preparadas para almacenar objetos sino para almacenar tuplas (también llamados registros). En cambio en las bases de datos orientadas a objetos se realiza la persistencia de los objetos utilizados en la programación de manera transparente, sin necesidad de transformar los datos (ver Figura 8.4).

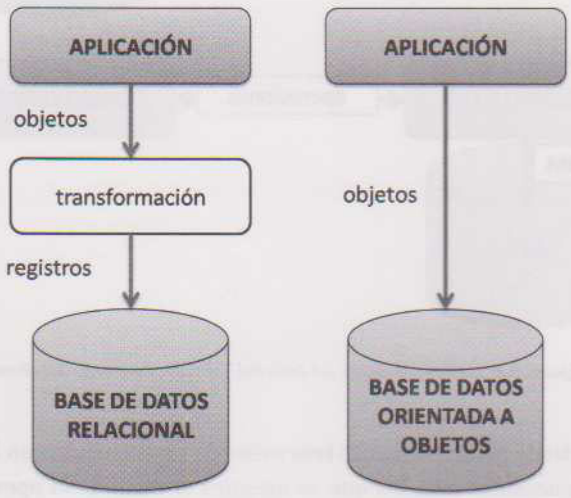


Figura 8.4. *Persistencia en bases de datos relacionales y orientadas a objetos*

Actualmente en las aplicaciones web existen principalmente dos técnicas de bases de datos locales:

- **WebSQL (basado en SQLite)**. Actualmente no tiene más soporte pero que ha sido y es utilizado por los navegadores.
- **Indexed Database API**. Impulsado por la W3C y Oracle es la propuesta de estándar actualmente vigente. Se encuentra parcialmente implementado en las últimas versiones de los navegadores.

8.2.1 WEBSQL

Esta propuesta ya no se encuentra en mantenimiento activo por el grupo de trabajo de aplicaciones web de la W3C desde noviembre de 2010 (W3C, 2010). Está basada en SQLite, un gestor de base de datos relacionales construido en el lenguaje C.

Entre las ventajas relativas que tiene este tipo de motor de base de datos está que se enlaza directamente con la lógica de negocio de la aplicación (en este caso, por ejemplo con el código del lado del cliente). Por lo tanto, la base de datos se encuentra en el cliente y no en el servidor (ver Figura 8.5).

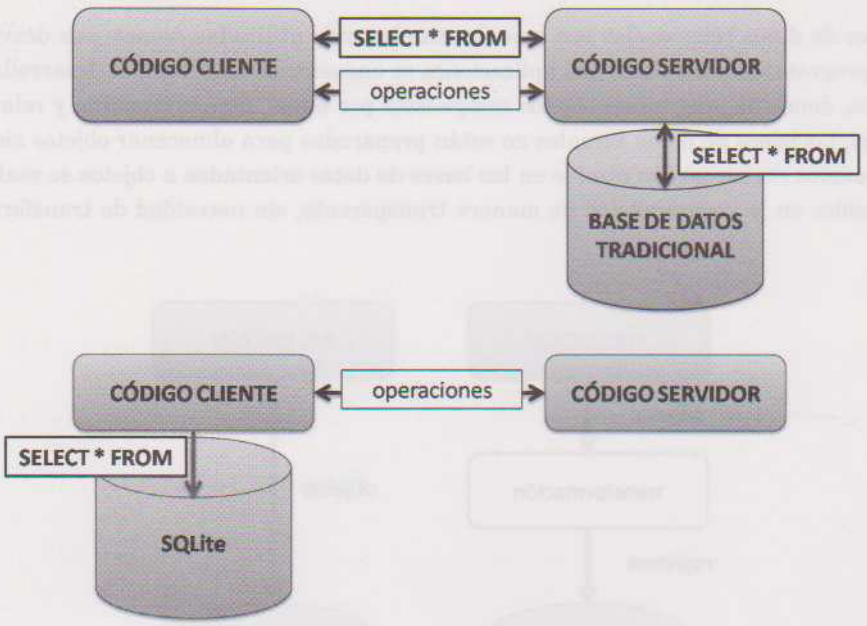


Figura 8.5. *La base de datos del lado del servidor y del lado del cliente*

La API de esta propuesta contiene principalmente tres métodos que consisten en la apertura y creación de la base de datos, la creación de las transacciones (con las que se asegura el éxito de la operación con la base de datos) y la ejecución o aplicación de consultas SQL.

Actualmente esta propuesta no es estable y los principales navegadores del mercado (Microsoft y Mozilla) no lo soportan, apostando por el *Indexed Database API*.

8.2.2 INDEXED DATABASE API

Esta propuesta, también denominada IndexedDB es una interfaz de aplicación de programas (API) para el almacenamiento de gran cantidad de datos de manera estructurada utilizado en HTML 5 y es el sustituto natural de WebSQL cuya especificación ha sido abonada en el año 2010 por la W3C (W3C, 2011b).

Definiciones

Debido a que IndexedDB almacena directamente objetos es posible persistir objetos JavaScript. Para ello hace uso de un mecanismo denominado *Object Store* para lograr la persistencia (ver Tabla 8.3 para una comparación entre conceptos relacionales y esta API).

Tabla 8.3 Comparación entre conceptos de bases de datos relacionales y la tecnología Indexed Database API

Característica	BD relacional	Indexed Database API
Tabla	La tabla contiene filas y columnas.	Los *Object Store* o contenedores que contienen objetos Javascripts y claves.
Consultas	SQL.	Cursores y rangos de clave. Son necesarios los índices para realizar las búsquedas.
Transacciones	De lectura/escritura y a nivel de tablas o registros.	`Read_Version_Change`, `Read_Write`, `Read_Only`.
Estructura de datos	Cumple con las reglas de forma normal.	Estructura no normalizada.

En IndexedDB no se trabaja con el concepto de consultas SQL, en cambio se trabaja con índices y cursores que recorren los objetos. El *Object Store*, que a partir de ahora llamaremos "contenedor" almacena pares clave/valor. Este concepto de pares clave/valor es utilizado también en los objetos *DOM Storage* vistos anteriormente. La diferencia se encuentra en que, en este caso, los valores por lo general serán objetos con una estructura compleja. Así mismo, la clave podrá ser una propiedad de este objeto. Por último, la búsqueda la podemos realizar a partir de propiedades del mismo objeto.

Clave: un valor a partir del cual los objetos almacenados son organizados. Por lo tanto, en cada contenedor no podemos repetir la clave. A continuación indicamos las características de la clave (llamada *key* en la especificación original de IndexedDB):

- **Un generador de claves**. Con el cual se producen nuevas claves de forma artificial y ordenada.
- **In-line Key**. La clave es almacenada como parte del valor.
- **out-of-line key**. La clave es almacenada fuera del valor almacenado.
- **key path**. Define el lugar desde donde el navegador extrae la clave en el contenedor.

Otro de los conceptos a tener en cuenta son los índices. Un índice es creado en un contenedor y permite buscar objetos dentro del mismo en base a los valores que contiene y no solamente en base a la clave del objeto.

La especificación de la API es extensa y actualmente todavía se encuentra como borrador de trabajo. Los conceptos del IndexedDB sobre los que trabajaremos a continuación serán los siguientes:

- Crear o abrir una base de datos e indicar su versión.
- Eliminar una base de datos.
- Iniciar una transacción.
- Realizar una petición de operación en base de datos.

Crear, abrir y eliminar una base de datos

Para la gestión de la base de datos es indispensable el método `open()`. Este método funciona de dos maneras: crea la base de datos si esta no existe o la abre en caso de que exista. Si todo funciona correctamente el evento *success* es disparado y la función indicada en la propiedad `onsuccess` es aplicada.

Como vemos en el código fuente, llamamos a la API de IndexedDB y aplicamos el método `open()` con el nombre de la base de datos. Junto con ello, asignamos a la propiedad `onsuccess` a la función "`respuesta_exito`", la cual tiene un parámetro "`evento`". El parámetro es un objeto que incluye información de la operación actualmente realizada. De hecho, la propiedad `evento.result` contiene el resultado de la operación, es decir la misma base de datos.

En caso existir un error se dispara la función relacionada con la propiedad `onerror` de la petición con el mismo evento de error como argumento. La API IndexedDB ha buscado minimizar el manejo de errores. Aun así, existen situaciones que producen errores, como por ejemplo cuando el usuario no permite al sitio web la creación de la base de datos.

```
var peticion = window.indexedDB.open("nombre");

peticion.onsuccess = respuesta_exito;
peticion.onerror  = respuesta_error;

respuesta_exito(evento){
  var bbdd = evento.result;

  alert("Base de Datos abierta",bbdd);
};

respuesta_error(evento){
  alert(evento);
};
```

Uno de los atributos más importantes de la base de datos es la versión, que sirve como identificador único. La aplicación web podrá acceder en todo momento a una única versión de la base de datos. Este atributo es una cadena de caracteres inicializada al valor vacío una vez creada la base de datos.

A continuación detallamos cómo establecer la versión en la base de datos creada:

```
if(bbdd.version != "VERSION_1")
 var peticion = bbdd.setVersion("VERSION_1");

peticion.onsuccess = respuesta_exito;
peticion.onerror  = respuesta_error;
```

Por último, la eliminación de una base de datos es similar a la creación. Una característica muy importante es que no se producirá ningún error en caso de intentar borrar una base de datos inexistente.

```
var peticion = window.indexedDB.deleteDatabase("nombre");

peticion.onsuccess = respuesta_exito;
peticion.onerror  = respuesta_error;
```

Creación de contenedores e índices

En este ejemplo veremos cómo crear una estructura en la base de datos instanciada. Como primer paso, tenemos un objeto JavaScript estructurado en un *array* de dos elementos que representan contactos, con teléfono y dirección de correo electrónico.

Establecemos la versión de la base de datos, lo cual es similar a intentar abrir una conexión a la misma. En caso de que la versión de la base de datos no exista el evento será un error. En otro caso la petición finalizará con un evento exitoso. En el caso de que el evento sea exitoso se ejecutará la función incluida en la propiedad `onsuccess` de la petición.

En este caso la función está escrita de manera anónima y asignada directamente a la propiedad `onsuccess`. Por lo tanto, las operaciones asociadas con la alteración de la estructura de una base de datos la realizaremos en `setVersion()`, dentro de la ejecución de la función que es llamada en la propiedad `onsuccess`. En este caso, se realizan tres acciones.

La primera es la creación de un contenedor de objetos (similar a las tablas en el modelo de base de datos relacionales, ver Tabla 8.3). El objeto se denomina `contactos` y su `keyPath` será la variable `tel`. Es decir, que los objetos incluidos en el contenedor tendrán un valor de nombre `tel` que también será la clave.

Lo segundo es la creación de un índice mediante la función `createIndex()`. Se incluye el nombre del índice y el valor asociado al índice (en este caso será la variable `correo`). Por último, se incluye un *array* de parámetros opcionales que en este caso solo incluye la opción `unique`. Esto significa que podremos listar los objetos del *Object Store* por el valor de una variable denominada `correo` y que no pueden existir dos objetos con el mismo valor en la variable.

Lo tercero es la carga de objetos en el contenedor. En este caso simplemente se utiliza el método `add()` del contenedor y le pasamos como parámetro cada objeto JavaScript. Como vemos al inicio del ejemplo, los objetos son pares nombre/valor, siendo dos de los nombres `tel` y `correo`; el primero será la clave del contenedor y se construirá un índice para el segundo con la restricción adicional de que dos contactos no pueden tener el mismo correo.

```
contacto = [
  {tel: "1234", correo:"juan@mail.com", movil:"666123"},
  {tel: "2345", correo:"pedro@mail.com", movil:"664392"}
];

var peticion = bbdd.setVersion("VERSION_1");

peticion.onsuccess=function(event){
  var contenedor = bbdd.createObjectStore("contactos",
    {keyPath:"tel"});

  contenedor.createIndex("correo", "correo", {unique:true});

  for (n in contacto){
    contenedor.add(contacto[n]);
  }
};
```

Operaciones con la base de datos

Las operaciones con las que se agregarán, modificarán y eliminarán datos debemos realizarlas en una transacción. En base de datos se indica como transacción a un conjunto de operaciones que se aplican formando una unidad de tal manera que dentro de la transacción se mantiene la integridad de los datos, haciendo que estas operaciones no puedan finalizar en un estado intermedio.

En caso de que se cancele la transacción en alguna operación intermedia, se empiezan a deshacer las modificaciones aplicadas hasta dejar la base de datos en su estado inicial, como si las operaciones dentro de la transacción nunca se hubiesen realizado. La transacción en IndexedDB está formada por tres parámetros:

El primer parámetro son los contenedores implicados en la transacción. En caso de no indicar nada se asume que todos los contenedores están en la transacción.

El segundo parámetro es el modo de la transacción que indica el tipo de operación a ser realizada y si dos transacciones pueden aplicarse de manera concurrente. La transacción puede ser abierta de tres modos diferentes:

- **Read_Only**. Solo lectura y sin modificación de datos (es la opción por defecto).
- **Read_Write**. Permite la lectura y escritura de datos en contenedores existentes.
- **Version_Change**. Permite cualquier tipo de operación, incluyendo la creación y borrado de contenedores e índices. Este tipo de transacción no se ejecutará de manera concurrente junto con ninguna otra transacción.

El tercer parámetro es la instancia de base de datos de la transacción. A continuación podemos ver un ejemplo de transacción donde la lista de contenedores asociados es solo uno. La transacción en este caso será de lectura/escritura; para indicarlo utilizamos el objeto enumerado `IDBTransaction` definido en la API de la especificación IndexedDB.

La transacción puede finalizar de tres maneras posibles: completada sin error, con error o abortada. Es necesario manejar estos tres eventos mediante diferentes funciones, como vemos en el ejemplo:

```
var transaccion = bbdd.transaction(["contactos"],
  IDBTransaction.READ_WRITE);

transaccion.oncomplete = function(evento){…};
transaccion.onabort = function(evento){…};
transaccion.onerror = function(evento){…};
```

Una vez que tenemos la transacción abierta podemos operar con los contenedores y objetos. Para agregar un nuevo elemento utilizamos la función `add()`. El resultado de agregar un nuevo objeto (`evento.result`) es la clave de dicho objeto añadido.

Por lo tanto, no es posible agregar dos objetos con la misma clave mediante la operación `add()`. En lugar de ello, para modificar un elemento existente utilizamos la función `put()`.

```
var contenedor = transaccion.objectStore("contactos");

for (var i in contacto) {
  var peticion = contenedor.add(contacto[i]);
```

```
  peticion.onsuccess = function(evento) {…};
}

peticion = contenedor.put({tel:"2345",
  correo:"pedro@mail.com",movil:"661111"});
```

Por último, para eliminar un elemento utilizamos la función `delete()`. En el siguiente ejemplo llamamos al contenedor directamente a partir de la transacción y borramos el objeto cuya clave es "1234".

```
var peticion = bbdd.transaction(["contactos"],
  IDBTransaction.READ_WRITE).objectStore("contactos")
  .delete("1234");

peticion.onsuccess = function(evento) {…};
```

La obtención de datos

La API permite obtener tanto objetos simples como conjunto de objetos. Para el caso de los objetos simples se utiliza el método `get()` con la clave del objeto a ser recuperado. El objeto devuelto se encontrará en la propiedad `result` del evento.

```
var transaccion = bbdd.transaccion(["contactos"]);
var contenedor = transaccion.objectStore("contactos");
var peticion = contenedor.get("2345");

peticion.onerror = function(evento){…};

peticion.onsuccess = function(evento){
  if(evento.result.tel=="2345")
     alert("Objeto recuperado correctamente!!!");
};
```

En el caso de que busquemos obtener más de un objeto es necesario utilizar un cursor para recorrerlos, limitando o no el rango de objetos. En el ejemplo siguiente abrimos y recorremos un cursor a partir de un contenedor de objetos del tipo `contacto`. El cursor es el resultado del evento en caso de éxito (es decir, se encuentra en la propiedad `result`. La apertura del cursor se realiza mediante el método `openCursor()` pudiendo tener parámetros opcionales que limiten el rango.

En este caso el cursor no tiene ningún parámetro adicional por lo que recorrerá los contactos hasta finalizar. En la variable cursor se encuentra el objeto actual obtenido, de tal manera que con el método `continue()` cargamos la variable con el siguiente objeto (ver Figura 8.6).

```
var contenedor = bbdd.transaccion(["contactos"])
  .objectStore("contactos");

contenedor.openCursor().onsuccess = function(evento){
  var cursor = evento.result;
  if (cursor) {
    alert("CONTACTO " + cursor.tel);
    cursor.continue();
  }else {
    alert("FINALIZADO");
  }
};
```

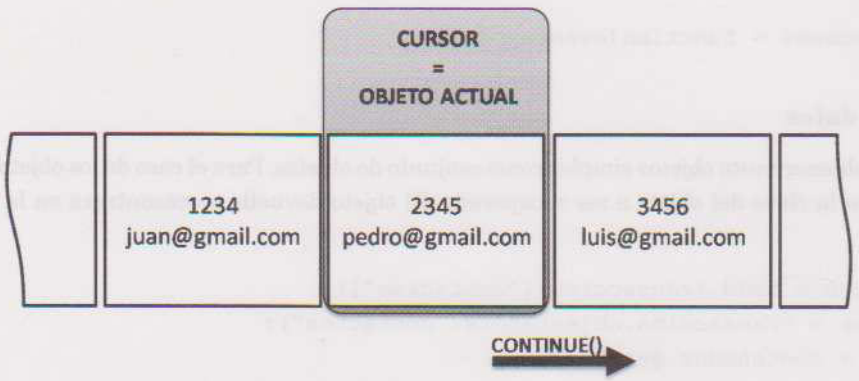


Figura 8.6. *Esquema de un cursor en IndexedDB*

Así mismo, existe la posibilidad de limitar el resultado obtenido por el cursor a un rango de claves (llamado *key range* en la API). Este rango de claves es flexible, pudiendo contener un único valor, o un único valor extremo (inferior o superior) o ambos valores extremos. A su vez las claves extremas pueden o no estar contenidas dentro del rango.

A continuación veremos algunos ejemplos. Suponiendo que tengamos en un contenedor llamado `contactos` un total de seis objetos cuya clave sea el teléfono de la siguiente manera: 1234, 2345, 3456, 4567, 5678 y 6789, el resultado de limitar el rango de acuerdo con lo indicado por la API de IndexedDB es:

- Solo un objeto cuya clave es 1234. Se utiliza la función `only()`:

```
rango = IDBKeyRange.only("1234");
```

- Cinco objetos, incluyendo a 2345: 2345, 3456, 4567, 5678 y 6789. Utilizamos la función `lowerBound()` con un parámetro indicando el límite inferior:

```
rango = IDBKeyRange.lowerBound("2345");
```

- Cuatro objetos, sin incluir a 2345: 3456, 4567, 5678 y 6789. Utilizamos la función `lowerBound()` con un parámetro adicional booleano indicando que no se excluye el límite (dado por el primer parámetro de la función), entre los objetos del resultado.

```
rango = IDBKeyRange.lowerBound("2345", true);
```

- Tres objetos, incluyendo a 3456: 1234, 2345 y 3456. Utilizamos la función `upperBound()`:

```
rango = IDBKeyRange.upperBound("3456");
```

- Tres objetos entre 2345 y 4567, sin incluir el límite inferior: 2345, 3456 y 4567. Utilizamos la función `bound()` que contiene como parámetros el límite inferior, el límite superior y el indicador de la exclusión de los valores límites.

```
rango = IDBKeyRange.bound("1234","4567",true,false);
```

La variable `rango` es obtenida del elemento `IDBKeyRange` que forma parte de la API de IndexedDB y que debe ser implantado por el navegador. Esta variable se emplea como parámetro en el método `openCursor()` para la limitación de los objetos obtenidos.

¿SABÍAS QUE...?

Existen librerías que implementan una interfaz genérica para poder trabajar con IndexedDB o WebSQL de manera transparente.

ACTIVIDADES 8.2

- Investigue si es posible crear bases de datos en el cliente cuando realiza una navegación privada.
- Compare las diferentes implementaciones actuales de IndexedDB de los principales navegadores (Firefox, Internet Explorer y Chrome). ¿En qué se diferencian y que similitudes tienen? ¿Es posible construir aplicaciones que hagan uso de IndexedDB de forma sencilla o serán necesarias librerías genéricas?

8.3 APLICACIONES EN CACHÉ

La caché de las aplicaciones web es el proceso de almacenar datos generados dinámicamente para que puedan ser utilizados nuevamente sin necesidad de realizar peticiones al servidor. Se utiliza principalmente para mejorar el rendimiento de las aplicaciones web y disminuir el tiempo de carga (Fielding et. al, 2008).

8.3.1 VENTAJAS Y DESVENTAJAS

Al igual que otras técnicas, esta también contiene un conjunto de ventajas y desventajas que la hacen beneficiosa solo en determinados ámbitos. Entre las ventajas más características encontramos principalmente dos:

- ✓ El rendimiento en las aplicaciones web es un aspecto fundamental. En ese sentido cualquier mejora provoca un cambio significativo en la experiencia del usuario.
- ✓ Es una tecnología ampliamente extendida y, por tanto, los navegadores web implementan componentes de caché utilizables por las aplicaciones.

Así mismo, entre las desventajas más reconocidas están las siguientes:

- ✓ En algunos escenarios (por ejemplo en aplicaciones ya construidas) puede ser complejo agregar aspectos de caché.
- ✓ El comportamiento de la aplicación puede verse afectado, especialmente si realizamos una mala programación.

8.3.2 ML 5

HTML 5 soporta actualmente la caché de los recursos de las aplicaciones web (W3C, 2008). El objetivo de estas nuevas características, junto con la API de IndexedDB es la posibilidad de obtener aplicaciones web y sitios web que funcionen correctamente cuando no tienen conexión con el servidor web (modo fuera de línea).

La funcionalidad consiste en obtener toda la información del sitio web antes de que sea explícitamente pedida por el usuario de tal manera que una vez almacenadas en la caché de aplicación puedan ser utilizadas en modo fuera de línea.

Como diferencia principal entre la caché habitual de una página web y la caché de aplicación es que en el segundo caso podremos acceder también a páginas que no hayan sido visitadas anteriormente.

En ese sentido el primer paso para realizar la caché de aplicación es la elaboración de un fichero llamado *MANIFEST* (o manifiesto en español). En este fichero listaremos el conjunto de recursos con los que se trabajaran en modo fuera de línea.

El manifiesto

Existen en total tres características que debemos tener en cuenta para utilizar la caché de aplicación de HTML 5 (en el caso de que esté implementado en el navegador). El primero de ellos es la realización del fichero manifiesto, el cuál será un fichero en texto plano con extensión ".appcache" y que contendrá los nombres de los ficheros y recursos en general que serán almacenados por el navegador para ser utilizados en modo fuera de línea.

El manifiesto comienza siempre con la cadena: "CACHE MANIFEST" y contendrá por lo menos tres partes, cada una con las URL absolutas o relativas al mismo manifiesto de los ficheros del sitio web. Las partes son las siguientes:

- **CACHE**. Las URL que se encuentran en esta sección serán mantenidas en la caché de la aplicación por el navegador para ser vistas en modo fuera de línea. Para cumplir con la sintaxis necesaria debe existir solo una URL por línea y debe apuntar a un único recurso (no se admiten caracteres comodín o anclas de página). De esta manera el navegador recibirá el manifiesto, lo leerá y mantendrá en la caché de aplicación todas las URL de esta sección.
- **NETWORK**. Las URL de esta sección no serán cargadas en la caché de aplicación. Es común utilizar el carácter comodín asterisco "*" para indicar que por defecto ninguna página será mantenida en la caché de aplicación (a no ser que se encuentre referenciada en la sección CACHE).
- **FALLBACK**. Indica el contenido que será mostrado en caso de que un recurso no sea encontrado, por ejemplo cuando la aplicación se encuentra fuera de línea y es necesario cargar un recurso que se encuentra en la sección NETWORK.

A continuación vamos a mostrar un ejemplo de manifiesto con las tres secciones. El nombre del fichero es "principal.appcache" y de acuerdo con lo descrito el fichero index.html estará en la caché de aplicación, junto con un fichero CSS y JavaScript. Así mismo, los ficheros "pago.html" y "ultimasnoticias.html" nunca estarán en la caché de aplicación. Por último, en caso de que mostremos las últimas noticias en el fichero "index.html", en lugar de ello mostraremos el fichero "offline.html".

```
CACHE MANIFEST

CACHE:
 index.html
 pagina.css
 pagina.js

NETWORK:
 pago.html
 ultimasnoticias.html

FALLBACK:
 offline.html
```

Referenciar el manifiesto

Una vez creado el manifiesto debemos referenciarlo dentro de la etiqueta HTML de las páginas HTML que estarán en la caché de aplicación. Por ejemplo, si analizamos el contenido de la página principal, esta hace referencia a otros dos recursos: la hoja de estilos CSS y el código de JavaScript. Los tres recursos están en el manifiesto y serán mantenidos en la caché.

```
<html
  <head>
    <title>TITULO</title>
    <script src="pagina.js"></script>
    <link rel="stylesheet" href="pagina.css">
  </head>
  <body>
    <p>EN CONSTRUCCIÓN<p>
  </body>
</html>
```

Por lo tanto, lo que queda es incluir el atributo `manifest` con el nombre del fichero en la etiqueta HTML:

```
<html manifest="principal.appcache">
  ...
</html>
```

Mantener la integridad de la aplicación web

Una vez realizado ambos pasos descritos anteriormente el sitio web estará preparado para funcionar con la caché de aplicación. El funcionamiento interno recomendado por la W3C es el siguiente:

Al visitar el sitio web por primera vez se descargan los recursos que aparecen en el manifiesto. En caso de que sea la segunda vez que se visita el sitio web y si el contenido del manifiesto ha cambiado, se descargan nuevamente todos los recursos hacia la caché de aplicación.

El problema se encuentra cuando al pasar del modo fuera de línea al modo en línea se ha actualizado en el servidor algún recurso que se encuentre en la caché de aplicación. Debido a que el manifiesto no se ha modificado (por que el recurso no ha cambiado de nombre), tampoco se recarga la página desde el servidor. Como resultado de esto se sigue mostrando la versión antigua contenida en la caché de aplicación.

La solución es actualizar el manifiesto. Y para no modificar el contenido del fichero recomendamos incluir un comentario con la versión o fecha de última actualización del manifiesto, para ir modificándolo a partir de los cambios del contenido de los recursos listados.

¿SABÍAS QUE...?

Los navegadores actualmente implementan mecanismos para conocer el estado de la caché de aplicación.

ACTIVIDADES 8.3

- Investigue los navegadores que actualmente implementan la caché de aplicación y sus características especiales.
- ¿De qué manera puedo conocer, en mi aplicación del lado del cliente si el navegador se encuentra en línea o fuera de línea? Sugerencia: revisar la especificación *Offline Web Applications* de la W3C: *www.w3.org/TR/offline-Webapps/*

RESUMEN DEL CAPÍTULO

En este capítulo hemos presentado los conceptos y tecnologías principales del almacenamiento de datos del lado del cliente, así como la posibilidad del trabajo fuera de línea de HTML 5. Hemos estudiado las características de los objetos DOM *Storage* y su relación con la tecnología anterior más utilizadas: las *cookies*.

HTML 5 soporta actualmente la caché de los recursos de las aplicaciones web. El objetivo de estas nuevas características, junto con la API de IndexedDB es la posibilidad de obtener aplicaciones web y sitios web que funcionen correctamente cuando no tienen la conexión con el servidor web (modo fuera de línea).

IndexedDB es una API para el almacenamiento de gran cantidad de datos de manera estructurada y es el reemplazante natural de WebSQL cuya especificación ha sido abonada en el año 2010 por la W3C. De esta manera pasamos de una base de datos relacional a una orientada a objetos JavaScript para realizar una persistencia directa a partir de los contenedores y utilizando índices y cursores para las consultas.

Por último, en HTML 5 aparece el concepto de caché de aplicación. La funcionalidad consiste en obtener toda la información del sitio web antes de que sea explícitamente pedida por el usuario de tal manera que una vez almacenadas en la caché de aplicación podrán ser utilizadas en modo fuera de línea.

EJERCICIOS PROPUESTOS

1. Implemente una agenda de tareas diarias en HTML que almacene en el lado del cliente las tareas (utilizar IndexedDb):

- La agenda constará de dos páginas.
- En la primera página se cargarán los datos de la tarea:
  - Nombre de la tarea.
  - Descripción.
  - Importancia.
  - Horas estimadas en la realización de la tarea.
- En la segunda página (visualizable a partir de la primera página) se listarán las tareas almacenadas.
- Las tareas pueden borrarse.

TEST DE CONOCIMIENTOS

1 ¿Podría mantener un fichero XML en una *cookie*? Las características del fichero son las siguientes: Nombre: `proveedores.xml`, cantidad de caracteres: 1.400, tamaño: 10.240 bytes:

a) Sí, es la opción más recomendable.
b) Sí, puedo hacerlo aunque existen tecnologías más modernas y, por tanto, debería probar esas.
c) Depende, el fichero XML será solo texto plano por lo que el servidor tendrá que tener la funcionalidad necesaria para utilizar correctamente la *cookie*.
d) No puedo hacerlo por problemas con el tamaño del fichero.

2 DOM Storage se refiere a:

a) Un árbol DOM para acceder a los elementos HTML.
b) Un tipo de almacenamiento de datos en el cliente y servidor.
c) Un almacenamiento del lado del cliente HTML 5 que puede ser: `localStorage` o `sessionStorage`.

3 Indicar la frase que no sea correcta:
- a) Los objetos `localStorage` pueden almacenar más bytes que las *cookies*.
- b) Los objetos `sessionStorage` no tienen fecha de expiración.
- c) La persistencia en los objetos *DOM Storage* es instantánea.

4 En caso de buscar mantener información entre sesiones necesitaría:
- a) Un objeto `localStorage`.
- b) Un objeto `sessionStorage`.
- c) *Cookies* sin fecha de expiración.
- d) Tanto un objeto `localStorage` como `sessionStorage` es correcto.
- e) Ninguna respuesta es correcta.

5 ¿Qué es un *Object Store*?
- a) Un objeto JavaScript.
- b) Un mecanismo de almacenamiento.
- c) Una tabla de base de datos relacional.

6 El cursor es sinónimo de:
- a) Un *array* de elementos accesibles por un índice numérico.
- b) Una posición de memoria donde persisten datos del cliente.
- c) El elemento actual, así como la posibilidad de recorrer una lista de elementos.

7 Necesito incluir un recurso en la caché de aplicación. Debo tenerlo en:
- a) `CACHE`.
- b) `NETWORK`.
- c) `FALLBACK`.
- d) `CACHE o FALLBACK`.

9 Integración avanzada de componentes

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer los formatos y mecanismos de reproducción multimedia actuales.
- ✓ Detallar la adición de elementos de vídeo HTML 5 en el desarrollo de sitios web.
- ✓ Detallar la adición de elementos de audio HTML 5 en el desarrollo de sitios web.
- ✓ Indicar las características de geolocalización especificadas por la API de geolocalización de HTML 5.
- ✓ Describir los aspectos de seguridad y usos principales de la geolocalización.

En este capítulo presentamos los conceptos de integración avanzada de componentes multimedia y de geolocalización descritos por HTML 5. Vamos a comentar los diferentes formatos soportados por los navegadores actuales y su relación con HTML 5. También vamos a detallar los mecanismos de inclusión de elementos multimedia (audio y vídeo) en páginas HTML. Por último, comentaremos las funcionalidades de geolocalización actuales y la API de geolocalización de HTML 5, haciendo especial hincapié en los aspectos de seguridad y capacidad de uso.

9.1 REPRODUCTORES MULTIMEDIA Y PLUGINS ASOCIADOS

Al inicio Internet solamente se utilizaba para el intercambio de texto. Actualmente se ha convertido en un espacio plural, donde el intercambio de información escrita se encuentra complementado por sonidos, vídeos o animaciones. Justamente esto es a lo que se refiere la multimedia (Real Academia Española, 2001).

Para una correcta utilización de este tipo de ficheros es necesario complementos o *plugins* en los navegadores que amplían sus capacidades. Aun así, los navegadores actuales mantienen un soporte para una gran cantidad de formatos multimedia. En ese sentido, los diferentes formatos relacionados con *plugins* como RealPlayer o QuickTime han ido ganando el mercado, aunque se ha reconocido a la tecnología Flash como la tecnología más adecuada hasta el momento (Shankland, 2010).

Como competencia a estas tecnologías se encuentra HTML 5 que incluye etiquetas especiales de audio y vídeo y que no necesita de un complemento especial para su ejecución. En la Tabla 9.1 podemos ver un resumen comparando las tecnologías Flash y HTML 5.

Tabla 9.1 Relación entre la tecnología Flash y la multimedia con HTML 5

Atributo	Flash	HTML 5
Ventajas	Animación.	Mejor accesibilidad web y JavaScript.
Soportes no tradicionales	Soporte de *webcams*.	Soporte de iPad y iPhone.
Seguridad	*Streaming* de vídeo y audio para evitar las descargas.	Es código abierto. Podemos tener acceso a la URL del fichero multimedia.

9.1.1 REPRODUCCIÓN DE VÍDEOS EN HTML 5

De acuerdo con las diferentes tecnologías para reproducir vídeos existen también diferentes formatos de ficheros. Por ejemplo, podemos encontrar los ficheros con extensión "`.flv`" para vídeos en Flash. En este y otros casos es necesario tener un complemento en el navegador para decodificar el formato.

En el caso de HTML 5 es el propio navegador el que tiene la capacidad de realizar la decodificación (Beres, 2011). Para ello utiliza estándares como H.264, Ogg o WebM, los tres que han sido implementados por la mayoría de los

navegadores actuales. A continuación podemos ver en la Tabla 9.2 los formatos de vídeos relacionados con HTML 5 más utilizados por los principales navegadores.

Tabla 9.2 Formatos de vídeo soportados por los principales navegadores

Nombre	Ogg Theora	H.264	WebM VP8
Internet Explorer	No	Sí	Se puede agregar un componente
Mozilla Firefox	Sí	No	Sí
Google Chrome	Sí	Sí	Sí
Safari	Se puede agregar un componente	Sí	Se puede agregar un componente

Respecto de la especificación actual de HTML 5, esta no indica un formato de vídeo determinado aunque se espera que en el futuro exista por lo menos un formato de vídeo que se convierta en el estándar de facto (*World Wide Web Consortium*, 2011).

Entre las características más importantes buscadas para un formato de vídeo encontramos la buena compresión, la calidad de la imagen y que sea libre (por lo tanto que no tenga costes de licencia o este protegido por patentes). El formato de vídeo que ha tenido el apoyo inicial de HTML 5 ha sido Theora, quien junto con el formato de audio Vorbis conforman el contenedor multimedia Ogg (Pfeiffer, 2003). Otro de los formatos más utilizados es H.264 por su capacidad de compresión y calidad del vídeo (Wiegang, 2003). El problema principal es que se encuentra protegido por patentes por lo que es necesario pagar una licencia. Los que actualmente impulsan este estándar son Microsoft y Apple. Por último, encontramos el estándar VP8 (que forma parte del contenedor multimedia WebM). Es un estándar libre impulsado por Google, Firefox y Ópera.

Independientemente de ello, la manera de incluir vídeos en HTML 5 es a partir de la etiqueta `video`. Junto con esta etiqueta, otra de las principales es la etiqueta `source` donde se identifica la URL y el tipo de vídeo a ser mostrado. Como podemos ver en el ejemplo existen diferentes orígenes del vídeo. De esta manera el navegador elegirá la mejor opción.

```
<video width="640" height="480" controls="controls">
  <source src="video.mp4" type="video/mp4"/>
  <source src="video.ogg" type="video/ogg"/>
  Formato de video no soportado.
 </video>
```

En la Tabla 9.3 describimos los atributos que podemos utilizar en las etiquetas `video`. En el caso del ejemplo modificamos el alto y ancho del visor. Así mismo incluimos los botones de control del vídeo mediante al atributo `controls`.

Tabla 9.3 Atributos de los elementos "video" y "audio" de HTML 5

Atributo	Descripción
`preload`	Indica si el navegador podrá descargar el vídeo o audio antes de que el usuario haga clic en el botón de inicio.
`autoplay`	Indica si el navegador iniciará automáticamente el vídeo o audio.
`loop`	Indica si el vídeo o audio se repetirá una vez finalizado.
`poster`	Indica la imagen que se mostrará mientras el vídeo es cargado.
`controls`	Especifica que los controles del vídeo se mostrarán como los botones de inicio o fin.
`src`	Indica la URL del vídeo.
`width y height`	Indican el ancho y alto del visor en píxeles.
`muted`	El audio del vídeo puede ser silenciado.

Junto con los atributos, que pueden ser incluidos dentro de la etiqueta, también están las propiedades del elemento `video`. Estas propiedades son utilizadas por los reproductores multimedia JavaScript para interaccionar con el vídeo o audio. En la Tabla 9.4 y Tabla 9.5 podemos ver las principales propiedades y eventos utilizados por los desarrolladores.

Tabla 9.4 Principales propiedades de los elementos "video" y "audio"

Propiedad	Descripción
`buffered`	Un objeto que indica el rango de tiempo descargado.
`bufferedBytes`	Un objeto que indica el rango de bytes descargado.
`bufferingRate`	Indica el promedio de bits por segundo descargado.
`currentLoop`	Indica la cantidad de veces que se ha reproducido el audio o el vídeo en el bucle actual.
`currentTime`	Indica la cantidad de segundos que ha sido reproducido el fichero.
`duration`	Indica el número total de segundos del audio o vídeo.
`ended`	Indica si la reproducción ha finalizado.
`start`	Indica el lugar, en segundos, donde se encuentra actualmente la reproducción.
`volume`	Indica el volumen actual de reproducción. Es una propiedad editable.
`readyState`	Indica si el audio o vídeo puede ser reproducido. Es una codificación numérica que va del 0 al 3: 0. El contenido no está habilitado. 1. El fotograma actual puede ser reproducido. 2. El contenido puede ser reproducido. 3. El contenido puede ser reproducido de inicio a fin.

Tabla 9.5 Principales eventos de los elementos de "video" y "audio"

Evento	Descripción
abort	La descarga ha sido abortada.
canplay	La reproducción puede empezar.
dataunavailable	No existe contenido a ser reproducido.
pause	La reproducción se ha pausado.
play	Se ha empezado a reproducir el audio o vídeo.
volumechange	Se ha cambiado el volumen o la propiedad `muted`.
abort	La descarga ha sido abortada.
canplay	La reproducción puede empezar.
dataunavailable	No existe contenido a ser reproducido.

Existe una tercera etiqueta muy importante al reproducir un vídeo, la etiqueta `track` (*World Wide Web Consortium*, 2011). Este elemento es utilizado para especificar subtítulos, nombres de capítulos u otro tipo de textos que serán visibles mientras el vídeo se reproduce. Aun así, todavía no es implementada por los navegadores actuales.

```
<video width="640" height="480" controls="controls">
  <source src="video.mp4" type="video/mp4"/>
  <source src="video.ogg" type="video/ogg"/>

  <track src="subtitulo_es.vtt" kind="subtitles" srclang="es"
    label="Español">
  <track src="subtitle_en.vtt" kind="subtitles" srclang="en"
    label="English">

  Formato de video no soportado.
</video>
```

En el ejemplo anterior hemos agregado dos subtítulos, uno en español y otro en inglés. A la etiqueta `track` le hemos añadido cuatro atributos:

- **Src**. Tiene la URL del fichero que contiene el subtítulo.
- **Kind**. Indica el tipo de añadido. Puede ser un subtítulo, transcripciones de texto y audio, descripciones del vídeo, título del vídeo, etc.
- **srclang y label**. Identifica el lenguaje. El primero en formato oficial (Phillips, 2009) y el segundo en su nombre amigable.

Actualmente la W3C se encuentra trabajando en una propuesta a partir del grupo *Web Media Text Tracks Community Group*, creando ejemplos y definiendo formatos de subtítulo. En este caso el formato por el que la W3C apuesta es *Timed Text Markup Language* (TTML) y WebVTT (*Web Media Tracks Community Group*, 2011).

¿SABÍAS QUE...?

Cualquier texto entre las etiquetas `video` será visualizado en caso de que el vídeo no sea soportado por el navegador.

ACTIVIDADES 9.1

- Investigue el motivo por el cual Firefox no utiliza el protocolo H.264 para la visualización de vídeos.
- Investigue la manera en la que el visualizador jPlayer puede ser personalizado.

9.1.2 REPRODUCCIÓN DE AUDIO EN HTML 5

Al igual que con los vídeos, HTML 5 también incluye una nueva etiqueta para los audios. Las características, propiedades y eventos se encuentran descritas en la Tabla 9.3, Tabla 9.4 y Tabla 9.5, aunque existen algunas que no apliquen (las relacionadas solo con las imágenes).

La forma más sencilla de incluir audio es a partir del siguiente ejemplo:

```
<audio src="musica.mp3"></audio>
```

Al igual que con el vídeo, HTML 5 no recomienda ningún formato específico. Entre los formatos más utilizados (por su alta calidad y gran capacidad de compresión) se encuentran el MP3, originalmente desarrollado para vídeos. Así mismo, se encuentra el formato WAVE (extensión `.wav`) desarrollado por IBM y Microsoft, y el formato Vorbis que forma parte del proyecto Ogg. El soporte actual de ficheros de audio de acuerdo con cada navegador lo encontramos en la Tabla 9.6.

Tabla 9.6 Formatos de vídeo soportados por los principales navegadores

Nombre	Vorbis (Ogg y WebM)	MP3	WAV
Internet Explorer	No	Sí	Sí
Mozilla Firefox	Sí	No	Sí
Google Chrome	Sí	Sí	No
Safari	No	Sí	Sí
Opera	Sí	No	Sí

En el caso del ejemplo anterior hemos utilizado la propiedad `src` para incluir el fichero multimedia. También podemos utilizar la etiqueta `source` para incluir más de una referencia a elementos de audio de tal manera que el navegador reproduzca el primer formato que reconozca. En el siguiente ejemplo también hemos incluido la propiedad `controls` para mostrar al usuario el reproductor por defecto del navegador:

```
<audio controls>
  <source src="musica.wav" type="audio/mpeg" />
  <source src="musica.ogg" type="audio/ogg" />
  <source src="musica.mp3" type="audio/mpeg" />
</audio>
```

Este reproductor por defecto puede modificarse con las propiedades globales de la etiqueta `audio` como si se tratase de cualquier otra etiqueta HTML (esto sucede también con la etiqueta `video`). Así, por ejemplo, podemos modificar el color de fondo a partir del atributo global `style` (como podemos ver en la Figura 9.1):

```
<audio controls style="background-color:#0000CC">
```

Figura 9.1. *Ejemplo de reproductores de audio HTML 5 con diferentes colores*

Al igual que con el vídeo, el contenido entre las etiquetas `audio` o `video` es interpretado por el navegador. Esto puede ser utilizado por los desarrolladores para incluir soporte a otro tipo de reproducción multimedia como, por ejemplo, Flash.

En el siguiente ejemplo hemos incorporado un elemento `object` de tal manera que sea posible reproducir el audio mediante Flash en caso de que el navegador no tenga soporte para HTML 5. En el ejemplo anterior hemos incluido en la propiedad `data` la URL del reproductor y el nombre del fichero que se reproducirá.

```
<audio controls>
  <source src="archivo.ogg" type="audio/ogg" />
  <source src="archivo.mp3" type="audio/mpeg" />

  <object type="application/x-shockwave-flash"
    data="player.swf?audioUrl=musica.mp3">
  </object>
</audio>
```

Por último, es posible desarrollar los reproductores que manejan los ficheros de audio y vídeo. En el siguiente ejemplo hemos incorporado un botón que llamará a la función JavaScript "`reproducir_pausar`" en cada clic. La función preguntará si el elemento `audio` se encuentra pausado y en ese caso lo reproducirá. En otro caso lo pausará.

```
<audio id="reproductor"
  <source src="musica.mp3" type="audio/mp3">
  <source src="musica.oga" type="audio/ogg">
  Your user agent does not support the HTML5 Audio element.
</audio>

<button type="button" name="bReproductor"
  onclick="reproducir_pausar()">Reproducir</button>

<script>
  function reproducir_pausar () {
    var reproductor = document.getElementById("reproductor");
    var bReproductor =
      document.getElementById("bReproductor");

    if (reproductor.paused) {
      bReproductor.value="Detener";
      reproductor.play();
    } else {
      bReproductor.value="Reproducir";
      reproductor.pause();
    }
  }
</script>
```

9.2 GEOLOCALIZACIÓN

Cada día es más común hablar de la geolocalización. Este término está relacionado con la ubicación geográfica de un objeto. Las tecnologías que hacían posible la geolocalización estaban relacionadas con dispositivos GPS dedicados. Actualmente, con los teléfonos móviles la geolocalización es una funcionalidad básica. En ese sentido, unas de las características de HTML 5 es la inclusión de aspecto de geolocalización (Satrom, 2011).

Existen también otro tipo de mecanismos para conocer la ubicación de un usuario de Internet. Por ejemplo, a partir de su dirección IP mediante servicios que indican la ciudad de donde proviene. Como podemos apreciar, ésta es una información estática y cuyo objetivo no está orientado al usuario final.

En el caso de HTML 5 la geolocalización tiene como objetivo principal servir de utilidad al usuario del sitio o aplicación web. Es una información dinámica y al tratarse de una API incluida en un navegador puede utilizar los aspectos de programación y el hardware del dispositivo que lo hospeda.

En ese sentido, una de las premisas es que el mismo navegador será quien obtenga la información de localización. Desde el punto de vista de seguridad esto es muy importante ya que el usuario podrá permitir o denegar la posibilidad de publicar su localización.

9.2.1 API DE GEOLOCALIZACIÓN DE HTML 5

La especificación de la API de geolocalización se está llevando a cabo por el grupo *Geolocation Working Group* de la W3C. Esta API define una interfaz de alto nivel para la información de localización asociada con el dispositivo que hospeda la aplicación (Geolocation Working Group, 2010).

En ese sentido la API es independiente de la fuente de información que proporciona la ubicación. Entre las fuentes principales están el *Sistema de Posicionamiento Global* (GPS), las direcciones MAC o la localización GSM. Por lo tanto no existen garantías de que a partir de la API obtengamos la ubicación real del dispositivo.

Tabla 9.7 Atributos de localización de la API de geolocalización

Nombre	Descripción
`latitude y longitude`	Coordenadas geográficas especificadas en grados decimales.
`altitude`	Indica la altura de la posición, especificado en metros.
`speed`	Indica la velocidad en metros por segundo.

En la Tabla 9.7 detallamos los parámetros de localización indicados por la API. En los ejemplos típicos la información de localización es representada mediante las coordenadas de latitud y longitud.

```
<script>
  function localizar(){
    navigator.geolocation.getCurrentPosition(mostrar);
  }

  function mostrar(position) {
    var latitud = position.coords.latitude;
    var longitud = position.coords.longitude;
    alert("Coordenadas: " + latitud + " - " + longitud);
  }
</script>
```

El ejemplo muestra el objeto "`geolocation`" indicado en la API. Este objeto tiene un método denominado `getCurrentPosition()` con un parámetro. Este primer parámetro contiene la función que será llamada en caso de que el método `getCurrentPosition()` finalice con éxito.

Como resultado llamamos a la función `mostrar()` que recibirá como parámetro un objeto con la posición actual del dispositivo.

Como siguiente paso es necesario gestionar los errores, para mejorar la experiencia del usuario y debido a que la precisión de la geolocalización depende muchas veces del lugar donde se encuentre el dispositivo.

En este caso agregamos un segundo argumento al método `getCurrentPosition()`, que será la función a ser llamada en caso de que exista algún error. En este caso la función recibe el nombre de `errores()`.

Los errores que podemos obtener son:

- **Permiso denegado**, por parte del usuario.
- **Posición no disponible**, si se ha perdido la conexión.
- **Timeout**, si se ha excedido el tiempo máximo para calcular la localización del usuario.
- **Error desconocido**, en cualquier otro caso.

```
<script>
 function localizar() {
  navigator.geolocation.getCurrentPosition(mostrar, errores);
 }

 function mostrar(position){
  var latitud = position.coords.latitude;
  var longitud = position.coords.longitude;
  alert("Coordenadas: " + latitud + " - " + longitud);
 }

 function errores(codigo_error){
  if(codigo_error.code==0){
    alert("El error es desconocido");
  }
  if(codigo_error.code==1){
    alert("Permiso denegado");
  }
 }
</script>
```

Así mismo, podemos incorporar un tercer parámetro al método `getCurrentPosition()` con opciones del tiempo máximo de cálculo de la localización o aumentar la precisión de la localización. Por último, es necesario controlar si el navegador soporta la geolocalización por lo que podemos preguntar por la existencia o no del objeto `navigator.geolocation`.

Consideraciones de seguridad

Con este tipo de tecnologías obtenemos la localización geográfica del dispositivo cliente. Y en casi todos los casos esta información también sirve para conocer la ubicación del usuario del dispositivo. Por lo tanto, esto podría comprometer la privacidad del usuario. En ese sentido, desde la W3C no se permite el intercambio de información de localización sin el permiso explícito del cliente. Así mismo, se recomienda su encriptación.

La información de localización deberá ser obtenida y mantenida en un espacio de memoria por el navegador. Referido a la petición de información la API de geolocalización indica lo siguiente:

- ✓ La petición de información de localización deberá realizarse solamente cuando necesitemos la localización.
- ✓ La información de localización deberá ser utilizada solamente para la tarea que ha dado lugar a la petición.
- ✓ La información de localización debe ser eliminada una vez la tarea esté completada.
- ✓ La retención de información de localización posterior a la tarea solo puede realizarse mediante permiso explícito del usuario.
- ✓ En caso de almacenamiento de la localización el usuario podrá actualizar o borrar la información.

Por último, la W3C recalca la obligatoriedad de que el usuario conozca en todo momento que el navegador está recolectando datos de localización, su propósito y el tiempo de retención de dicha información.

9.2.2 UTILIZACIÓN DE LA GEOLOCALIZACIÓN

Como hemos comentado anteriormente la geolocalización será cada vez más importante, aumentando los usos no convencionales de la misma. En ese sentido la API de geolocalización de la W3C enumera algunas formas de utilización de la geolocalización en teléfonos móviles (Geolocation Working Group, 2010).

- **Encontrar puntos de interés cerca de la ubicación del usuario**. Esto es muy común y desde hace años podemos encontrar sitios web que indican las atracciones turísticas, museos, hoteles y otros servicios relacionados con alguna ubicación. En eso casos el usuario debe indicar la ciudad y el radio de kilómetros a la redonda. Con las nuevas tecnologías es posible conocer la ubicación del dispositivo móvil en todo momento por lo que la información es totalmente personalizada y al instante.
- **Relacionar capturas con la localización**. Los dispositivos móviles incorporan otras características como cámaras, vídeos, agenda, etc. Así como las cámaras actuales incorporan la fecha y hora en la que se ha capturado una fotografía los dispositivos móviles pueden incorporar información de la ubicación. Incluso se pueden incorporar otros servicios como, por ejemplo, los mapas para mejorar la experiencia del usuario.
- **Alerta de proximidad**. Actualmente las alarmas tienen en cuenta la proximidad en el tiempo. Por ejemplo, una reunión a las 8 de la mañana. Debido a que actualmente el teléfono móvil une tanto la agenda personal como la geolocalización se pueden disparar avisos por proximidad física.
- **Información local actualizada**. La información de actualidad también se modifica de acuerdo con la localización geográfica: el tiempo, el horario, etc.

ACTIVIDADES 9.2

- Investigue sobre la localización GSM.
- Indique otros usos de la geolocalización para un teléfono móvil.

RESUMEN DEL CAPÍTULO

Al inicio Internet solamente se utilizaba para el intercambio de texto. Actualmente se ha convertido en un espacio plural, donde el intercambio de información escrita muchas veces se encuentra complementado por sonidos, vídeos o la localización del usuario. HTML 5 incorpora mecanismos multimedia y de geolocalización que permiten una mejor experiencia de usuario.

Se han incorporado dos nuevos elementos a HTML: el elemento `video` y el elemento `audio`. Mediante su inclusión en código HTML permiten reproducir, controlar y visualizar ficheros multimedia. En ese sentido es muy importante el soporte que cada navegador hace de los formatos que puede reproducir. A ese respecto HTML 5 es independiente de los formatos, permitiendo incluir más de un tipo, así como texto plano que será interpretado en caso de no encontrar el soporte adecuado. Esto último permite la visualización de multimedia con mecanismos anteriores a HTML 5.

Mediante la API de geolocalización es posible gestionar, mantener y utilizar la localización del usuario mediante programación del lado del cliente (por ejemplo con JavaScript). En ese sentido la especificación es clara respecto de la privacidad de los usuarios: los navegadores no pueden enviar la información de ubicación sin el permiso expreso del usuario y deben eliminar dicha información una vez utilizada.

EJERCICIOS PROPUESTOS

1. Realice un reproductor de audio HTML 5 + CSS + JavaScript. Las características del control de audio deben ser las siguientes:

- Botón de Pausa/Reproducción.
- Visualizar el tiempo de reproducción actual.
- Visualizar el nombre del fichero de audio que se está reproduciendo actualmente.
- Al pausar el audio desaparecen todos los elementos del visualizador excepto el botón de reproducción.
- Debe funcionar como mínimo en dos de los siguientes navegadores: Google Chrome, Internet Explorer, Firefox, Safari, Ópera.

TEST DE CONOCIMIENTOS

1 ¿Cuál es actualmente la tecnología más utilizada para la visualización de ficheros multimedia en sitios web?
- a) Flash.
- b) HTML 5.
- c) MP3.
- d) VLC Media Player.

2 Si necesito dificultar la descarga del fichero multimedia que será mostrado en el sitio web, ¿Qué tecnología es la más apropiada?
- a) Flash.
- b) HTML 5.
- c) Todas son igual de accesibles.

3 Las visualizaciones multimedia en HTML 5 dependen del navegador:
- a) Verdadero, y depende de los formatos que soporte el navegador.
- b) Falso. Al ser HTML se visualiza en cualquier navegador y no necesita de componentes adicionales.

4 La información de geolocalización en HTML 5 proviene:
- a) Del navegador cliente.
- b) Del servidor.
- c) La fuente de la información de geolocalización es indistinta.
- d) Es necesario que provenga tanto del cliente como del servidor para comprobar la precisión de la localización.

5 En HTML 5 la posición del usuario es comunicada por defecto sin su consentimiento:
- e) Verdadero.
- a) Falso.

Material adicional

El material adicional de este libro puede descargarlo en nuestro portal Web: *http://www.ra-ma.es*.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

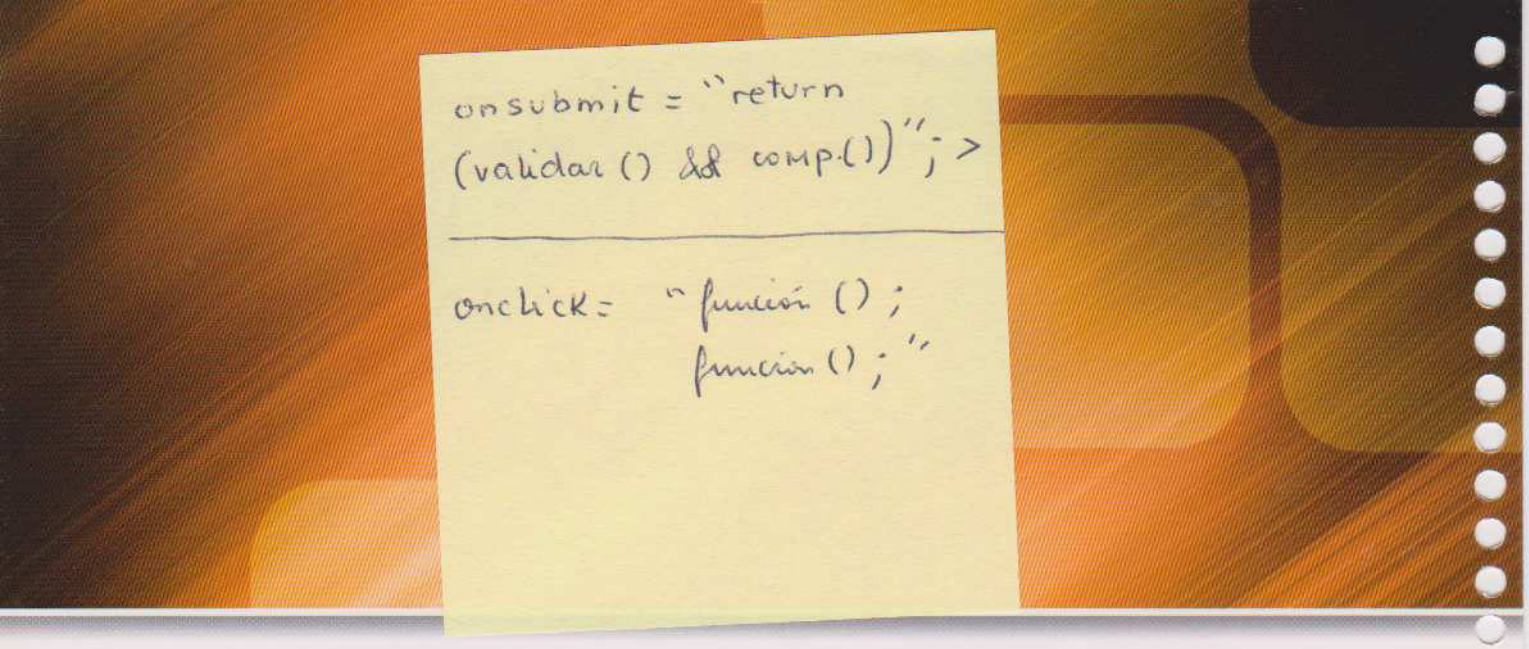
Podrá localizar el número de ISBN en la página 2 (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

Índice Alfabético



La presente obra está dirigida a los estudiantes del Ciclo Formativo **Desarrollo de Aplicaciones Web** de Grado Superior, en concreto para el módulo profesional **Desarrollo Web en Entorno Cliente**.

Los contenidos incluidos en este libro abarcan los conceptos básicos y las técnicas habituales para el desarrollo de aplicaciones web que serán ejecutadas en un cliente web. Además, se presentan acompañados de ejemplos intuitivos que sirven para ilustrar dichos conceptos y técnicas. Como punto de partida se introducen brevemente los diferentes navegadores, sus principales características y se presentan las arquitecturas y tecnologías existentes para el desarrollo de este tipo de aplicaciones. A continuación, se abordan los puntos principales relacionados con el uso de estas tecnologías, describiendo la sintaxis de Javascript. Posteriormente, se estudia el desarrollo de páginas web dinámicas. También se presenta el modelo de objetos, el modelo de gestión de eventos de Javascript y el modelo de objetos del documento que permite modificar la apariencia de las páginas web utilizando sentencias de ese lenguaje. Finalmente, se estudian los aspectos avanzados del desarrollo web en entorno cliente, que pasan por el desarrollo de aplicaciones AJAX, basadas en el uso extensivo de Javascript y la utilización de XML y/o JSON como formato de almacenamiento e intercambio de datos.

Todos los capítulos incluyen actividades y ejemplos con el propósito de facilitar la asimilación de los conocimientos tratados. Así mismo, se incorporan test de conocimientos y ejercicios propuestos con la finalidad de comprobar que los objetivos de cada capítulo se han asimilado correctamente. Además, reúne los recursos necesarios para incrementar la didáctica del libro, tales como un glosario con los términos informáticos necesarios, bibliografía y documentos para ampliación de los conocimientos.

En la página web de **Ra-Ma** (**www.ra-ma.es**) se encuentra disponible el material de apoyo y complementario.

ra-ma.es

Ra-Ma®