

Argentina \$ 49

PROGRAMADOR WEB Full Stack

Desarrollo frontend y backend

Integración de HTML5 y JavaScript

DOM y elementos HTML \ Objetos location, window, history y navigator

action mostrarAzul() {
 document.getElementById('imagen').src = 'azul.png';
}
function mostrarRojo() {
 document.getElementById('imagen').src = 'rojo.png';
}
function mostrarAmarillo() {
 document.getElementById('imagen').src = 'amarillo.png';
}
function mostrarGris() {
 document.getElementById('imagen').src = 'gris.png';
}

+

EVENTOS
JAVASCRIPT
PASO A PASO

Domina las últimas tecnologías para sitios web responsive y dinámicos



PROGRAMADOR WEB Full Stack

Desarrollo frontend y backend

Distribuidores. Argentina. Capital: Vaccaro Hnos. Representantes de Editoriales S.A., Av. Entre Ríos 919 1er piso (1080), Ciudad de Buenos Aires, Tel. +54 (011) 4305-3854/3908. Interior: Distribuidora Interplazas S.A. (DISA) Pte. Luis Sáenz Peña 1832 (C1135ABN), Buenos Aires, Tel. +54 (011) 4305-0114. México DF: Arredondo Distribuidora S.A. de C.V. Iturbide N° 18-d frente a cine Palacio Chino - Col. Centro, Tel. 55126069. México. Interior: Compañía distribuidora de periódicos, libros y revistas S.A. de C.V. Centeno N° 580. Col. Granjas México. Delegación: IZTACALCO. Distrito Federal. C.P. 08400. Tel. 5128-6670. Perú: Distribuidora Bolivariana S.A., Av. República de Panamá 3635 piso 2 San Isidro, Lima, Tel. 511 4412948 anexo 21. Uruguay: Espert S.R.L., Paraguay 1924, Montevideo, Tel. 5982-924-0766. Venezuela: Distribuidora Continental Bloque de Armas, Edificio Bloque de Armas Piso 9.º, Av. San Martín, cruce con final Av. La Paz, Caracas, Tel. 58212-406-4250. Curso visual y práctico Servicio Técnico. Editor responsable: Fox Andina S.A., 15 de Noviembre de 1889, 1339, C1130ABE, Ciudad Autónoma de Buenos Aires, Argentina. Fecha de edición y Copyright © III-MMXVII, ISSN: 2545-6865. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio, sin el permiso previo y por escrito de esta casa editorial. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen, analizan o publicitan. Las notas firmadas reflejan la opinión de los autores sobre los temas tratados, sin que ello implique solidaridad de la revista con su contenido. Impreso en Sevagraf S. A., Costa Rica 5226 y Fulton, Grand Bourg, Argentina.

Dirección Editorial
Miguel Lederkremer

Edición
Fernando Ojam

Autor
Fernando Luna

Producción gráfica
Gustavo De Matteo

Atención al lector,
suscripciones y ventas
usershop@redusers.com
Ar.: +54 (011) 4110-8700
Mx: +52-55-8421-9660

Publicidad
publicidad@redusers.com
+54 (011) 4110-8700

SUSCRIBETE



INCLUYE
576 PÁG. EN 24 FASCÍCULOS
VERSIONES IMPRESA Y DIGITAL
DIPLOMA DE FINALIZACIÓN DE CURSO
CUPÓN DESCUENTO

011-4110-8700 | USERSHOP.REDUSERS.COM
USERSHOP@REDUSERS.COM

SÓLO VÁLIDO PARA ARGENTINA Y HASTA AGOTAR STOCK DE 100 UNIDADES.

7 Integración de HTML5 y JavaScript

- 01** Integración de HTML5 y JavaScript
- 04** Manejo de elementos HTML
- 08** Eventos JavaScript
- 12** Event Listener
- 14** Cuadros de diálogo
- 16** Objeto location
- 17** Objeto window
- 20** Objeto history
- 21** Objeto navigator

CONTENIDO ADICIONAL ONLINE

Código fuente,
elementos gráficos utilizados
en los ejemplos y otros contenidos
adicionales se pueden descargar en
redusers.com/u/programadorweb

Integración de HTML y JavaScript

A lo largo de las clases anteriores y en forma progresiva, fuimos conociendo las principales características del lenguaje HTML5, CSS, y en esta última instancia, de JavaScript. En esta entrega, analizaremos en detalle cómo combinar de manera efectiva las funcionalidades del lenguaje de scripting con las páginas HTML5, para poder construir webs que cuenten con dinamismo y funcionalidades flexibles gracias a la potencia que entrega JavaScript.

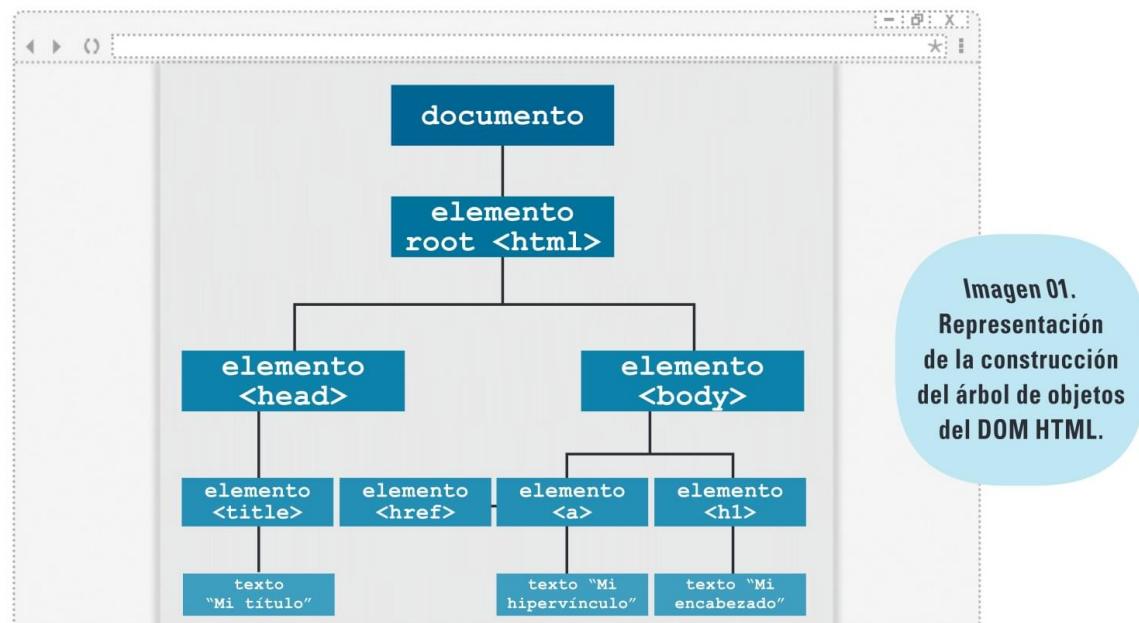
EL DOM HTML

Conocemos como DOM (*Document Object Model*) a un estándar impuesto por el **W3C**, que define la forma en la cual se puede acceder a los elementos que componen un documento HTML. El **DOM HTML** presenta las páginas HTML como una interfaz que permite, a los programas y scripts, acceder dinámicamente a cada tag HTML para leer su contenido, su estructura y sus estilos, aplicados a cada documento y, por supuesto, actualizar dichos datos. Dentro del DOM HTML se define la estructura de la siguiente manera:

- ✖ Los elementos HTML como objetos.
- ✖ Las propiedades de todos los elementos HTML.
- ✖ Los métodos de acceso a todos los elementos HTML.
- ✖ Los eventos para todos los elementos HTML.

Repasemos a través de la Imagen 01, la forma de constitución de los objetos HTML que componen el DOM. Con el modelo de objetos que plantea el DOM HTML, el lenguaje JavaScript puede potenciar los documentos HTML, transformando los estáticos en dinámicos que interactúan de forma directa con el usuario. Entre las posibilidades que JavaScript nos brinda, se encuentran:

- ✖ Cambiar el contenido de todos los elementos HTML de una página.
- ✖ Cambiar todos sus atributos.
- ✖ Cambiar los estilos CSS aplicados.
- ✖ Eliminar elementos HTML y sus atributos.
- ✖ Agregar nuevos elementos HTML y sus atributos.
- ✖ Crear nuevos eventos HTML en una página.
- ✖ Reaccionar ante determinados comportamientos de la página, ejecutando acciones desde JavaScript.



Métodos JS del DOM

Repasemos a continuación los métodos más comunes que podemos aplicar sobre el DOM, desde un script creado en JavaScript. Dentro de los métodos del DOM, existen acciones que es posible ejecutar desde JavaScript directamente sobre los elementos HTML. Cada propiedad de los elementos HTML es vista como un valor, que nosotros podemos establecer o cambiar desde JavaScript. Según lo representado en la Imagen 01, todos los elementos HTML son definidos como objetos para JavaScript. Estos objetos contienen propiedades y métodos que se pueden manipular desde el lenguaje de scripting.

GetElementByID

Construyamos a continuación un documento HTML que tendrá embebido un script. A esta página la llamaremos establecertextoenparrafo.html, y el código es el de la siguiente columna (el resultado de la página debe quedar como el de la Imagen 02):

El método getElementById ofrece la forma más efectiva para acceder a los elementos HTML. Tal como su nombre lo indica, debemos especificar a este método cuál es el elemento HTML al cual queremos acceder.

```
<script>
    document.getElementById("prueba").innerHTML = 'Hola mundo DOM!';
</script>
```

Tenemos que tener en cuenta que cuando creamos un documento HTML, cada uno de sus elementos deberá tener un id asociado. Esta será la forma más efectiva para trabajar dichos elementos desde el lado de JavaScript.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Trabajando con GetElementByID</title>
    </head>
    <body>
        <h1>Trabajando con GetElementByID</h1>
        <p id="prueba"></p>
    </body>
</html>
```

Al ejecutar esta página, solamente veremos el título establecido en el elemento H1. El elemento `<p>` no tiene ningún texto, solo cuenta con un ID establecido. Este ID nos permitirá desde JavaScript, manipular el contenido que se mostrará.

A continuación, agregaremos la línea de código que aparece debajo de este párrafo. Deberemos ubicar este fragmento de JavaScript al final del contenido HTML de la página, precisamente entre los elementos `</body>` y `</html>`. Nuevamente ejecutamos la página HTML, y debemos poder visualizar el texto establecido entre las comillas dobles, tal como aparece en la Imagen 03.

getElementById tiene asociada una propiedad, denominada `innerHTML`, la cual es utilizada para obtener o reemplazar el contenido de los elementos HTML.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Trabajando con GetElementByID</title>
5  ▶ 5.1
6      </head>
7      <body>
8          <h1>Trabajando con GetElem... </h1>
9      </body>
10
11
12
13 </html>
```

Trabajando con GetElementByID

Imagen 02.
Hasta ahora solo tenemos un formato de página estándar, sin contenido estático definido en el elemento `<p>`.

```

<head>
  <title>Trabajando con GetElementByID</title>
</head>
<body>
  <h1>Trabajando con GetElementByID</h1>
  <p id="prueba"></p>
</body>
<script>
  document.getElementById("prueba").innerHTML = 'Hola mundo DOM!';
</script>

```

Trabajando con GetElementByID

Hola mundo DOM!

Imagen 03.
Con el script ingresado,
desplegamos de
manera exitosa el texto
especificado, dentro
del elemento <p>.



COLECCIÓN DE ELEMENTOS

Como vimos, JS permite leer o modificar el contenido de un elemento determinado, utilizando el método `getElementById`. A su vez, el objeto `Document` posee otros métodos que permiten barrer el documento HTML y obtener por ejemplo cuántos elementos comparten un mismo atributo `class` o cuántos elementos comparten un mismo atributo `name`. Esto se realiza a través de los métodos `getElementsByClass` y `getElementsByName`. Como resultado devuelve una colección, y podremos modificar alguno de ellos, simplemente especificando el ID correspondiente, devuelto en dicha colección.

<EJERCICIO PRÁCTICO>

Copiar contenido de un elemento <p>



Armemos una página HTML con dos elementos `<p>`, y un hipervínculo. Las características adicionales del ejercicio son las siguientes:

- ☒ Debe haber un título H1.
- ☒ Debe contener dos elementos párrafo.
- ☒ Debe contener un hipervínculo.
- ☒ Debe existir una función JS dentro de la misma página.

Entonces, al presionar el hipervínculo, se deberá ejecutar la función JS. Dentro de la función JS, declaramos una variable, la cual debe almacenar el texto especificado en el primer elemento `<p>`. Luego de capturar el texto de dicho elemento, este se mostrará en el segundo elemento `<p>`. El resultado debe ser como el de la Imagen 04.

Si deseamos aplicar una decoración CSS al elemento `<a>`, tal como muestra la Imagen 04, podemos crear nosotros mismos una decoración específica, o aprovechar el siguiente código CSS:

```

a {
  font-size: 12px;
  width: 100px;
  background-color: #6699CC;
  text-decoration: none;
  color: white;
  padding: 10px 10px;
  margin: 8px 0;
  border: 1px solid, #6699CC;
  border-radius: 2px;
  cursor: pointer;
}

```

Imagen 04.
Al presionar el botón Copiar Texto, el segundo elemento <p> debe mostrar el texto que incluimos en el primer elemento <p>.

Manejo de elementos HTML

En el ejercicio anterior, vimos cómo podemos replicar, en otros elementos, un texto determinado, proveniente de un elemento HTML, o por qué no, agregado por nosotros mismos. Ahora veremos de qué manera JavaScript nos permite crear elementos HTML que no existen originalmente en el documento.

CreateElement()

El método **CreateElement()**, perteneciente también al **objeto document**, nos permite crear cualquier tipo de elemento HTML dentro del documento cargado en un navegador. Su sintaxis de uso es muy simple:

```
document.createElement("tipo _ de _ elemento");
```

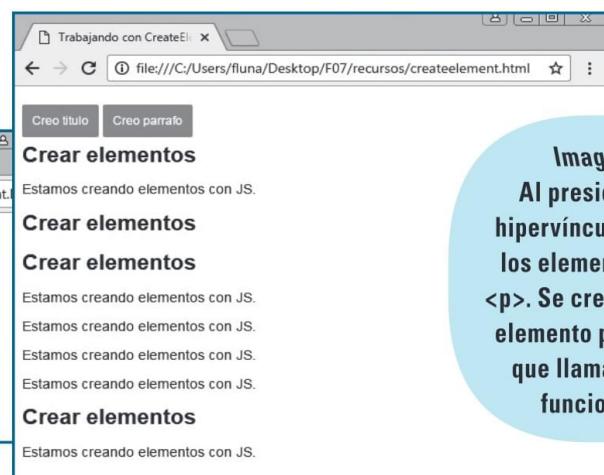
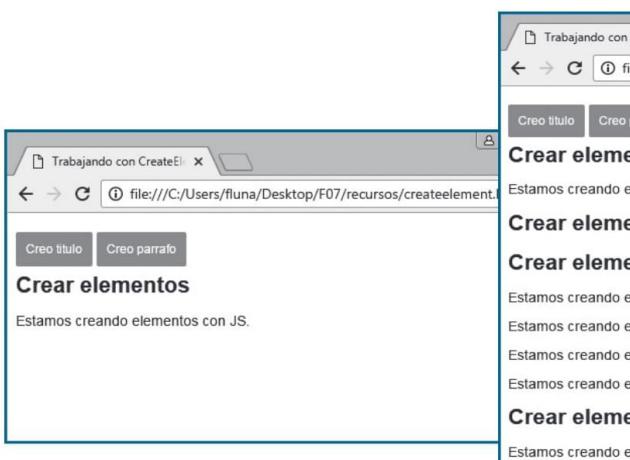
Veamos a continuación un ejemplo. Creamos un documento HTML con su estructura básica y, en <body>, agregamos dos hipervínculos que invocarán dos funciones JavaScript. El código de los hipervínculos es el siguiente:

```
<a href="javascript:creoTitulo();">Crea  
titulo</a>  
<a href="javascript:creoParrafo();">Crea  
parrafo</a>
```



Entre paréntesis debemos especificar el tipo de elemento por crear, como por ejemplo: p, h1 e img, entre otros. Cualquiera de los elementos anidados dentro del tag <body> pueden ser creados sin problema, invocando este método JS.

Ahora creamos un script que contendrá las dos funciones especificadas dentro de los elementos <a>. En la página siguiente, encontraremos el código.



En la función creamos dos variables: en la primera almacenamos la llamada al método createElement(), a la cual le pasamos el parámetro equivalente al elemento que deseamos crear. En la segunda función, invocamos el método createTextNode(), en el cual parametrizamos el texto que mostrará dicho elemento.

```
function creoTitulo() {  
    var t = document.createElement("H1");  
    var n = document.createTextNode("Crear elementos");  
    t.appendChild(n);  
    document.body.appendChild(t);  
}  
  
function creoParrafo() {  
    var parag = document.createElement("P");  
    var t = document.createTextNode("Estamos creando elementos con JS.");  
    parag.appendChild(t);  
    document.body.appendChild(parag);  
}
```

Invocamos el método appendChild() de la primera variable. Este adiciona el texto creado en la segunda variable al elemento que estamos creando.

Por último, invitamos de nuevo este último método pero del tag <body>, elemento padre de los anteriores.

El método appendChild() va anidando las propiedades y los métodos dentro de los objetos de nivel superior, hasta llegar finalmente al **elemento body**, que termina mostrándolos en el navegador en tiempo real sin necesidad de actualizar la página.

RemoveChild()

Ya aprendimos la forma de crear elementos HTML en un documento. Ahora veremos la manera de eliminar algún elemento existente y cómo controlar que no intente eliminar dos veces el mismo elemento. Utilizaremos para ello el **método removeChild()**. A continuación, creamos un documento HTML similar al anterior. En este creamos un título uno y dos elementos <p>. Todos deberán contener un texto obligatorio, por ejemplo:

```
<h1>Este es un t&iacute;tulo com&uacute;n y corriente</h1>  
<p id="parrafo1">Aqu&iacute; especificamos un p&aacute;rrafo cualquiera. Este no  
ser&aacute; eliminado.</p>  
<p id="parrafo2">Por el contrario, este segundo p&aacute;rrafo s&iacute; ser&aacute; eliminado.</p>
```

En el hipervínculo definimos el texto *Quitar párrafo* y, en su atributo href, agregamos el siguiente código:

```
javascript:quitoParrafo();
```

Por último, creamos una función JavaScript llamada quitoParrafo(), y agregamos en esta el siguiente código:

```
function quitoParrafo() {  
    var t2 = document.  
        getElementById("parrafo2");  
    if (!t2) {  
        alert('El elemento a eliminar  
        no existe.');  
    } else {  
        padre = t2.parentNode;  
        padre.removeChild(t2);  
    }  
}
```

El documento HTML se debe visualizar como en la Imagen 07.

En el ejemplo de texto HTML que especificamos, podemos ver que el primer elemento <p> tiene asociado un ID="parrafo1", mientras que el segundo



Imagen 07.
Nuestro documento
HTML ya está
listo para que su
segundo párrafo sea
eliminado.

Creamos una variable llamada padre, y que esta corresponde al valor del elemento asociado en la variable t2 más la propiedad parentNode. Esta propiedad retorna el nodo padre del nodo que especificamos para eliminar. Esto asegura que JS elimine correctamente el elemento indicado.

elemento <p> tiene asociado un ID="parrafo2". En el código JavaScript, mencionamos la eliminación del segundo elemento <p>, especificando dicho ID en document.getElementById(). Probemos entonces el funcionamiento de nuestro ejemplo, que debe verse como en la Imagen 08.



Imagen 08.
El segundo elemento
<p> fue eliminado
correctamente, y JS
controlará cualquier
nuevo intento de
eliminar un elemento
inexistente.

MÉTODO REPLACECHILD()

También podemos reemplazar elementos invocando al método replaceChild(). De la misma forma que el método removeChild(), replaceChild() permite cambiar el contenido o el texto de un elemento existente por otro nuevo contenido. Es muy utilizado en los desarrollos dinámicos donde se debe cambiar el contenido de una base de datos. Mientras que la base de datos se actualiza con una sentencia SQL, la información en pantalla se actualiza a través de una función JavaScript y así se evita la recarga de la página.

<EJERCICIO PRÁCTICO>

Agregar un elemento <p> dentro de <div>



Crearemos un nuevo documento HTML y, dentro de este, agregamos un elemento DIV. Luego creamos una función JavaScript, que invocará al evento createElement() para crear un nuevo elemento <p>. A diferencia del primer ejercicio, este elemento se deberá crear dentro del elemento div. El resultado debe ser como el de la Imagen 09.

Tengamos presentes los siguientes consejos para resolver este ejercicio:

- ☒ Al integrar un elemento DIV, este deberá contar con un ID.
- ☒ Las funciones getElementById() y appendChild() se deben combinar para obtener el resultado deseado.
- ☒ Si no controlamos el agregado de nuevos elementos, estos desbordarán el DIV.

Imagen 09. También es posible agregar nuevos elementos dentro de un elemento padre, como es el caso de DIV.

InsertBefore()

Este método permite insertar un nuevo nodo dentro de una lista, justo antes de un nodo ya existente. La posición donde deseamos insertar el nuevo nodo debe ser especificada mediante código. De la misma forma que utilizamos removeChild(), con insertBefore() debemos primero crear un elemento textNode, para luego agregarlo al elemento ; por último, insertamos el elemento en la lista.

Imagen 10. El método insertBefore() es útil tanto para agregar nuevos nodos en una lista como para pasar nodos de una lista hacia otra.

```
//Función JavaScript para insertar un nuevo nodo mediante el método insertBefore()

function insertar() {
    var newItem = document.createElement("LI");
    var textnode = document.createTextNode("Cafe");
    newItem.appendChild(textnode);
    var list = document.getElementById("Lista");
    list.insertBefore(newItem, list.childNodes[0]);
}
```

Eventos JavaScript

A través de los eventos JavaScript, podemos controlar un sinfín de actividades que realizan los usuarios: desde el uso del mouse (botones, rueda y movimientos) hasta el ingreso desde un teclado, la ejecución de animaciones en la pantalla y la acción del usuario sobre el navegador web, entre muchas otras opciones más. Veamos a continuación todas las posibilidades que nos ofrecen los eventos que pueden controlarse desde JS.

Cada uno de los eventos cuenta con propiedades y métodos adicionales, que permiten validar la transacción del evento, su cancelación por parte del usuario, o por una falla de conexión o de software, entre otras cosas. Veamos a continuación cómo sacar provecho de los eventos más utilizados en los sitios web, conocidos como **eventos HTML**.

¿QUÉ QUIERO CONTROLAR?	¿CÓMO?	¿POR MEDIO DE QUÉ EVENTOS?
Mouse	Controla los diferentes eventos del mouse.	<code>onclick()</code> , <code>oncontextmenu()</code> , <code>onmousedown()</code> , <code>onmousemove()</code> , etcétera.
Keyboard	Controla las teclas presionadas en el teclado.	<code>onkeydown()</code> , <code>onkeypress()</code> , <code>onkeyup()</code>
Frame	Controla eventos de carga y descarga de la página.	<code>onload()</code> , <code>onresize()</code> , <code>onscroll()</code> , <code>onunload()</code> , etcétera.
Form	Controla y valida los eventos que ocurren sobre un elemento <code><form></code> .	<code>onblur()</code> , <code>onchange()</code> , <code>onfocus()</code> , <code>oninput()</code> , etcétera.
Drag	Administra el contenido que es arrastrado hacia el navegador web.	<code>ondrag()</code> , <code>ondragend()</code> , <code>ondragleave()</code> , <code>ondrop()</code> , etcétera.
Clipboard	Eventos de copiado y pegado entre el portapapeles y el navegador web.	<code>oncopy()</code> , <code>oncut()</code> , <code>onpaste()</code>
Print	Maneja los eventos de impresión.	<code>onafterprint()</code> , <code>onbeforeprint()</code>
Media	Controla todos los eventos multimedia.	<code>onabort()</code> , <code>oncanplay()</code> , <code>onplaying()</code> , <code>onseeking()</code> , etcétera.
Animation	Controla el proceso de animación de CSS.	<code>animationend()</code> , <code>animationiteration()</code> , <code>animationstart()</code>
Transition	Valida las transiciones entre páginas.	<code>transitionend()</code>
Misc	Controla eventos como almacenamiento local, conexión y desconexión de red.	<code>onerror()</code> , <code>onmessage()</code> , <code>onopen()</code>
Server-sent	Atiende los eventos enviados a un servidor remoto.	<code>onwheel()</code> , <code>ononline()</code> , <code>onstorage()</code> , <code>onoffline()</code> , etcétera.
Touch	Controla los eventos de pantallas táctiles.	<code>ontouchcancel()</code> , <code>ontouchend()</code> , <code>ontouchmove()</code> , <code>ontouchstart()</code>



onLoad()

El evento **onLoad()** se utiliza para controlar cuando un determinado objeto de una página o la misma página se ha cargado. Con este evento, se puede ejecutar una alerta para el usuario o cambiar el cursor del mouse. También se suele utilizar para escribir cookies en el navegador web del usuario, detectar el tipo de SO y navegador, y redireccionar hacia otra página en el caso que sea necesario, entre otras tantas funciones. Su sintaxis, como podemos ver en la Imagen 11, es simple.



Imagen 11.
Función JS y evento
onload() que validan si un
navegador web soporta
cookies o no, y le comunican
esto al usuario, según la
necesidad del sitio.

Finalmente, una vez que tenemos la función JS creada, solo resta aplicarla en el elemento HTML que deseamos controlar. Los elementos que soportan el evento `onload()` son: `<body>`, `<frame>`, ``, `<link>`, `<script>`, `<style>`, e `<input type="image">`. El resultado del script está representado en la Imagen 12.

Imagen 12.
A través de `<body onload="miScript()">`, controlamos si el navegador del usuario soporta o no cookies, y sobre esta base lanzamos un tipo de alerta determinado.

OnChange()

Este evento es utilizado comúnmente en la mayoría de los input types que componen un formulario web. **onChange()** detecta cuando el valor de un elemento cambia y entonces ejecuta una acción determinada. Es utilizado usualmente para detectar el cambio de estado de un elemento de formulario, como el contenido de una caja de texto. Creemos un nuevo documento HTML con la estructura básica e incluyamos, dentro del elemento body, el siguiente código:

```
<h1>Trabajamos con el evento onChange()</h1>
<br>
<p>Modifique el texto del siguiente
campo, y presione la tecla TAB:</p>
<br>
Ingrese otro texto: <input type="text"
name="txt" value="Fullstack Developer"
onchange="textoCambio(this.value)">
```

Ahora agregamos un script en el mismo documento con el siguiente código:

```
function textoCambio(val) {
    alert("Ha cambiado el valor del
        texto por: [" + (val) + "]");
}
```

Ejecutemos este ejercicio para ver su resultado.

Imagen 13.
Luego de cambiar el texto predeterminado en el textbox, presionamos la tecla TAB y se acciona el script que nos indica, con un mensaje, el nuevo valor del contenido.

OnMouseOver y onMouseOut

Usualmente estos eventos trabajan juntos.

OnMouseOver() se ejecuta cuando el cursor del mouse se posiciona sobre un elemento, y **onMouseOut()** es invocado cuando el cursor sale del foco de dicho elemento. Con estos eventos se ejecuta la visualización de una imagen en un tamaño mayor, o se visualiza y oculta otro contenido HTML puntual.

Con más de una decena de **eventos OnMouse()** disponibles, podremos tener el control total de lo que el usuario hace con el cursor y cómo interactúa con nuestras páginas web.

<EJERCICIO PRÁCTICO>

Redimensionar una imagen



A continuación, realizaremos un ejercicio práctico utilizando los eventos explicados antes. Este ejercicio debe contar con las siguientes características:

- ☒ Desarrollar una página web que muestre una imagen mediana al 20% de su tamaño original.
- ☒ Esta imagen deberá tener dos funciones JavaScript, una en el evento `onMouseOver()` y otra para el evento `onMouseOut()`.
- ☒ Los eventos mencionados deben recibir un parámetro, que será el objeto `imagen`.

- ☒ Cuando posicionemos el cursor del mouse sobre la imagen, se ejecutará el primer evento, mientras que al retirar el cursor, se ejecutará el segundo evento
- ☒ El evento correspondiente a `onMouseOver` debe redimensionar la imagen al 50 o 100 % de su valor.
- ☒ El evento correspondiente a `onMouseOut` devuelve su imagen al estado original.

El resultado debe ser similar al de la Imagen 14.



OnMouseDown, onMouseUp, onMouseLeave

Otros eventos del mouse de suma importancia dentro de JavaScript, son **onMouseDown()**, **onMouseUp()** y **onMouseLeave()**. Estos permiten manipular cada clic con mayor precisión que un simple evento **onClick()** y especificar, en cada uno de los procesos de un clic, que ocurrirá un determinado evento. Veamos un ejemplo a continuación. Creamos tres imágenes diferentes, de un tamaño normal. Una se llamará gris.png; otra azul.png, y la última, naranja.png. Ahora creamos un documento HTML con un elemento img (cuyo id será 'imagen'), donde visualizamos la imagen gris.png. Por último, creamos las siguientes tres funciones JavaScript:

```
<script>
    function mostrarAzul() {
        document.getElementById('imagen').
            src = 'azul.png';
    }

```

```
function mostrarNaranja() {
    document.getElementById('imagen').
        src = 'naranja.png';
}
function mostrarGris() {
    document.getElementById('imagen').
        src = 'gris.png';
}
</script>
```

Finalmente agregamos, en el documento HTML, los eventos JavaScript correspondientes, y el código HTML quedará de la siguiente forma:

```

```

Ejecutemos el proyecto y comprobemos su comportamiento, como en la Imagen 15.

Al bajar el botón predeterminado del mouse sobre la imagen, esta cambia a color azul. Al soltar el botón del mouse presionado, la misma cambia a naranja, y al retirar el cursor sobre la imagen, esta vuelve a su estado inicial: gris.



Event Listener

Otra función interesante que podemos combinar entre HTML y JavaScript es el manejo de eventos mediante **Event Listener**.

Esto nos ofrece dos manejadores específicos: **addEventListener** y **removeEventListener**. Como sus nombres lo indican, sirven para adjuntar y remover funcionalidades específicas de las páginas web, y así poder ejecutar otras funciones determinadas.

AddEventListener()

Con el método **addEventListener()** es posible adjuntar una función específica a un elemento determinado de un documento web, aunque este elemento tenga otras funciones especificadas previamente. Así podremos controlar en detalle cualquier evento que ocurra sobre una página web. También es aplicable a cualquier elemento que la página contenga.

Su sintaxis es:

```
<script>
    document.addEventListener("focus", tieneFoco);
    function tieneFoco() {
        //Aquí especificamos código que maneje el evento focus de nuestra página web
    }
</script>
```

addEventListener requiere de **dos parámetros** para funcionar: por un lado, **el evento que debe escuchar**; por otro, **la función que se debe ejecutar cuando dicho evento es escuchado**.

Como podemos ver en el script de abajo, el objeto **document** es quien tiene como método **addEventListener**. Al invocar este método, le pasamos dos parámetros al mismo.

El primero de ellos hace referencia a “qué evento debe escuchar este método”. El segundo parámetro es la función que se debe ejecutar, cuando el evento escuchado es accionado.

La función a ejecutar puede ser una función simple que haga determinada tarea, una función que recibe parámetros, o una función que retorne un valor. Por supuesto, debemos tener cuidado al ejecutar una función con parámetros, que dichos parámetros contengan valores determinados, para que la función se ejecute de manera correcta.

Si no prevemos esto, seguramente JavaScript dará un error, y nuestro objetivo de que la página web realice una determinada tarea o instaure un comportamiento determinado, quedará truncado.

RemoveEventListener()

Este método permite remover cualquier **addEventListener** asignado a un determinado elemento de una página. Su sintaxis es la siguiente:

```
document.removeEventListener("focus",
    tieneFoco);
```



CONTROLAR CON ADDEVENTLISTENER()

Pausar audio y video, cuando el usuario minimiza el navegador, cambia de pestaña, o de aplicación, es uno de los usos más populares de esta función.

Cómo utilizar RemoveEventListener



Veamos a continuación la manera de implementar correctamente los scripts AddEventListener() y RemoveEventListener().

- 1 Creamos una página HTML con su estructura básica, agregamos un elemento `<p>` cuyo ID sea "parrafo", y por último escribimos el siguiente script:

```
<script>
    window.addEventListener("resize", tamano);
    function tamano() {
        w = window.innerWidth;
        h = window.innerHeight;
        document.getElementById("parrafo").innerHTML = "Ancho de la ventana: " + w +
            "<br> Alto de la ventana: " + h ;
    };
<script>
```

- 2 Ejecutemos nuestro ejercicio y redimensionemos la página tanto en ancho como en alto.



Al redimensionar la página, JavaScript escucha el evento `resize`, y muestra en pantalla el alto y el ancho del documento HTML.

- 3 Ahora agregamos la siguiente función JavaScript:

```
function quitarEventListener() {
    window.removeEventListener("resize", tamano);
    return;
}
```

Y el siguiente botón dentro del elemento `<body>`:

```
<button onclick="quitarEventListener()">Quitar Event Listener</button>
```

- 4 Ejecutemos nuevamente nuestra página y, antes de redimensionar la ventana del navegador, presionemos el botón. A continuación, redimensionemos la ventana del navegador y veamos qué ocurre.

Cuadros de diálogo

Los cuadros de diálogo en JavaScript permiten tanto comunicar determinados mensajes a los usuarios que navegan nuestra página web como también interactuar con ellos cuando deben ingresar contenido a dicha página. Veamos a continuación qué tipos de cuadros de diálogo nos brinda JavaScript.

Alert box

El cuadro de diálogo Alert es el que ya estuvimos trabajando en algunos ejemplos JS. Este se utiliza para mostrarle información al usuario, tal como ocurre con los cuadros de diálogo clásicos de cualquier sistema operativo. Finalmente, el usuario presionará el botón OK o Aceptar, para poder continuar operando en el sitio web.

```
alert('Mensaje que verá el usuario en pantalla');
```

Confirm Box

Similar a Alert Box, Confirm box sirve para interactuar con el usuario, aunque a través de este cuadro de

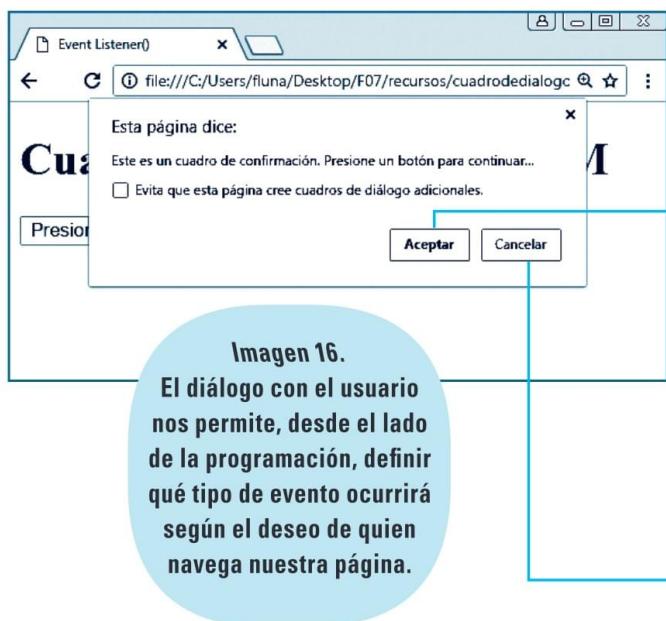
diálogo, le pedimos una confirmación. Este cuadro de diálogo mostrará dos botones OK y Cancel; el usuario deberá presionar uno de los dos para continuar. Su código es simple:

```
Var c = confirm("Presione un botón para continuar.");
```

Como necesitaremos validar qué botón presionó el usuario, creamos una variable que recibirá un valor booleano con el botón que el usuario presionó. Veamos esto a través del próximo ejercicio. Creamos un nuevo documento HTML, con un elemento button y un elemento <p>.

Creamos a continuación una función JavaScript con el siguiente código:

```
function dialogo() {  
var c = confirm("Este es un cuadro de confirmación.  
Presione un botón para continuar...");  
if (c == true) {  
    texto = "Ha presionado el botón OK";  
}
```



```

} else {
    texto = "Ha presionado el botón
    CANCEL";
}
document.getElementById("parrafo").
innerHTML = texto;
}

```

Y en elemento button agregamos el evento onclick() con la llamada a la función dialogo().

Al presionar el botón se ejecutará el cuadro de diálogo Confirm, y cuando presionemos uno de sus botones, el elemento <p> de la página nos dirá qué botón hemos presionado.

Prompt()

Otro cuadro de diálogo, denominado **prompt()**, nos permite interactuar con el usuario, y deja que este ingrese un texto determinado, que condicionará la siguiente acción en la página web.

Por ejemplo, podemos preguntarle al usuario su nombre o nickname y, cuando él lo ingresa, lo saludamos grabando su nombre en la página. Si combinamos este cuadro de diálogo con las cookies, podemos hacer que, a partir de ahora en más, cuando el usuario ingresa a nuestra página, sea saludado directamente sin volver a preguntarle su nombre.

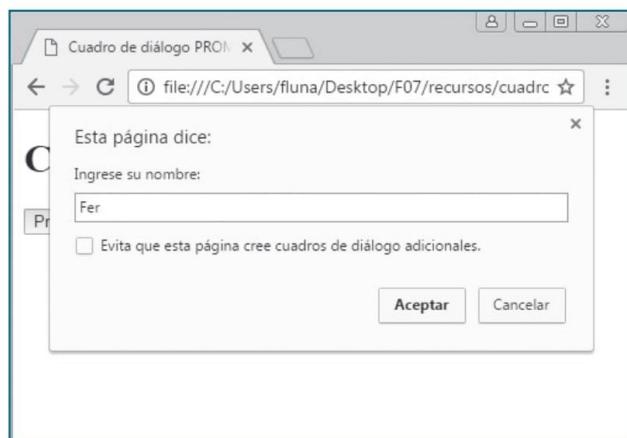
```

var c = prompt("Ingrese su nombre: ", "");
texto = "Bienvenido <b>" + c + "</b>";

```

Tal como lo hicimos con Confirm Box, creamos una variable donde se almacene el texto que ingresa el usuario, y luego aplicamos dicho contenido de acuerdo a nuestra necesidad.

A pesar de sus múltiples funcionalidades y aptitudes para controlar interacciones, uno de los puntos en contra que tiene JavaScript nativo es la imposibilidad de personalizar los cuadros de diálogo con títulos, imágenes y botones de opciones.



LINE BREAKS

Puede que en muchos casos debamos ingresar varias líneas de texto en los cuadros de diálogo JavaScript, por lo tanto, para hacer un trabajo prolífico, necesitaremos generar un salto de línea entre los párrafos. Para realizar esto, dentro del mismo texto que acompaña los cuadros de diálogo, intercalaremos la combinación '\n'. De esta manera, creamos un salto de renglón sobre el texto del cuadro de diálogo.

Objeto location

Cuando debemos tratar diferentes parámetros relacionados con una web en particular, podemos aprovechar las propiedades y métodos que nos presenta el **objeto Location**.

En el siguiente cuadro, se detallan sus principales propiedades.

A través de la propiedad href, podemos hacer que el navegador del usuario navegue hacia una nueva URL cuando presione un link o un botón. Es el equivalente a <a href> de HTML. El código para esta acción es:

```
location.href = www.redusers.com;
```

PROPIEDAD	DESCRIPCIÓN
Hostname	Devuelve el nombre del servidor web. Por ejemplo: www.misitio.com .
Pathname	Devuelve el path completo del documento HTML actual. Por ejemplo: /html/contenido/mipagina.html.
Protocol	Devuelve el protocolo utilizado por la página web. Por ejemplo: http o https , según el tipo de sitio web que tengamos implementado.
Href	Devuelve o establece una URL completa de una página web. Por ejemplo: www.misitio.com/mipaginaweb.html
Hash	Devuelve el hipervínculo interno de una página web, especificado en su URL.



Imagen 18. Las propiedades del objeto Location nos permiten, desde JavaScript, conocer datos de la página así como también forzarlos.

 **NUEVA VENTANA**

location.href nos brinda la posibilidad de redireccionar hacia una web o URL diferente, abriéndola en una nueva ventana. Si necesitáramos abrir una URL en una nueva ventana o pestaña en el navegador, utilizaremos el método open del objeto window, que aprenderemos en las tres siguientes páginas.

<EJERCICIO PRÁCTICO> Redireccionar una página web



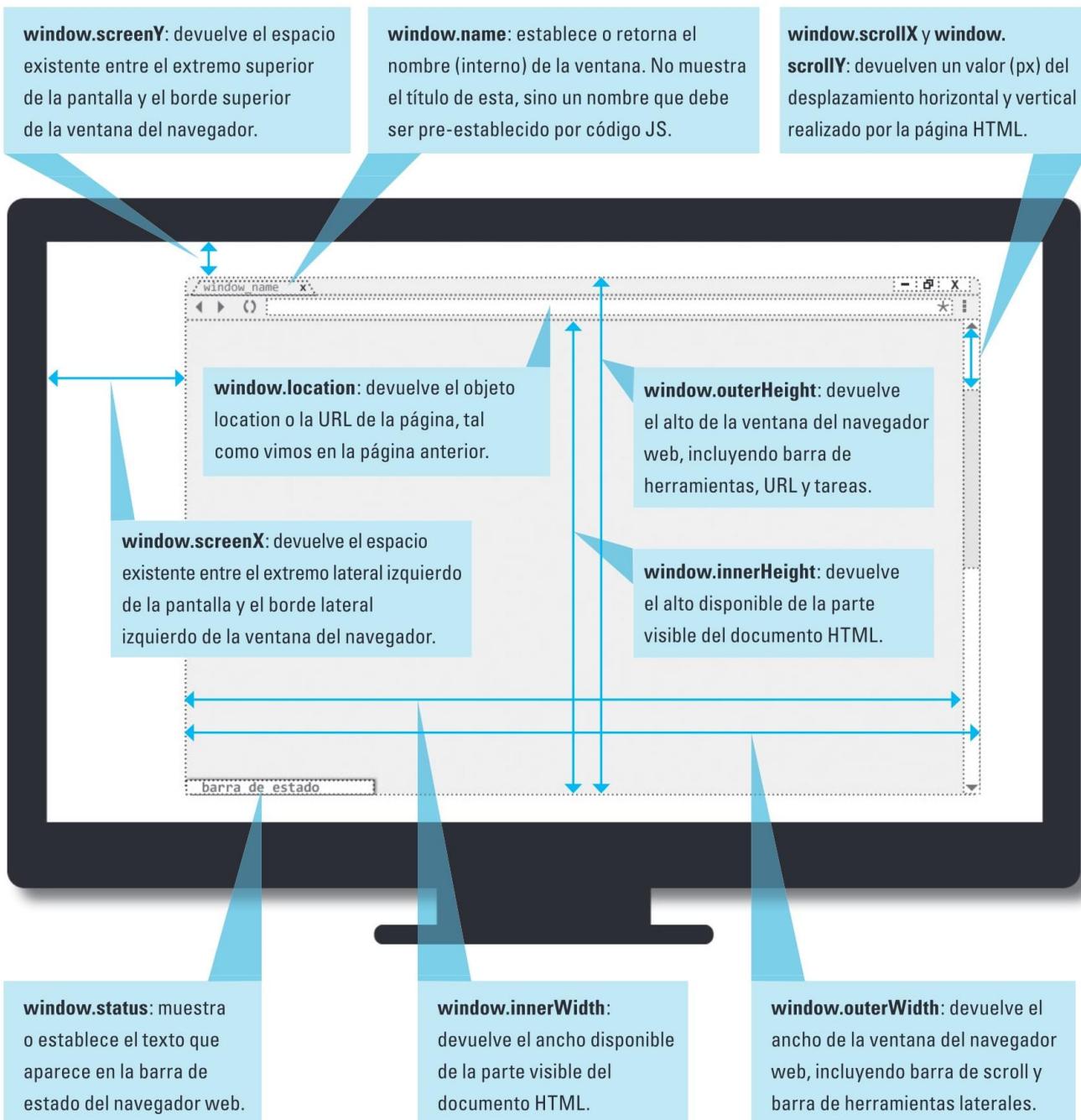
Agregaremos dos imágenes de manera que, al hacer clic sobre estas, nos redireccionen a un sitio web determinado, a través de la propiedad href. Para no crear código de más generando una función Javascript por cada sitio web, aprovecharemos lo aprendido sobre las funciones parametrizadas en JS. De esta manera, creamos una única función, que se invocará desde cada imagen, pasándole como parámetro la URL hacia donde redireccionamos al usuario.

Objeto window

El **objeto window** en JS brinda información acerca de la ventana donde se está ejecutando nuestra web; puede prestar información respecto de la computadora o del dispositivo móvil en general. Cuenta con propiedades, métodos y con otros objetos anidados dentro de window.

Propiedades

Veamos a continuación un gráfico explicativo sobre las propiedades más importantes que el objeto window contiene. En las dos siguientes páginas, veremos cómo ponerlo en práctica.



Métodos

Los métodos nos ayudan a interactuar con la ventana y con algunas funciones propias del navegador web.

Veamos a continuación una lista con los métodos más utilizados en el objeto window.

MÉTODO	DESCRIPCIÓN
<code>open()</code>	Abre una ventana nueva con determinadas características o parámetros establecidos.
<code>close()</code>	Cierra la ventana actual o alguna ventana abierta, parametrizando para este último caso, un nombre interno preestablecido.
<code>print()</code>	Llama al método de impresión del navegador.
<code>resizeBy()</code>	Abre una nueva ventana, y ajusta el tamaño de esta en el alto y el ancho pasados como parámetro a este método.
<code>resizeTo()</code>	Abre una nueva ventana e, independientemente del tamaño inicial establecido para esta, la redimensiona hacia un nuevo alto y ancho establecido.
<code>focus()</code>	Envía el foco hacia una determinada ventana abierta. Esta debe tener un nombre previamente asignado, para poder identificarla con facilidad desde el código.

Imprimir páginas desde JS

Veamos ahora la manera más óptima de sacar provecho del método print() de JS. Armemos para ello un nuevo ejercicio. Este consta de un documento HTML con su estructura estándar, al cual le agregamos los siguientes elementos, respetando el orden de mención: <button>, <h1>, , <p>. La estructura del documento debe quedar como en la Imagen 19.

Creamos a continuación una función JS llamada `imprimir()`, y establecemos en esta el método correspondiente que invoca al sistema de impresión del navegador web.



```
function imprimir() {  
    window.print();  
}
```

Para evitar visualizar el elemento button y otros elementos que no deseamos que aparezcan en la página por imprimir, debemos modificar la función JS para que quede de la siguiente manera:

```
var boton = document.getElementById("btn");
boton.style.visibility = 'hidden';
window.print();
boton.style.visibility = 'visible';
```

Este cambio agrega una variable botón, en la que capturaremos el elemento cuyo ID es 'btn'. A continuación establecemos que su propiedad visibility sea oculta. Allí invocamos el método print() y, al finalizar este método (se imprima o no la página), cambiamos su propiedad visibility a visible.

Si nuestra página HTML contiene pocos elementos y queremos ofrecer una forma amigable de impresión, con ocultar previamente los elementos que no deseamos que salgan, solucionaremos nuestro problema. Si los elementos por ocultar son demasiados, es conveniente invocar una página HTML nueva, donde solo mostremos el contenido que el usuario debe imprimir.

<EJERCICIO PRÁCTICO>

Control de pestañas y ventanas con JS



Crearemos dos documentos HTML

con su estructura básica:

ejercicio03.html y

ventanaemergente.html.

El ejercicio presentará una página HTML simple (ejercicio03.html), con un elemento button que, al ser presionado, abrirá una ventana en modo popup (ventanaemergente.html) con determinadas características. En este último documento HTML, incluiremos un elemento button que invocará a una función JS para cerrar automáticamente la ventana. El código JS del documento ejercicio03.html para la función de apertura, es el siguiente:

```
<script>
    function abrirVentana() {
        window.open("ventanaemergente.html", "_blank", "toolbar=no,scrollbars=auto,
        resizable=no,top=100,left=100,width=400,height=400");
    }
</script>
```

Luego nos queda invocar la función abrirVentana() desde el atributo onclick del elemento button. Por último, resta el script JS que va dentro del documento ventanaemergente.html, que será invocado desde el atributo onclick de su elemento button.

```
<script>
    function cerrarVentana() {
        window.close();
    }
</script>
```

Al invocar al método window.open(), podemos ver en el script correspondiente, que se le

Apertura y cierre de ventanas

Presione el botón 'Abrir'

Abrir

Imagen 20.
Con las funciones
correctamente
declaradas, ya podemos
comenzar a abrir y cerrar
las ventanas definidas.



Presione el botón a continuación, para cerrar
esta ventana.

Cerrarme

parametrizan una serie
de propiedades.

La primera de ellas es el archivo
HTML que debe abrir. Si no
especificamos un archivo válido, la
función se ejecutará igual abriendo
una ventana en blanco about:blank.

```
window.open
    ("ventanaemergente.html",...
```

El segundo parámetro indica que
este archivo se abrirá en una ventana
independiente a la actual.

```
..., "_blank",...
```

Por último, en el tercer parámetro
enviamos una serie de propiedades,
separadas por coma (,), donde
indicamos que no visualice la barra
de herramientas, que las barras de
desplazamiento sean automáticas,
que no se pueda redimensionar la
ventana, y que se abra a 100 pixeles
del borde superior y 100 pixeles del
borde inferior de la pantalla. En el
último parámetro indicamos el alto
y el ancho de la ventana.

```
... "toolbar=no,scrollbars=
    =auto,resizable=no,top=
    100,left=100,width=400,
    height=150");
```

Objeto history

Si miramos con detenimiento todo el compendio de propiedades que conforman el objeto JS window, encontraremos a **history** como una de sus propiedades. History a su vez es un objeto JS que puede manejarse de forma independiente a window. Este posee una sola propiedad (`length`) y tres métodos (`back`, `forward`, `go`).

history.length

A través de la propiedad **length**, podremos conocer desde JS cuál es el número de páginas o direcciones web navegadas en la pestaña del navegador web. Para capturar esta información, debemos agregar la siguiente línea de código a una función JS:

```
//conocer el total de páginas navegadas.  
var t = history.length;  
alert('total de historial generado: ' + t  
+ ' páginas.');
```

back() y forward()

Estos métodos nos permiten desplazarnos hacia adelante y hacia atrás en la navegación de una página web.

Por supuesto que, para que los métodos aquí mencionados funcionen correctamente, primero debemos generar un historial de navegación dentro de la misma pestaña donde nos encontramos. Estos métodos no devuelven ningún error en el caso de no haber generado historial.

Aun así, si existe código o procesos JS después de la invocación a uno de estos métodos, siempre conviene validar con la propiedad `history.length`, si contamos con historial generado para navegar a través de estos métodos.

```
function hagoAlgo() {  
    var t = history.length;  
    if (t > 0) {  
        //invoco a history.back() o history.  
        forward()  
        //y el resto del código  
    } else {  
        //salgo de la función sin hacer nada  
        return false;  
    }  
}
```



Imagen 21.
Los métodos `history.back()` y `history.forward()` son equivalentes a los botones adelante y atrás del web browser.



ÍNDICE BASE DE HISTORY.LENGTH

Esta propiedad requiere un cuidado especial, dependiendo del navegador web que esté ejecutando nuestra página web. El punto de partida del total de páginas devuelto por `history.length` inicia en 0 cuando el navegador web es Opera, IE o Edge, mientras que con Firefox, Safari y Chrome este índice comienza en 1. Para tener sumo cuidado con esto, debemos verificar con qué navegador web se cuenta, usando `navigator.userAgent`.

Objeto navigator

Hoy en día contamos con una diversidad de dispositivos (móviles, portátiles y de escritorio), que contienen un navegador web para navegar por internet. Y junto con estos dispositivos nos encontramos con una variedad importante de pantallas, sistemas operativos, y demás características que engloban a cada uno de ellos, lo que hace que debamos tener sumo cuidado con el contenido web que vamos a visualizar. Este contenido puede ser efectivo en determinados navegadores, mientras que en otros puede no resultar cómoda su lectura o la disposición de los elementos HTML en pantalla. Por suerte, las propiedades del **objeto navigator**, nos facilitarán la tarea de programar una web efectiva, cuyo contenido se adapte de manera óptima al dispositivo y a la pantalla que están cargando nuestra web. En la tabla inferior podemos ver un detalle de las principales propiedades de navigator.

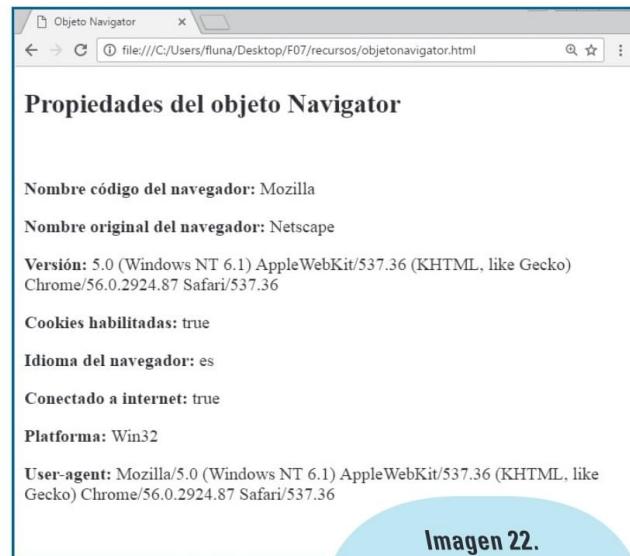


Imagen 22.

En el contenido exclusivo para nuestros suscriptores, se encuentra un ejemplo en el archivo objetonavigator.html, que visualiza todas las propiedades aquí listadas.



Imagen 23.
Detectar el User-Agent de manera efectiva nos permitirá cargar el CSS o adaptar el contenido web para el dispositivo.

PROPIEDAD	DESCRIPCIÓN
appCodeName	Devuelve el nombre código del navegador web.
appName	Devuelve el nombre del navegador web.
appVersion	Devuelve la versión de compilación del navegador web.
cookieEnabled	Indica si el navegador web posee habilitada la grabación y consulta de cookies.
language	Devuelve el idioma del navegador web.
onLine	Determina si el navegador web está o no conectado a internet.
userAgent	Devuelve una cadena con un resumen de las principales características del navegador web, motor, y sistema operativo donde se está ejecutando.

Propiedad userAgent

De todas las propiedades del objeto navigator, userAgent es la que nos brinda, en un único string, toda la información del SO, el motor de render, la versión y el navegador web. Veamos a continuación ejemplos de string que recibe userAgent, según el navegador web o el sistema operativo.

Ya sea desde JavaScript o desde CSS, detectar el navegador y el sistema operativo del usuario para poder adaptar nuestro contenido web, es una tarea fundamental en estos tiempos.

Internet Explorer para Windows



```
User-agent header sent: Mozilla/5.0 (Windows NT 6.1; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E; Tablet PC 2.0; rv:11.0) like Gecko
```

Microsoft Edge para Windows



```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393
```

Google Chrome para Windows



```
User-agent header: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36
```

Mozilla Firefox para Windows



```
User-agent header: Mozilla/5.0 (Windows NT 6.1; rv:51.0) Gecko/20100101 Firefox/51.0
```

Google Chrome para Android



```
User-agent header sent: Mozilla/5.0 (Linux;Android 7.1.1;Nexus 6P Build/N4F260) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.02924.87 Mobile Safari/537.36
```

Safari para iPad



```
Mozilla/5.0 (iPad; CPU OS 9 _ 3 _ 5 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13G36 Safari/601.1
```

Como vemos en estos ejemplos, cada string devuelto dentro de la propiedad userAgent difiere mucho de los otros.

Por lo tanto, a simple vista se nos complicará identificar, de forma fácil y directa, qué navegador web, motor de render, o sistema operativo cargó nuestra página web.

Veamos a continuación una función de ejemplo, que analiza el tipo de sistema operativo donde se carga una página HTML, y la redirecciona según el SO:

```
function detectoSO() {  
    var osName = "Desconocido";  
    if (navigator.userAgent.indexOf("Win")!=-1)  
        osName = "Windows";  
    if (navigator.userAgent.indexOf("Android")!=-1) osName = "Android";  
    if (navigator.userAgent.indexOf("iOS iPad iPhone iPod")!=-1) osName = "iOS";  
    if (osName == "Windows") {  
        //window.location = "index.html  
        //para Windows"  
    }  
}
```

```

    }
    else if (osName == "Android") {
        //window.location = "index.html
        para Android"
    }
    else if (osName == "iOS") {
        //window.location = "index.html
        para iOS"
    }
    // else if {para otros sistemas operativos}
    else {
        alert('No tenemos una web adaptada
              a su sistema operativo.');
        return false;
    }
}

```

En este pequeño ejemplo JS, construimos una función que analiza la cadena de texto de navigator.userAgent a través de indexOf, y consulta si dentro de la cadena existe un determinado string: Win, iOS, Android,



DETECCIÓN SIMPLIFICADA

En las épocas que corren, la detección de la plataforma que carga nuestra web se simplifica a mobile y desktop. Muchas web y soluciones basadas en librerías se ocupan de detectar el tipo de sistema operativo y la resolución de pantalla. Si el sistema operativo es Android o iOS, entonces redireccionan al usuario hacia una web móvil. Si el SO es Windows, Mac OS, Linux o iPad, redireccionan al usuario a una versión de escritorio. Existen decenas de opciones de detección de dispositivo, solo debemos elegir la que más nos convenga sobre la base del alcance de nuestro proyecto.

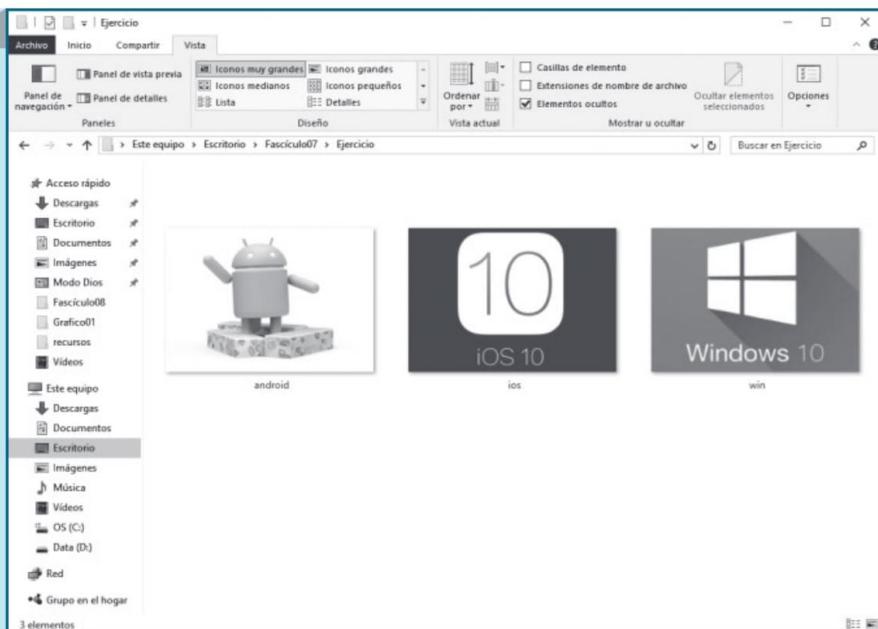
etcétera. Luego, a través de un if–else if, utilizamos window.location para redireccionar a la página HTML adaptada para el sistema operativo detectado. Veamos el comportamiento de esta función en un ejemplo real mediante el siguiente paso a paso.

Nuestro redireccionador JS



Construimos a continuación nuestro propio redireccionador web JS, según el sistema operativo que cargue nuestro navegador web.

1 Conseguimos algunas imágenes relacionadas a sistemas operativos móviles o de escritorio, en formato PNG, JPG o GIF. A continuación, normalizamos el tamaño de estas, llevándolas a una misma medida. Ejemplo: 600 x 350 pixeles.



2 Armamos la estructura básica de una página

HTML. Agregamos dentro de <header> la función JS descripta antes y, para que nos funcione, en lugar de window.location utilizamos document.getElementById('imagen').src = "win.jpg"; Esto hará que se muestre una imagen según el SO detectado.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Trabajando con getElementById</title>
5      <link rel="stylesheet" href="ejercicio01.css" type="text/css" />
6      <script>
7        function detectoSO() {
8          var osName = "Desconocido";
9          if (navigator.userAgent.indexOf("Win")!=-1) osName = "Windows";
10         if (navigator.userAgent.indexOf("Android")!=-1) osName = "Android";
11         if (navigator.userAgent.indexOf("iOS iPad iPhone iPod")!=-1) osName = "iOS";
12         //if (navigator.userAgent.indexOf("Linux")!=-1) osName = "Linux";
13         //if (navigator.userAgent.indexOf("Macintosh")!=-1) osName = "MacOS";
14         //... seguir agregando otros sistemas operativos
15
16         document.getElementById("useragent").innerHTML = navigator.userAgent;
17
18         if (osName == "Windows") {
19             //window.location = "index.html para Windows"
20             document.getElementById('imagen').src = "win.jpg";
21         }
22         else if (osName == "Android") {
23             //window.location = "index.html para Android"
24             document.getElementById('imagen').src = "android.jpg";
25         }
26         else if (osName == "iOS") {
27             //window.location = "index.html para iOS"
28             document.getElementById('imagen').src = "ios.png";
29       }
30     }
31   </script>
32 </head>
33 <body>
34   <h1>Detectando el sistema operativo</h1>
35   
36   <p>Windows 10</p>
37 </body>
38 </html>
```

3 Ejecutamos nuestra página HTML en el dispositivo de escritorio, llamando la función desde <body onload>, y verificamos que la imagen visualizada sea correlativa al sistema operativo. Si disponemos de un servidor web personal, como Nginx o similar, copiamos el contenido HTML a este.



4 Si nuestro código fue estructurado de manera correcta, al probar este ejercicio en un dispositivo móvil o una tablet, debemos visualizar la imagen correspondiente a dicho dispositivo en la página HTML.



5 En algunos dispositivos móviles Apple, podemos encontrarnos con el problema de no detectar de manera correcta la plataforma. Para solucionar esto, consultaremos sitios como www.whatismybrowser.com, para buscar una posible solución.



6 Una posible solución a dicho problema es establecer, dentro de la cadena de búsqueda, diferentes opciones de dispositivos móviles Apple. En nuestro ejemplo utilizamos 'iOS iPad iPod iPhone', para aumentar las chances de detección.

```
8           var osName = "Desconocido";
9           if (navigator.userAgent.indexOf("Win")!=-1) osName = "Windows";
10          if (navigator.userAgent.indexOf("Android")!=-1) osName = "Android";
11          if (navigator.userAgent.indexOf("iOS iPad iPhone iPod")!=-1) osName = "iOS";
12          //if (navigator.userAgent.indexOf("Linux")!=-1) osName = "Linux";
13          //if (navigator.userAgent.indexOf("Macintosh")!=-1) osName = "MacOS";
14          //... seguir agregando otros sistemas operativos
```



EN LA PRÓXIMA CLASE...

USERS

CURSO VISUAL Y PRÁCTICO

8

Argentina \$ 49

PROGRAMADOR WEB Full Stack

Desarrollo frontend y backend

Formularios web

Elemento form \ Input types y atributos \ Envío de datos \ Integración con JS

```
Trabajo con Formularios HTML</h2>
<label for="nombre">Nombre:</label><br>
<input type="text" name="nombre" id="nombre" placeholder="Ej: Juan" />
<label for="apellido">Apellido:</label><br>
<input type="text" name="apellido" id="apellido" placeholder="Ej: Pérez" />
<label for="edad">Edad:</label><br>
<input type="number" name="edad" id="edad" placeholder="Ej: 25" />
<input type="submit" name="submit" value="Enviar" />
<input type="reset" name="cancel" value="Cancelar" />
</form>
```



MÉTODOS
POST Y GET
PASO A PASO

Domina las últimas tecnologías para sitios web responsive y dinámicos



PROGRAMADOR WEB Full Stack

Desarrollo frontend y backend

¿Por qué aprender PROGRAMACIÓN WEB FULL STACK?

- Porque **desde cero**, y sin ningún conocimiento previo, este curso te enseña a diseñar un simple sitio que luego transformaremos en uno **dinámico, interactivo y responsivo**, conociendo y aprovechando las últimas tecnologías de desarrollo.
- Porque a lo largo de 24 fascículos, repletos de ejemplos, ejercicios y explicaciones visuales, aprenderás tanto los lenguajes y tecnologías frontend como backend: HTML, CSS, JavaScript, PHP, MySQL, JQuery y más. ¡Son varios cursos en uno solo!
- Porque actualmente el perfil **FULL STACK** es uno de los más demandados en el **mercado laboral**. Te enseñaremos a darles vida a tus propios proyectos, generar un portfolio y a dar tus primeros pasos laborales como desarrollador.



Explicaciones
paso a paso



Guías
visuales



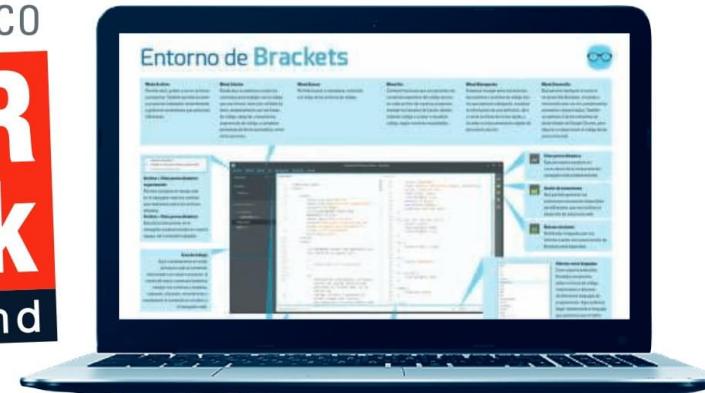
Ejercicios prácticos
e integradores



Servicio de profesores en línea
profesores@redusers.com

Clases anteriores

Puede adquirirlas escribiendo a
usershop@redusers.com



PLAN DE LA OBRA

- 01 Ecosistema web
- 02 HTML5
- 03 CSS3
- 04 Diseño UI con CSS
- 05 Introducción a JavaScript
- 06 JavaScript orientado a objetos
- 07 Integración de HTML5 y JavaScript
- 08 Formularios web
- 09 Multimedia y APIs
- 10 CSS Avanzado
- 11 Diseño web responsive
- 12 Sitios multiplataforma con Bootstrap
- 13 PHP
- 14 MySQL
- 15 PHP y MySQL
- 16 Webs dinámicas con Ajax y PHP
- 17 Buenas prácticas:
análisis, tests y optimización
- 18 Fundamentos del ecosistema mobile
- 19 Jquerymobile: la web móvil
- 20 Funcionalidades extendidas en mobile web
- 21 Potenciando la faceta full stack
- 22 Webapps y plataformas amigables
- 23 Versionando el desarrollo: GIT y Github
- 24 Salida laboral

