APUNTES HTML5: JavaScript

TOMO I:

Manejo de componentes HTML

José Juan Urrutia Milán

Reseñas

- Curso HTML5: https://www.youtube.com/playlist?list=PLU8oAlHdN5BnX63lyAeV0LzLnpGudgRrK
- Curso JavaScript: https://www.youtube.com/playlist?list=PLU8oAlHdN5BmpobVmj1IlneKlVLJ84TID
- w3c (página que decide los estándares CSS/HTML...): www.w3c.org

Siglas/Vocabulario

- **IDE:** Entorno de Desarrollo Integrado.
- **OS:** Operator System (Sistema Operativo).
- **BBDD:** Bases de Datos.

Leyenda

Cualquier abreviatura o referencia será subrayada.
Cualquier ejemplo será escrito en **negrita.**
Cualquier palabra de la que se pueda prescindir irá escrita en cursiva.
Cualquier abreviatura viene explicada a continuación:

Abreviaturas/Referencias:

- 123: Hace referencia a cualquier número.
- nombre: Hace referencia a cualquier palabra/cadena de caracteres.
- a: Hace referencia a cualquier icaracter.
- cosa: Hace referencia a cualquier número/palabra/cadena.
- Tipo_var o **…** : Hace referencia a cualquier tipo de variable primitiva o de tipo String.
- nombre_var: Hace referencia a cualquier nombre que se le puede dar a una variable.
- código: Hace referencia a cualquier instrucción. (Se usará para indicar dónde se podrá inscribir código.)
- variable: Hace referencia a cualquier variable.
- condición: Hace referencia a cualquier condición. Entiéndase por condición, una afirmación que devuelve un true o un false. Ej: (variable == 123). *Una variable del tipo boolean puede ser usada como una condición.

Índice

keypress

keyup

load

message

mousedown

mousemove

mouseout

mouseover

mouseup

popstate

reset

resize

scroll

submit

success

unload

Título IV: Identificación de elementos html

Identificación mediante etiquetas

Identificación mediante ids

Identificación mediante clases

Identificación mediante selectores css

Título V: Propiedades de elementos html

width

length

innerHTML

appendChild(child)

offsetLeft/Right/Top/Bottom

pageX

value

checked

disabled

style.background

style.visibility

src

files

getAttribute(atrib)

Obtener coordenadas x, y del ratón

Título VI: Objeto vídeo / audio

duration

currentTime

paused

ended

play()

pause()

Título VII: Propiedades especiales de formularios

setCustomValidity(mensaje)

checkValidity()

submit()

Estados de validación

validity.valid

validity.valueMissing

validity.patternMismatch

validity.typeMismatch

validity.tooLong

validity.rangeUnderflow

validity.rangeOverflow

validity.stepMismatch

validity.customError

Título VIII: Ventanas emergentes

open(url, nombre, propiedades)

close()

Capítulo II: Canvas

width

height

Título I: Creación de objeto “context”

Título II: Creación de rectángulos y establecimiento de colores

fillRect(x, y, width, height)

strokeRect(x, y, width, height)

clearRect(x, y, width, height)

fillStyle

strokeStyle

globalAlpha

Título III: Creación de degradados

Degradados lineales

Degradados circulares

Título IV: Creación de trazados

Inicialización y final

beginPath()

closePath()

Crear trazados

moveTo(x, y)

lineTo(x, y)

rect(x, y, width, height)

arc(x, y, r, alpha, beta, circf)

bezierCurveTo(a, b, c, d, x, y)

quadraticCurveTo(a, b, x, y)

Añadir trazados

stroke()

fill()

clip()

Ejemplos

Dibujar un triángulo

Crear una máscara

Dibujar una cara sonriente (cambiando el tipo de línea)

Título V: Tipos de líneas

lineWidth

lineCap

lineJoin

miterLimit

Título VI: Texto en Canvas

Métodos para dibujar texto

strokeText(texto, x, y)

fillText(texto, x, y)

Propiedades de configuración de texto

font

measureText(texto)

textAlign

textBaseline

Ejemplo

Título VII: Propiedades de configuración de las sombras

shadowColor

shadowOffsetX

shadowOffsetY

shadowBlur

Título VIII: Transformaciones

translate(x, y)

rotate(alpha)

scale(x, y)

transform(ex, r, t, ey, dx, dy)

setTransform(ex, r, t, ey, dx, dy)

save()

restore()

Pasos para crear o conectarse a una BBDD

Título II: Creación de BBDD

Título III: Almacenamiento en BBDD

Título IV: Lectura de BBDD

Capítulo VII: File

Título I: Teoría

readAsText(archivo)

readAsBinaryString(archivo)

readAsDataUrl(archivo)

readAsArrayBuffer(archivo)

Título II: Obtener objeto File

Título III: Lectura de un objeto File

Lectura de imágenes

Título IV: Propiedades de archivos

type

name

size

fullPath

Título V: Creación de archivos “virtuales”

pasos

creación

Capítulo VIII: Ajax

Ventajas

Elementos en Ajax

Pasos para enviar petición y procesar el resultado

get

post

Título I: Consulta de archivo HTML con JQuery

load(archivoHtml)

Título II: Consultas a servidor con JQuery

$.get(url, datos, procesa)

$.post(url, datos, procesa)

$.ajax($parametros)

Título III: Excepciones en consultas con JQuery

Título IV: Objetos JSON

Acceder a JSON

Capítulo IX: Web Worker

Título I: Enviar y recibir mensajes al worker

Título II: Recibir y enviar mensajes desde el worker

Título III: SharedWorker

Capítulo X: History

back()

forward()

go(pasos)

length

window.history.pushState(estado, titulo, url)

window.history.replaceState(estado, titulo, url)

state

window.addEventListener(“popstate”, funcion, bool)

Capítulo XI: API Offline

Capítulo XII: JQuery

Librería

Ventajas

Agregar JQuery a página web

DOM

Título I: Objeto JQuery

next(selector)

width(x)

height(x)

color(c)

val(v)

focus()

Título II: Características especiales

Loops automáticos

Funciones encadenadas

Título III: Agregar contenido HTML

html()

html(codHtml)

append(codHtml)

prepend(codHtml)

before(codHtml)

after(codHtml)

text() / text(texto)

Título IV: Eliminar y reemplazar contenido

remove()

replaceWith(codHtml)

Título V: Cambiar componente HTML de clase

addClass(clase)

removeClass(clase)

toggleClass(clase)

Título VI: Leer y Cambiar atributos CSS

css(propiedad)

css(propiedad, valor)

css({propiedad:valor, propiedad:valor, …})

Título V: Cambiar componente HTML de clase

addClass(clase)

removeClass(clase)

toggleClass(clase)

Título VI: Leer y Cambiar atributos CSS

css(propiedad)

css(propiedad, valor)

css({propiedad:valor, propiedad:valor, …})

Título VII: Leer, Cambiar y eliminar atributos HTML

attr(atributo)

attr(atributo, valor)

removeAttr(atributo)

Título VIII: Función each(funcion)

Título IX: Eventos

hover(funcionIn, funcionOut)

toggle(funcionOn, funcionOff)

bind(tipoEvento, param, funcion)

Título X: Efectos

hide()

show()

toggle()

fadeIn(ms)

fadeTo(ms, opacity)

fadeOut(ms)

fadeToggle(ms)

slideDown(ms)

slideUp(ms)

slideToggle(ms)

Título XI: Animaciones

animate(propiedades, tiempo)

stop()

delay(ms)

Título XII: Plugins

Añadir plugins

Título XIII: Preloader de imágenes

Título XIV: Imágenes con FancyBox

fancybox(propiedades)

Introducción

El objetivo de este curso es aprender JavaScript y su comunicación y ayuda en páginas HTML..
La principal función de JavaScript es aportar funcionalidad a páginas web creadas con HTML y CSS3.
JavaScript es un lenguaje de programación.
En JavaScript, todas las sentencias terminan en un “;”.
En JavaScript, para especificar textos podemos usar tanto comillas simples ‘ como dobles “ aunque se recomiendan las segundas.
El código JavaScript de una página HTML se puede indicar en 3 sitios diferentes:

1. En un archivo independiente con extensión .js
2. Dentro de la etiqueta head de un documento html.
3. Dentro de una etiqueta inline en casos concretos (desaconsejada).

Capítulo I: Relación con HTML

alert(mensaje)

Muestra una ventana emergente con el texto **mensaje**.
*Pausa el hilo de ejecución hasta que se cierra.

confirm(mensaje)

Muestra una ventana emergente con el texto **mensaje** y dos botones de Aceptar (devuelve **true**) y Cancelar (devuelve **false**).
*Pausa el hilo de ejecución hasta que se cierra.

Título I: Colocación de código

inline

Donde podemos especificar el código JavaScript dentro de una etiqueta. Esto se suele usar cuando pulsamos en un componente para que se muestre una ventana de alerta, añadiendo el siguiente atributo a una etiqueta html:
onClick=”alert(‘Hola’);

Dentro del head

Donde creamos una etiqueta **script** en la que introducimos todo el código JavaScript.

Fichero externo

Forma más recomendada.
Creamos un documento .js , al que hacemos referencia dentro de nuestro .html:
Al final del body, (o en el head) incluimos la etiqueta:
<script src=”ruta”></script>
Donde indicamos en **ruta** la ruta del archivo.js (absoluta o relativa).

Título II: Eventos

Para crear un evento, creamos la función que se ejecutará cuando el evento se produzca. Una vez creada la función, la nombramos desde el código html, especificando el atributo **onClick** el nombre de la función (Consultar título III, ya que esto es una mala práctica):
(html)
** Hola **

(JavaScript)
```
function saludo(){
        alert(“Hola”);
}
```

Evento al pasar el ratón por encima

Cambiar **onClick** por **onmouseover**.

Evento al cargar la página

Podemos especificar dentro del código JavaScript la función que queremos que se ejecute cuando se cargue la página. Esto lo hacemos de la siguiente forma:
window.onload=saludo;
o
window.addEventListener(“load”, saludo, false);
Este onload lo podemos usar para especificar una “función main”, como hay en otros lenguajes.
Es recomendable que todo código JavaScript lleve un load haciendo referencia a una función que inicialice o identifique los diferentes componentes html.

addEventListener(evento, funcion, bool)

Aplicamos este método sobre un elemento del documento html que previamente hemos identificado (Consultar Título III).

Le pasamos como parámetro el tipo de evento (ver Título III), la función que se ejecuta cuando se ejecuta el evento (no indicar paréntesis de función o indicar función anónima) y un valor booleano (true o false).
*Esta instrucción no se puede meter dentro de un bucle ya que sólo se añadiría el evento al último elemento de este (como en python).
Lo que sí podemos hacer es meterla en un bucle y hacer que la función oyente reciba un parámetro evento, al que le aplicamos la propiedad **target para recibir el objeto que ha producido el evento.

funcion

Indicar una función cualquiera sin paréntesis.
Las funciones oyentes pueden ser de dos tipos: que no reciban parámetros o que reciban un parámetro.
Si no reciben parámetros no ocurre nada extraño.
Si reciben un parámetro, este será el objeto que desencadena el evento.

```
boton.addEventListener("click", funcionar, false);
function funcionar(e){
    var botoncillo = e.target;
}
```

boton y botoncillo hacen referencia al mismo elemento, pero boton antes de que suceda el evento y botoncillo una vez que el evento ya ha tenido lugar.

Valor booleano

Si metemos un elemento dentro de otro y ambos tienen eventos de click, al hacer click se ejecutarán los dos ya que estamos haciendo click sobre ambos. El valor booleano nos sirve para indicar la prioridad:

- **false**: No establece prioridad (si todos son false, se ejecuta primero el del hijo y luego los padres en orden ascendiente).

- **true**: Da prioridad a los elementos con true antes que a los que tienen false.

Objeto Evento

El objeto evento es el que se pasa automáticamente a una función oyente que recibe parámetros. Este presenta ciertas propiedades y métodos:

target

Devuelve el objeto que ha producido el evento.

clientX

Devuelve la coordenada horizontal del ratón cuando se produce el evento.

clientY

Devuelve la coordenada vertical del ratón cuando se produce el evento.

stopPropagation()

Para la propagación del evento:
En caso de que un botón y el documento tengan eventos, primero se ejecuta el evento del botón y luego el del documento, ya que se desencadenan los dos casi a la vez.
Si en el evento del botón especificamos **stopPropagation()** sobre nuestro objeto evento, detendremos esta propagación y el evento del documento no tendrá lugar.

preventDefault()

Cancela el comportamiento por defecto que realizan algunos componentes (como los **<a>** redirigiendo a páginas).

*Conseguimos el mismo efecto indicando un **return false;** al final de la función del evento.

Título III: Tipos de eventos

*Todos los eventos especificados a continuación han de ir entrecomillados si se especifican en la función **addEventListener()** .
**Si deseamos añadir los siguientes eventos desde HTML, especificar antes "on" y luego la función que ejecutamos:
<button onclick="saludar();"></button>
***Si queremos añadir estos eventos a un objeto JQuery, la función a usar es idéntica a sus nombres: **$("button").click(saludar);**

blur

Se ejecuta cuando se deselecciona el componente.

change

Se ejecuta cuando el componente cambia.
Por ejemplo: al mover un **input type="range".**

click

Se ejecuta cuando se pulsa sobre el componente con click izquierdo.

connect

Se ejecuta cuando algún fichero se conecta a un fichero **SharedWorker** (Consultar Capítulo IX, título III).

dblclick

Se ejecuta cuando se pulsa sobre el componente con click izquierdo dos veces seguidas.

input

Se ejecuta cuando se inserta texto en un campo de texto.

*Si se ejecuta sobre un formulario, cuando se inserta texto en cualquiera de sus campos de texto.

invalid

Se ejecuta cuando un campo de un formulario en inválido o no está bien validado (se aplica sobre el formulario).
*Cuando este evento ocurre, genera un error (si la función oyente recibe parámetros, el parámetro será el error). Podemos obtener el elemento que ha ocasionado el error con **error.target**:

```
document.getElementById("formulario").addEventListener("invalid", error, true);

function error(e){
	var elemento = e.target;
	elemento.style.background="FF0000";
}
```

keydown

Se ejecuta cuando se pulsa una tecla del teclado sin soltarla.

keypress

Se ejecuta cuando se pulsa una tecla del teclado.

keyup

Se ejecuta cuando se suelta una tecla del teclado pulsada.

load

Se ejecuta cuando se carga la página web (aplicar sobre window).
O cuando se crea cualquier otro componente.

message

Se ejecuta cuando el fichero actual recibe un mensaje de otro fichero (ver Capítulo IX).

mousedown

Se ejecuta cuando se pulsa un botón del ratón sin soltarlo.

mousemove

Se ejecuta cuando el usuario mueve el ratón.

mouseout

Se ejecuta cuando el ratón sale de encima de un componente.

mouseover

Se ejecuta cuando el ratón se posa encima de un componente.

mouseup

Se ejecuta cuando se suelta un botón del ratón que estaba pulsado.

popstate

Se ejecuta cuando la url de la página web cambia (ver Capítulo X).

reset

Se ejecuta cuando se borran todos los datos de los campos de texto de un formulario.

resize

Se ejecuta cuando se modifica el tamaño de la ventana del navegador.

scroll

Se ejecuta cuando el usuario se desplaza verticalmente utilizando la barra de scroll.

submit

Se ejecuta cuando se envía el formulario pulsando en el **input type=”submit” .**

success

Se ejecuta cuando algo se completa correctamente (ver Capítulo VI, Título III, punto 2).

unload

Se ejecuta cuando se cierra la página web y se vacía la memoria.

Título IV: Identificación de elementos html

Para no programar código correspondiente a eventos dentro de etiquetas, identificamos los elementos html desde JavaScript.
Podemos usar referencias a elementos html para modificar sus propiedades:

Identificación mediante etiquetas

Esto lo hacemos con el método **document.getElementsByTagName(name)**, que devuelve un array con todas las etiquetas del tipo **name** que hay en nuestro documento .html.
Posteriormente, podemos indicar el campo **.onClick** que sea igual a una función, como hicimos en el título II:
document.getElementsByTagName(“p”)[0].onClick = saludo;

Identificación mediante ids

Esto lo hacemos con el método **document.getElementById(id)**, que devuelve un objeto con el id especificado en el archivo .html.
Dentro del código html, debemos especificar la propiedad **id**=”id” para poder hacer referencia a esta etiqueta desde el código JavaScript:
(HTML)
** Hola **

(JavaScript)
document.getElementById(“etiqueta”).onClick = saludo;

Identificación mediante clases

Esto lo hacemos con el método **document.getElementsByClassName(clase)**, que devuelve un array con todos los elementos que pertenecen a esta clase.
(HTML)
** Hola **

(JavaScript)
var z = document.getElementsByClassName(“importante”);
z[0].onClick = saludo;

Identificación mediante selectores CSS

-La función **querySelector(cssSelector)** nos devuelve el primer elemento que corresponde al grupo de selectores indicados en **cssSelector**.
(html) ** Hola **

(JavaScript) **document.querySelector(“p”).onClick = saludo;**

-La función **querySelectorAll(cssSelector)** nos devuelve todos los elementos que corresponden al grupo de selectores indicados en **cssSelector**. Estos elementos se almacenan en un array, conteniéndolos en el mismo orden en el que aparecen en el documento.
(html) ** Hola **
** A **
** B **

(JavaScript)

```
var z = document.querySelectorAll("p");
for(var i=0; i<z.length; i++){
        z.onClick = saludo;
}
```

Título V: Propiedades de elementos html

Podemos modificar o ver propiedades de componentes html desde JavaScript haciendo uso de diferentes propiedades y métodos de cada objeto.

*Podemos cambiar cualquier propiedad de CSS (ver Apuntes CSS, Capítulo II), siempre y cuando especifiquemos **style** antes:

componente.style.background = "#0FF";
componente.style.border = "1px solid #999";

width

Establece lo ancho que será un componente.

componente.width = 60;

length

Establece la altura de un componente.

componente.lenght = 100;

innerHTML

Establece el texto interno de un componente.

componente.innerHTML = "Hola";

*Lo que de verdad hace es establecer un componente dentro de otro, aunque si establecemos texto, este es como si fuera un componente:

componente.innerHTML = "<img src="ruta">";

Así, establecemos una imagen dentro de un componente ya creado.

**Indicar “” para vaciar el componente:
componente.innerHTML=””;

appendChild(child)

Establece el componente **child** dentro del componente sobre el que se aplica el método (sin borrar los ya existentes (a diferencia de **innerHTML**)).

offsetLeft/Right/Top/Bottom

Devuelve la distancia (int) en píxeles que hay entre el margen izquierdo (/derecho/superior/inferior) de la página y el borde izquierdo (/derecho/superior/inferior) del componente.

pageX

Devuelve la distancia del margen izquierdo a la que el ratón es pulsado. Para usar esta propiedad, debemos hacer que nuestra función oyente reciba como parámetro un objeto sobre el que posteriormente aplicaremos esta propiedad, aunque no se especifiquemos ningún parámetro en la llamada.

```
function oyente(parametro){
    document.write(parametro.pageX);
}
```

elemento.addEventListener(“click”, oyente, false);

value

Devuelve / establece el texto que tienen los campos de texto.

checked

Devuelve/establece **true** si el componente (radiobutton o checkbox) está seleccionado, **false** si no.

disabled

Establece/Devuelve si el componente está desactivado (**true)** o no (**false)**.

style.background

Devuelve / establece el color de fondo (hexadecimal: #000000).

style.visibility

Establece la visibilidad del componente:
“hidden” para que el componente no sea visible.
“visible” para que el componente sea visible.

src

Devuelve el atributo **src** del componente HTML.

files

Devuelve todos los archivos que han sido seleccionados en un componente HTML **input** de **type=file** como si fuese un array.

getAttribute(atrib)

Devuelve el atributo especificado en **atrib**:
var ruta = imagen.getAttribute(“src”);

Obtener coordenadas x, y del ratón

Para ello, primero debemos establecer un evento “mouesmove” sobre **window** para detectar cada vez que el ratón se mueva y luego ejecutar las propiedades **clientX** y **clientY**:

```
window.addEventListener(“mousemove”, mover, false);

function mover(e){
    var x = e.clientX;
    var y = e.clientY;
```

}

Título VI: Objeto vídeo / audio

Al hacer referencia a una etiqueta html **video** o **audio** , obtenemos un objeto vídeo (o audio) el cual tiene una serie de propiedades y de métodos que podemos usar y ejecutar para modificar la reproducción de este.

duration

Duración del vídeo (o audio).

currentTime

Tiempo actual del vídeo (o audio) (por dónde se va reproduciendo).

paused

true: vídeo (o audio) pausado, false: vídeo (o audio) no pausado.

ended

true: vídeo (o audio) finalizado, false: vídeo (o audio) no finalizado.

play()

Reproduce el vídeo (o audio).

pause()

Para el vídeo (o audio).

Título VII: Propiedades especiales de formularios

Desde JavaScript, podemos asignar una serie de propiedades especiales a ciertos formularios.

setCustomValidity(mensaje)

Se establece sobre un componente **input** de un formulario.

Lo que hace es imprimir el mensaje **mensaje** como una falta de validación en un componente. Ej:

```
if (campo.value==””){
        campo.setCustomValidity(“Debes introducir un valor”);
}
```

*Cuando se llama a este método, el mensaje se mantiene durante todo el programa, por lo que debemos anularlo con **setCustomValidity(“”)**

```
if (campo.value==””){
        campo.setCustomValidity(“Debes introducir un valor”);
}else{
        campo.setCustomValidity(“”);
}
```

checkValidity()

Se establece sobre un formulario.
Devuelve true si los campos del formulario están validados, false si no.
if(document.formulario.checkValidity() == true){ código; }
Donde **formulario** es el id de un **form** .

submit()

Se establece sobre un formulario.
Hace la misma función que si pulsamos sobre el **input** de **type=”submit”** , aunque no valida los campos**.**
document.formulario.submit();
Donde **formulario** es el id de un **form** .

Estados de validación

Estos estados son propiedades que nos permiten saber si un campo está validado o no en una propiedad específica.

Estas propiedades nos permiten mostrar ayuda al usuario para saber qué es lo que está mal validado en el formulario.
Todos estos estados se aplican sobre componentes de un formulario.

validity.valid

Es igual a **true** si este componente está validado, false si no.

validity.valueMissing

true cuando el campo es **required** y está vacío.

validity.patternMismatch

true cuando el valor no coincide con el formato pattern.

validity.typeMismatch

true cuando la sintaxis no coincide con el tipo del campo.

validity.tooLong

true cuando se excede del **maxlength** establecido en html.

validity.rangeUnderflow

true cuando sobrepasa **min**

validity.rangeOverflow

true cuando sobrepasa **max**

validity.stepMismatch

true cuando el valor de un **input number** no se corresponde con **step**.

validity.customError

true cuando declaramos error personalizado con **setCustomValidity()**

Título VIII: Ventanas emergentes

open(url, nombre, propiedades)

Crea una ventana emergente a la url especificada en **url** con el nombre interno especificado en **nombre** (no lo verá el usuario) y con las propiedades especificadas en el String **propiedades**:

open(“https://www.google.com”, “Google”, “width=300, height=300, left=50, top=50”)

width y height indican el ancho y alto y left y top las coordenadas respecto a la ventana del navegador.

close()

Podemos ejecutar este método sobre una ventana para cerrarla:

var ventana = open(“ruta”, “nombre”, “propiedades”);
código;
e.preventDefault();
ventana.close();

*Las ventanas emergentes son bloqueadas por el navegador, ya que pueden ser utilizadas de forma maliciosa para mostrar publicidad, por lo que se recomienda utilizar etiquetas **<iframe> .**

Capítulo II: Canvas

Los objetos Canvas de HTML (ver Apuntes HTML, Capítulo II) tienen una serie de métodos que nos permiten dibujar y eliminar objetos del Canvas.
Para ello, necesitamos nuestro objeto **context** que obtenemos con el método **getContext(d)**
*Consultar API Canvas.

width

Devuelve/Establece el ancho del Canvas:
var canvas = document.getElementById(“canvas”);
canvas.width = 50;

height

Devuelve/Establece la altura del Canvas:
var canvas = document.getElementById(“canvas”);
canvas.height = 80;

Título I: Creación de objeto “context”

getContext(d) devuelve un objeto que nos servirá para dibujar o borrar en el Canvas. Ejecutar sobre nuestro objeto canvas. Especificar dimensiones en **d** (**“2d”** o **“3d”**):

var canvas = document.getElementByTagName(“canvas”);
lienzo = canvas.getContext(“2d”);

*En este título, trabajaremos siempre en 2d.

Una vez que tenemos nuestro objeto, podemos hacer lo siguiente:

Título II: Creación de rectángulos y establecimiento de colores

fillRect(x, y, width, height)

Dibuja un rectángulo de **width** x **heigth** en las coordenadas **x** , **y** .
lienzo.fillRect(100, 100, 50, 50);
*El color por defecto es negro.

strokeRect(x, y, width, height)

Dibuja los bordes de un rectángulo de **width** x **heigth** en las coordenadas **x** , **y** .
*El color por defecto es negro.

clearRect(x, y, width, height)

Borra con una goma rectangular de **width** x **heigth** en las coordenadas **x** , **y** . (Pone el color del Canvas)

fillStyle

Establece el color en el que se dibujarán los siguientes elementos:
lienzo.fillStyle = “#FF0000”;
*El color por defecto es el negro.

strokeStyle

Establece el color de los bordes que se dibujen con **strokeRect()**
lienzo.strokeStyle = “#00FF00”;
*El color por defecto es el negro.

globalAlpha

Establece el alpha (transparencia) de todos los componentes a dibujar. Acepta valores entre 0 y 1 (donde 0 es transparente total y 1 opaco total).
*El valor de alpha por defecto es de 1.

Título III: Creación de degradados

Para crear degradados, primero debemos crear un objeto que será nuestro degradado y luego tenemos que aplicar este al canvas.
Hay dos tipos de degradados que podemos crear:

Degradados lineales

1. Creamos un objeto con la ayuda de la función **createLinearGradient(x0, y0, x, y)** donde especificamos que va a ser un degradado de tipo lineal y el punto donde este empieza (x0, y0) y el punto donde este acaba (x, y).
2. Posteriormente, debemos darle dos colores a este objeto. Esto lo hacemos con la ayuda del método **addColorStop(posicion, color)** , donde especificamos qué color estamos añadiendo (el primero (0) o el segundo (1)) y el color a añadir (en hexadecimal) (*).
3. Añadimos nuestro degradado al objeto Context igualando nuestro objeto degradado a la propiedad **fillStyle** de nuestro objeto Context, como si fuera un color normal.
4. Creamos nuestro rectángulo con el tamaño que queramos (como si lo hacemos de igual tamaño que el canvas, para establecer el degradado de fondo).
(El Canvas es de 500x300 px):

```
var canvas = document.getElementByTagName("canvas");
lienzo = canvas.getContext("2d");

var degradado = lienzo.createLinearGradient(0, 150, 500, 150);
degradado.addColorStop(0, "#FF0000");
degradado.addColorStop(1, "#00FF00");
lienzo.fillStyle = degradado;

lienzo.fillRect(0, 0, 500, 300);
```

*Verdaderamente, en el área del degradado, se define el margen izquierdo como un 0 y el margen derecho como un 1, indicando valores decimales para la parte intermedia entre ambos.
Lo que hacemos con este valor es especificar dónde queremos que el color acabe, que por lo general suele ser en 0 y 1 (aunque también podemos modificar los valores para mostrar más un color u otro).

Degradados circulares

1. Creamos un objeto con la ayuda de la función **createRadialGradient(x1, y1, r1, x2, y2, r2)**, donde especificamos el centro del primer círculo (x1, y1) y su radio (r1) y el centro del segundo círculo (x2, y2) y su radio (r2).
2. Agregamos los colores a este objeto con el método **addColorStop(posicion, color)** donde especificamos qué color estamos añadiendo (el primero (0) o el segundo (1)) y el color a añadir (en hexadecimal) (si no funciona, invertir 1 por 0 y 0 por 1).
3. Añadimos nuestro degradado al objeto Context igualando nuestro objeto degradado a la propiedad **fillStyle** de nuestro objeto Context, como si fuera un color normal.
4. Creamos nuestro rectángulo con el tamaño que queramos (como si lo hacemos de igual tamaño que el canvas, para establecer el degradado de fondo).
(El Canvas es de 500x300 px):

```
var canvas = document.getElementByTagName(“canvas”);
lienzo = canvas.getContext(“2d”);

var degradado = lienzo.createRadialGradient(250, 150, 50, 250, 150, 300);
degradado.addColorStop(1, “#FF0000”);
degradado.addColorStop(0, “#00FF00”);
lienzo.fillStyle = degradado;
```

lienzo.fillRect(0, 0, 500, 300);

Título IV: Creación de trazados

Para dibujar un trazado en Canvas, tenemos que seguir diferentes pasos:
1. Crear el trazado (con beginPath() , otros métodos de trazado y closePath()).
2. Añadir el trazado al Canvas (con método como stroke(), fill() o clip()). Esto métodos añaden todos los trazados que se crearon desde el último **beginPath()**.

Para ello, debemos usar los métodos especificados a continuación (todos ellos se ejecutan sobre nuestro objeto "context" del canvas):
*La línea del trazado se puede personalizar (Consultar Título V).

Inicialización y final

beginPath()

Marca el inicio de la creación del trazado. Toda creación comienza por este método.

closePath()

Marca el final de la creación del trazado. Toda creación termina por este método (aunque en algunas veces se puede prescindir de él).

Crear trazados

moveTo(x, y)

Mueve el lápiz a la posición inicial marcada en las coordenadas.
*Es necesario usar este método antes de otros métodos de trazado, aunque no en todos.

lineTo(x, y)

Genera una línea recta desde la posición actual del lápiz hasta las nuevas coordenadas. (moviendo el lápiz a su vez).
*Necesario indicar **moveTo(x, y)** antes.

rect(x, y, width, height)

Genera un rectángulo en **x, y** de **width x height**.

arc(x, y, r, alpha, beta, circf)

Genera un arco de una circunferencia con centro en las coordenadas **x, y** y con el radio **r**. El arco irá desde el ángulo inicial **alpha** hasta el ángulo final **beta** (ambos en radianes).
El último parámetro **circf** es un booleano al que le indicamos si queremos recibir el arco indicado (**false**) o el resto de la circunferencia a la que le hemos truncado este arco (**true**).

*Los ángulos se mueven en sentido horario, siendo el ángulo 0 el extremo que se encuentra más a la izquierda de la circunferencia.
Si especificamos alpha = 0, beta = Math.PI y el método **fill, creamos un círculo.

bezierCurveTo(a, b, c, d, x, y)

Genera una curva bézier cúbica (con dos puntos de control).

Dibuja una curva desde el punto especificado en **moveTo(x, y)** hasta el punto especificado en **x, y** (del propio método), siguiendo dos puntos de control que se sitúan en **a, b**. (mover punto de control para cambiar apariencia de la curva) y en **c, d** (mover punto de control para cambiar apariencia de la curva).

*Necesario indicar **moveTo(x, y)** antes.
****a,b** se corresponden con el punto **moveTo(x, y)**, mientras que **c, d** con el punto **x, y**.

quadraticCurveTo(a, b, x, y)

Genera una curva bézier cuadrática (con un punto de control).

Dibuja una curva desde el punto especificado en **moveTo(x, y)** hasta el punto especificado en **x, y** (del propio método), siguiendo un punto de control que se sitúa en **a, b**. (mover punto de control para cambiar apariencia de la curva).

*Necesario indicar **moveTo(x, y)** antes.

Añadir trazados

stroke()

Dibuja sólamente el contorno de la figura en el Canvas.

fill()

Dibuja la figura sólida en el Canvas con el color que tenga el objeto context en ese momento (consultar Título II).

clip()

Crea una máscara a partir de la figura.
Si queremos crear una máscara con un trazado, no tenemos que cerrar el trazado con closePath() .
*Una “ventana” que sólo nos deja ver una parte del canvas.
**Si no hay una máscara, se mostrarán todos los trazados. Si hay una máscara, sólo se mostrarán aquellos que estén dentro de una máscara (que sus coordenadas estén dentro de las coordenadas de la máscara).
***Las máscaras son invisibles, ya que sólo se ven los componentes que hay en su interior.

Ejemplos

Dibujar un triángulo

```
var canvas = document.getElementById(“canvas”);
```

```
lienzo = canvas.getContext("2d");

lienzo.beginPath();          //Empieza el trazado.
lienzo.moveTo(100, 200);     //Especifica las coords de inicio.
lienzo.lineTo(200, 200);     //Dibuja una línea (100, 200, 200, 200).
lienzo.lineTo(150, 150);     //Dibuja una línea (200, 200, 150, 150).
lienzo.lineTo(100, 200);     //Cierra el triángulo con la última línea.
lienzo.closePath();          //Termina el trazado.

lienzo.stroke();             //Dibuja el trazado en el Canvas.
```

Crear una máscara

```
var canvas = document.getElementById("canvas");
lienzo = canvas.getContext("2d");

lienzo.beginPath();          //Empieza el trazado.
lienzo.moveTo(100, 200);     //Especifica las coords de inicio.
lienzo.lineTo(200, 200);     //Dibuja una línea (100, 200, 200, 200).
lienzo.lineTo(150, 150);     //Dibuja una línea (200, 200, 150, 150).
lienzo.lineTo(100, 200);     //Cierra el triángulo con la última línea.
lienzo.clip();     //Termina el trazado, crea la máscara y la añade.
```

Dibujar una cara sonriente (cambiando el tipo de línea)

```
var canvas = document.getElementById("canvas");
lienzo = canvas.getContext("2d");

lienzo.beginPath();
lienzo.arc(200, 150, 100, 0, Math.PI*2, false); //Crea la cara.
lienzo.stroke();

lienzo.lineWidth=10;
lienzo.lineCap="round";
lienzo.beginPath();
```

```
lienzo.arc(200, 170, 60, 0, Math.PI, false); //Crea la boca.
lienzo.stroke();

lienzo.lineJoin = "miter";
lienzo.lineWidth = 5;
lienzo.beginPath();
lienzo.moveTo(195, 135);
lienzo.lineTo(215, 155);          //Dibuja
lienzo.lineTo(195, 155);          //la nariz

lienzo.lineWidth = 1;
lienzo.beginPath();
lienzo.arc(168, 106, 7, 0, Math.PI*2, false);      //Crea interior ojo
lienzo.arc(225168, 106, 7, 0, Math.PI*2, false);   //Crea interior ojo
lienzo.fill();

lienzo.beginPath();
lienzo.arc(168, 106, 15, 0, Math.PI*2, false);     //Crea exterior ojo
lienzo.arc(225168, 106, 15, 0, Math.PI*2, false);  //Crea exterior ojo
lienzo.stroke();
```

Título V: Tipos de líneas

Existe la posibilidad de cambiar el tipo de línea antes de realizar un trazado.

*Cada vez que cambiemos las características de la línea, ha de comenzarse un nuevo trazado con **beginPath()** después de cambiar la línea, para que el objeto se "actualice".

**Todos los campos de clase vistos a continuación pertenecen a nuestro objeto context.

lineWidth

Establece el grosor de la línea (por defecto: **lineWidth = 1**).

lineCap

Establece la terminación de la línea.
Puede adoptar tres valores: **“butt”**, “**round”**, “**square”**.
(butt es similar a square pero menos brusco).

lineJoin

Establece la conexión entre líneas.
Puede adoptar tres valores: **“round”**, “**bevel”, “miter”**.
(round deja la unión redondeada, miter en pico y bevel recorta la unión).

miterLimit

Establece cuánto ha de extenderse la unión entre líneas. Sólo se usa cuando el **lineJoin** = **miter** .
(por defecto **miterLimit=5**)
*Cuando menor sea el número, más se acorta.

Título VI: Texto en Canvas

Para mostrar texto en un Canvas, tenemos que especificar al objeto context todas las propiedades de configuración del texto y de su sombra. Una vez especificadas, usaremos los métodos **strokeText()** o **fillText()** para mostrar el texto en el Canvas.

Métodos para dibujar texto

strokeText(texto, x, y)

Dibuja el contorno del texto (sin relleno) **texto** en las coordenadas **x, y** del Canvas.

fillText(texto, x, y)

Dibuja el texto (con relleno) **texto** en las coordenadas **x, y** del Canvas.

Propiedades de configuración de texto

font

Fuente del texto. Acepta los mismos valores que CSS (Consultar Apuntes CSS, Capítulo II, font).
(**font = “bold 24px Verdana”;** //por ejemplo)

measureText(texto)

Devuelve información sobre el espacio que ocupa en el Canvas el texto especificado.
*Almacenar en variable y aplicar .width:
(Útil en animaciones).
var dimensiones = lienzo.measureText(“Hola mundo”);
var ancho = dimensiones.width;

textAlign

Alineamiento horizontal. Acepta los siguientes valores:
“start”, “end”, “left”, “right”, “center” .

*start y end indican que o bien se coloque el texto después de las coordenadas especificadas (“start”) o bien que el texto termine justo en las coordenadas especificadas (“end”).

textBaseline

Alineamiento vertical. Acepta los siguientes valores:
“top”, “hanging”, “middle”, “alphabetic”, “ideographic”, “bottom” .

*top y bottom indican si las coordenadas especificadas están por debajo (“bottom”) o por encima del texto (“top”).

Ejemplo

var canvas = document.getElementById(“canvas”);

```
lienzo = canvas.getContext(“2d”);

lienzo.font = “bold 24px Verdana”;
lienzo.textAlign = “start”;
lienzo.textBaseLine = “top”;
lienzo.fillText(“Hola mundo”, 100, 100);
```

Título VII: Propiedades de configuración de las sombras

Se pueden crear sombras sobre casi todos los componentes que hay en un Canvas, como un rectángulo, un trazado o un texto. Para ello, cambiar los valores de los siguientes campos de clase pertenecientes al objeto context:

```
var canvas = document.getElementById(“canvas”);
lienzo = canvas.getContext(“2d”);

lienzo.shadowColor = “rgba(0, 0, 0, 0.5)”;
lienzo.shadowOffsetX = 5;
lienzo.shadowOffsetY = 5;
lienzo.shadowBlur = 6;

lienzo.fillText(“Hola”, 50, 50);

lienzo.beginPath();
lienzo.moveTo(50, 50);
lienzo.lineTo(100, 50);
lienzo.closePath();
lienzo.stroke();
```

*El texto y la línea tendrán la misma sombra.

shadowColor

Indica el color de la sombra. Acepta los mismos colores que CSS. (Consultar Apuntes CSS, Capítulo II, **color** y Título III).

shadowOffsetX

Indica el desplazamiento horizontal de la sombra en píxeles (no indicar px, sólo el número entero).

shadowOffsetY

Indica el desplazamiento vertical de la sombra en píxeles (no indicar px, sólo el número entero).

shadowBlur

Indica el difuminado de la sombra (número entero).

Título VIII: Transformaciones

Para realizar transformaciones, debemos ejecutar los siguientes métodos sobre nuestro objeto context para realizarlas sobre todo el canvas.

Cualquier componente agregado al Canvas antes de realizar una transformación permanecerá igual, pero si se realiza una transformación y se añade un nuevo componente, a este le afectará la transformación realizada.

```
var canvas = document.getElementById("canvas");
lienzo = canvas.getContext("2d");

lienzo.fillText("Texto normal", 100, 100);
lienzo.scale(2, 2);
lienzo.fillText("Texto el doble de ancho", 100, 100);
```

translate(x, y)

Mueve el origen del canvas (el eje de coordenadas) al punto **x, y** .

rotate(alpha)

Rota el canvas **alpha** radianes utilizando la esquina superior izquierda como centro de giro (permite ángulos negativos, los cuales invierten la dirección del giro).

scale(x, y)

Aumenta o disminuye el tamaño del canvas.
Multiplicando el tamaño actual de las x por **x** y el tamaño en el eje y por **y** .

Por lo tanto, si aplicamos **scale(2, 3)** le estaremos dando el doble de anchura y el triple de altura a nuestro canvas.

transform(ex, r, t, ey, dx, dy)

Cambia las características del canvas.
(valores por defecto: **transform(1, 0, 0, 1, 0, 0)**)

- **ex:** Corresponde a la escala a aplicar en el eje x (ver **scale(x, y)**).
- **r:** Rota el canvas **r** radianes (ver **rotate(alpha)**).
- **t:** "Tumba" el canvas según el número especificado.
- **ey:** Corresponde a la escala a aplicar en el eje y (ver **scale(x, y)**).
- **dx:** Indica nueva coordenada x del origen (ver **transform(x, y)).**
- **dy:** Indica nueva coordenada y del origen (ver **transform(x, y)).**

*El método transform es acumulativo, por lo que la próxima vez que se aplique un **transform**, este se ejecutará sobre el **transform** ya realizado la vez anterior.

setTransform(ex, r, t, ey, dx, dy)

Reinicia las características del canvas.

Funciona igual que **transform()** , salvo que este no es acumulativo, sino que establece sólo las propiedades establecidas.

*Propiedades de un canvas recién creado (valores default):
setTransform(1, 0, 0, 1, 0, 0)

save()

Graba el estado del canvas.

restore()

Restaura el estado grabado anteriormente del canvas.

Título IX: Superposiciones

Hay ocasiones en las que varios elementos de dentro de un Canvas se pueden superponer uno encima de otro. Para controlar esta superposición de la forma que queramos, tenemos el campo de clase **globalCompositeOperation**, (del objeto context) el cual puede adoptar diferentes valores:

“source-in”, “source-out”, “source-atop”, “lighter”, “xor”, “destination-over”, “destination-in”, “destination-out”, “destination-atop”, “darker”, “copy” .
(Consultar API).

*Por defecto, los últimos elementos puestos se superponen a los elementos antiguos.
Indicar valor de **globalCompositeOperation al principio del canvas o antes de la superposición en cuestión.

Título X: Dibujar imágenes

Para dibujar imágenes, usaremos el método **drawImage()** , el cual presenta sobrecarga de métodos y, por lo tanto, puede recibir diferentes parámetros:

*Todos estos métodos reciben un objeto **Image**.
El método **drawImage lo tenemos que indicar después de crear el objeto Image, como función oyente del evento "load" del objeto Image. (ver **uso**)

Objetos Image

Podemos crear objetos Image de forma fácil especificando la ruta de la imagen:

```
var nombre = new Image();
nombre.src = ruta;          //La ruta puede ser absoluta o relativa.

var foto = new Image();
foto.src = "C:/Images";     //Podemos usar la carpeta del .js como
                            //raíz de la ruta
```

drawImage(imagen, x, y)

Donde colocamos en las coordenadas **x, y** la imagen **imagen**.

drawImage(imagen, x, y, ancho, alto)

Donde colocamos en las coordenadas **x, y** la imagen **imagen** de **ancho** x **alto**.

drawImage(imagen, x1, y1, ancho1, alto1, x2, y2, ancho2, alto2)

Establece que colocará el recorte (que empieza en **x1, y1**) (y de tamaño **ancho2 x alto2**) de la imagen **imagen** de **ancho1 x alto1** píxeles de tamaño en las coordenadas **x2, y2** .

uso

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

```
lienzo = canvas.getContext("2d");

var foto = new Image();
foto.src = ruta;

function carga(){
        lienzo.drawImage(foto, 20, 20);
}

foto.addEventListener("load", carga, false);
```

*Se puede sustituir **carga** por una función anónima.
Se puede usar otra sobrecarga del método **drawImage().

Imágenes como datos

Podemos manipular imágenes ya existentes en nuestro Canvas usando las imágenes como datos.
Para ello, podemos usar los siguientes métodos (los cuales se ejecutan sobre nuestro objeto context):

*Podemos modificar los valores del array que devuelve **getImageData** para modificar el color de los píxeles (ver vídeo 36 del curso HTML5).

getImageData(x, y, width, height)

Devuelve un array en el que se especifican todos los datos de la imágen extraída en las coordenadas **x, y** de **width** x **height** .

*En caso de que el width y el height no coincidan con los de la imagen, se obtendrá un recorte de esta.

putImageData(info, x, y)

Establece la imágen de información **info** en las coordenadas **x, y**.

Título XI: Creación de animaciones

Podemos usar los siguientes métodos para crear animaciones dentro de un Canvas:

*Realmente, las animaciones son limpiar parte del canvas y volver a crear los componentes en otro estado y así sucesivamente.

clearRect(x, y, width, height)

(Visto anteriormente).
Borra con una goma rectangular de **width** x **heigth** en las coordenadas **x** , **y** . (Pone el color del Canvas).

setInterval(funcion, delay)

(Se verá próximamente).
Podemos crear un bucle infinito (para al cerrar el programa) que ejecute una función repetidas veces especificando un delay entre llamada y llamada (esto sin que detenga el hilo en el que se encuentra).
Esto lo hacemos ejecutando la función:
setInterval(funcion, delay) , a la que le especificamos la función que se ejecutará varias veces y los milisegundos de delay:

setInterval(escribir, 1000);
La función escribir será llamada infinitamente cada 1000 milisegundos = 1 segundo.

clearInterval()

(Se verá próximamente).
Elimina el **setInterval()** que se esté reproduciendo en ese momento.

Capítulo III: Drag and Drop

Para crear un sistema Drag and Drop, debemos crear dos componentes: el objeto que será arrastrado y el objeto de destino, donde soltaremos el objeto arrastrado.
Drag and drop funciona principalmente con imágenes y documentos.

preventDefault()

Los navegadores, por defecto, no permiten arrastrar componentes. Esto lo solventamos con el método **preventDefault()**:
destino.addEventListener("dragenter", function(e){e.preventDefault();}, false);

*Esto lo hacemos sobre los eventos **dragenter**, **dragover y drop**, para que el usuario vea que puede soltar ahí el objeto.

Título I: Eventos

Estos dos objetos (el arrastrado y el destino) cuentan con la posibilidad de emitir eventos en diferentes ocasiones:

Objeto arrastrado

- **"dragstart"** : Cuando se comienza a arrastrar el objeto.
- **"drag"** : Durante el desplazamiento del objeto.
- **"dragend"** : Cuando se termina de arrastrar el objeto.

*Estos métodos permiten que si la función oyente recibe un parámetro, este sea un objeto evento. Podemos aplicar la propiedad **target** para obtener el objeto que hace referencia al componente HTML:
arrastrado.addEventListener("dragenter", function(e){var arrastra = e.target; código;}, false);

Objeto destino

- **“dragenter”** : Cuando el ratón entra en el área de destino.
- **“dragover”** : Cuando el ratón se mueve sobre el área de destino.
- **“drop”** : Cuando el elemento se suelta en el área de destino.
- **“dragleave”** : Cuando el elemento arrastrado sale del área de destino (sin ser soltado dentro de esta).

*Estos métodos permiten que si la función oyente recibe un parámetro, este sea un objeto evento. Podemos aplicar la propiedad **target** para obtener el objeto que hace referencia al componente HTML:

destino.addEventListener(“dragenter”, function(e){var desti= e.target; código;}, false);

Título II: Creación del objeto arrastrado

1. Para ello, lo primero es identificar al objeto HTML (que puede ser cualquiera, por ejemplo: una imagen) desde código JavaScript y almacenarlo en una variable (ver Capítulo I, Título III).

*Sobre esta variable, podemos añadir los eventos vistos en el Título I, Objeto arrastrado.

2. Posteriormente, dentro del oyente del evento **dragstart** , establecemos cuál es el componente que queremos mover, guardando con la ayuda del método **setData()** el componente que queremos colocar en la zona destino.

```
var arrastrado = document.getElementById(“arrastrado”);

function empieza(e){
    var codigo = “<img src=”” + arrastrado.getAttribute(“src”) + ““>”;

    e.dataTransfer.setData(“Text”, codigo);
}
```

arrastado.addEventListener(“dragstart”, empieza, false);

*Cambiar la variable **codigo** en función de qué componente HTML estamos usando como objeto arrastrado.

Título III: Creación del objeto destino

1. Para ello, lo primero es identificar al objeto HTML (que puede ser cualquiera, por ejemplo: una **section**) desde código JavaScript y almacenarlo en una variable (ver Capítulo I, Título III).
*Sobre esta variable, podemos añadir los eventos vistos en el Título I, Objeto destino.
**Es buena idea establecer un texto como “Arrastre aquí el elemento” y un borde con la ayuda de CSS .
2. Dentro del oyente de **drop** , reseteamos el comportamiento con **preventDefault()** y creamos una nueva etiqueta HTML dentro de nuestro objeto destino con el objeto arrastrado, para que este lo contenga:

```
var destino = document.getElementById(“destino”);

function soltar (e){
        e.preventDefault();
        destino.innerHTML = e.dataTransfer.getData(“Text”);
}

destino.addEventListener(“drop”, soltar, false);
```

Título IV: Creación del objeto destino de archivos

Aparte de crear un objeto de destino para componentes de la página web (como fotos), también lo podemos hacer para archivos externos a la página web, que cualquier usuario tenga en su dispositivo.

En este caso, no hace falta crear un objeto arrastrable, ya que de esto se encarga nuestro OS.
*El usuario podrá incluir en nuestro objeto destino tantos archivos como desee, aunque esto lo podemos impedir con código JavaScript.

1. Para ello, lo primero es identificar al objeto HTML (que puede ser cualquiera, por ejemplo: una **section**) desde código JavaScript y almacenarlo en una variable (ver Capítulo I, Título III).
*Sobre esta variable, podemos añadir los eventos vistos en el Título I, Objeto destino.
**Es buena idea establecer un texto como "Arrastre aquí el archivo" y un borde con la ayuda de CSS .
2. Dentro del oyente de **drop** , reseteamos el comportamiento con **preventDefault()** y creamos un objeto que almacene el array que nos devuelva la propiedad **files** de **dataTransfer** (cada elemento de este array se corresponde con un archivo y cada elemento tiene unas propiedades específicas con las que podemos obtener información de cada archivo):

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

function soltarArchivo (e){
	e.preventDefault();

	var archivos = e.dataTransfer.files;
	var lista = "";
	for(var i=0; i<archivos.length; i++){
		lista += archivos[i].name + " ";
	}

	destino.innerHTML = "Los archivos incluidos son: " + lista;
}
```

destino.addEventListener(“drop”, soltarArchivo , false);

.name : Devuelve el nombre del archivo.
.size : Devuelve el tamaño del archivo (en Bytes).
.type : Devuelve el tipo de archivo.
.lastModifiedDate.toLocaleDateString() : Devuelve la última fecha de modificación del archivo.

Capítulo IV: Geolocation

Existe una **API Geolocation** que nos permite obtener la localización actual del usuario que está accediendo a nuestra página web.
Esta API consta principalmente de varios métodos que nos devuelven un objeto **Position**.

*Para que esta API funcione, el navegador debe tener la opción de geolocalización activada.

Título I: Objeto Position

El objeto Position lleva con él una serie de propiedades que podemos usar para identificar la posición del usuario:

coords

Es el que nos permite conocer la ubicación. Contiene un conjunto de valores que determinan la posición:

- **latitude :** Latitud del usuario.
- **longitude :** Longitud del usuario.
- **altitude :** Altitud del usuario.
- **accuracy :** Exactitud de la latitud y longitud (margen de error en metros).
- **altitudeAccuracy :** Exactitud de la altitud (margen de error en metros).
- **heading :** Orientación en grados de la posición.
- **speed :** El desplazamiento en **m/s** .

```
var latitud = posicion.coords.latitude;
var longitud = posicion.coords.longitude;
...
```

timestamp

Indica el momento exacto en el que se determinó la posición.

Título II: Obtener ubicación del usuario

Para obtener la ubicación del usuario, usamos el método **getCurrentPosition(exito, errores, config)** , donde sólo es obligatorio indicar el parámetro **exito** (los demas se pueden ignorar). El parámetro **exito** hace referencia a una función que recibirá nuestro objeto **Position** (ver Título I):

```
navigator.geolocation.getCurrentPosition(mostrar);

function mostrar (posicion){}
```

errores

Es posible que la geolocalización no se realice de forma exitosa (por no tener la opción activada en el navegador, tarda más de lo esperado...) Debido a esto, contamos con este parámetro del método **getCurrentPosition()** .

Este parámetro funciona de forma similar al parámetro **exito**, por lo que crearemos un método que haga referencia a una función que recibe un parámetro (el parámetro en este caso será un objeto error en vez de un objeto **Position** (el cual tiene un par de propiedades)). Dentro de este método, es buena idea mostrar un mensaje haciendo referencia a que ha habido un error.

code

Devuelve el código de error del error:

- **1 :** Permiso denegado.
- **2 :** Ubicación no disponible.
- **3 :** Tiempo para detectar ubicación excedido.

message

Devuelve el mensaje del error, el cual depende del error generado.

```
navigator.geolocation.getCurrentPosition(mostrar, error);

function error(er){
        alert("Error " + er.code + ": " + er.message + ".");
}
```

config

Podemos establecer tres propiedades sobre un objeto configuración para personalizar la localización:

enableHighAccuracy

Booleano. Se utilizarán sistemas GPS para localización si es **true**. Si es **false**, no. (por defecto es **false**).

timeout

Entero. Establece el tiempo en milisegundos para llevar a cabo la localización. Si no se obtiene, devuelve TIMEOUT.

maximumAge

Entero. Si la ubicación que hay almacenada en la caché del navegador tuvo lugar hace menos de los milisegundos especificados, usará esta ubicación. Si no encuentra nada antes de estos milisegundos, obtiene la posición de nuevo.

```
var param = {enableHighAccuracy: true, timeout: 10000,
maximumAge: 60000};

navigator.geolocation.getCurrentPosition(mostrar, error, param);
```

Título III: Obtener ubicación cada cierto tiempo

Para obtener la ubicación del usuario cada cierto tiempo, nos basta con sustituir el método **getCurrentPosition()** por el método **watchPosition()** , que acepta los mismo parámetros que el método anterior (ver Título II).
Este método lo que hace es llamarse en bucle con un tiempo de delay igual al valor del atributo **maximumAge** del objeto **config**.

*No es recomendable indicar sólo el parámetro **exito**, sino **errores** y **config** también.

```
var param = {enableHighAccuracy: true, timeout: 10000,
maximumAge: 60000};
navigator.geolocation.watchPosition(exito, error, param);

function exito(posicion){
	alert("Estas en la altura " + posicion.coords.altitude);
}
function error(er){
	alert(er.message);
}
```

clearWatch(id)

El bucle que hemos creado anteriormente lo podemos detener con el método **clearWatch(id).** Donde en **id** especificamos la variable que devuelve el método **watchPosition()** , a modo de identificador:

```
var param = {enableHighAccuracy: true, timeout: 10000,
maximumAge: 60000};
var id = navigator.geolocation.watchPosition(exito, error, param);
```

```
function exito(e){
        alert("Estas en la altura " + e.coords.altitude);
        navigator.geolocation.clearWatch(id);
}
```

Título IV: Mostrar posición en un mapa

Para mostrar nuestra posición en un mapa, utilizaremos la API Google Maps:

```
navigator.geolocation.getCurrentPosition(mostrar, error, param);

function mostrar(posicion){
        var section = document.getElementByTagName("section");

        var url =
"http://maps.google.com/maps/api/staticmap?center=" +
posicion.coords.latitude + "," + posicion.coords.longitude + "," +
"&zoom=12&size=400x400&sensor=false&markers=" +
posicion.coords.latitude + "," + posicion.coords.longitude;

        section.innerHTML = "<img src='" + url + "'>";
}
```

*Si no funciona, consultar API.

Capítulo V: Web Storage

Título I: Teoría

Generalmente, hay dos formas de guardar información:

- En un servidor: conectándose a él y guardando la información en una BBDD.
- Mediante Cookies: las cookies son ficheros de texto que se almacenan en la carpeta de archivos temporales de internet de nuestro navegador y nos permiten almacenar pequeñas cadenas de texto. Esta información es temporal.

Estas dos formas pueden no resultar útiles en todos los casos, por eso se inventó la API Web Storage:

La API Web Storage nos permite almacenar información.
Esta información se puede guardar como:

- **sessionStorage** : Almacenamiento temporal, sólo se guarda durante la sesión de la ventana activa (similar a las cookies). Cada **sessionStorage** es perteneciente a cada pestaña, por lo que no se podrá acceder a él desde una pestaña externa.
- **localStorage** : Almacenamiento permanente, información disponible aún en un navegador cerrado. Esta información se guarda en el dispositivo del usuario.

La información la podemos guardar con **setItem()** y la podemos recuperar con **getItem()** (o con una sintaxis alternativa).
La información se almacena en “variables” o ítems, los cuales llevan un protocolo de clave-valor.

Título II: Escritura con sessionStorage

1. Para guardar datos usando **sessionStorage**, debemos identificar los datos que usaremos como clave-valor y guardarlos en variables.

2. Posteriormente, ejecutamos el método **setItem(clave, valor)** al que le pasamos nuestras dos variables.

```
var clave, valor;
código;
sessionStorage.setItem(clave, valor);
```

Sintaxis alternativa

Existe otra sintaxis diferente para guardar elementos en **sessionStorage**. Esta sintaxis consiste en tratar al objeto **sessionStorage** como si de un array se tratara:

```
var clave, valor;
código;
sessionStorage[clave] = valor;
```

Título III: Lectura con sessionStorage

1. Para leer los datos que tenemos almacenados en formato clave-valor, debemos conocer la clave del ítem que queremos extraer del almacenamiento.
2. Una vez que conozcamos la clave, ejecutamos el método **getItem(clave)** , que devuelve el valor de la clave especificada.

```
var clave, valor;
código;
valor = sessionStorage.getItem(clave);
```

Sintaxis alternativa

Al igual que con la escritura, existe otra sintaxis para leer elementos de un **sessionStorage**, que consiste en tratar a este objeto como un array (esta forma nos permite recorrer el objeto con un bucle for):

var clave, valor;
código;
valor = sessionStorage[clave];

Acceso mediante índices numéricos

El objeto **sessionStorage** es una especie de array, y para poder recorrerlo con un bucle for, necesitamos especificar índices numéricos.

El método **key(i)** nos devuelve la clave del índice especificado en **i**. (los elementos de un **sessionStorage** se empiezan a almacenar desde la posición 0, al igual que un array convencional).
De esta forma, podemos recuperar toda la información que haya guardada en un **sessionStorage**:

```
var valores = “”;
for(int i=0; i<sessionStorage.length; i++){
        var clave = sessionStorage.key(i);
        valores += sessionStorage[clave] + “ ”;
}
```

Título IV: Eliminar datos de sessionStorage

Tenemos dos formas de eliminar datos: eliminar un dato concreto o eliminar todos los datos:

clear()

Elimina todos los datos del objeto: **sessionStorage.clear();**

removeItem(clave)

Elimina el ítem cuya clave es igual a **clave**:

```
var clave = valor;
sessionStorage.removeItem(clave);
```

Título V: localStorage

localStorage funciona de forma idéntica, a **sessionStorage**.

El almacenamiento de esta información depende de qué navegador estemos usando: algunos guardan la información en una BBDD MySql Lite, en ficheros XML, en ficheros .dat o ficheros con un formato propio.

Capítulo VI: IndexedDB

La API IndexedDB nos permite acceder a una BBDD especial (ver título I) para almacenar grandes cantidades de información.
IndexedDB es la alternativa a la API Web Storage , ya que esta la usamos cuando Web Storage se nos queda corta y necesitamos almacenar más información.

Título I: Teoría

IndexedDB es una BBDD que contiene diferentes "almacenes" (lo que en una BBDD convencional serían tablas) que a su vez contienen diferentes objetos (lo que en otra BBDD serían registros), que contienen diferentes propiedades.
Todos los objetos del mismo almacén no tienen por qué compartir los mismos campos de clase, sino que unos objetos pueden tener unos campos que otros objetos no tengan.

Las BBDD IndexedDB se guardan en el dispositivo del usuario. La ruta depende del navegador en cuestión; La ruta de Chrome es la siguiente:
C:\Users\usuario\AppData\Local\Google\Chrome\User Data\Default\IndexedDB

Pasos para crear o conectarse a una BBDD

1. Crear la BBDD con el atributo **indexedDB** y el método **open()**.
2. Crear almacenes con el método **createObjectStore()** .
3. Crear la transacción para agregar los elementos con **transaction()** .
4. Agregar elementos con el método **add()**.
5. Mostrar los resultados de la consulta.
    5.1. Abrir un cursor con el método **openCursor()** .

Título II: Creación de BBDD

1. Para crear una BBDD (lo cual se suele hacer cuando se carga la página web, a la que se identifican los componentes HTML), debemos crear una variable que igualamos a **indexedDB.open(nombre)** , al que le pasamos como **nombre** el nombre de nuestra BBDD (esta se puede llamar como queramos, si el nombre especificado ya existe se conectará a esta BBDD, si no existe, la creará).
2. Si la solicitud de creación de BBDD que hicimos en el punto 1 funciona (**solicitud.onsuccess**) , tenemos que almacenar en una variable el resultado de esta, que será la que nos permita acceder a la BBDD.
3. Posteriormente, debemos comprobar si la BBDD ha sido creada (**solicitud.onupgradeneeded**) o si ya existía de antes. Si ha sido creada, debemos crear un almacén para posteriormente almacenar objetos dentro de él (también debemos repetir el código del punto 2). Esto lo hacemos con el método **createObjectStore(nombre, key)** al cual le indicamos el nombre del almacén en **nombre** y cual va a ser el campo clave del almacén en **key**.

```
var solicitud = indexedDB.open("registro_civil");
solicitud.onsuccess = function(e){
        bbdd = e.target.result;
}
solicitud.onupgradeneeded(e){
        bbdd = e.target.result;
        bbdd.createObjectStore("ciudadanos", {keyPath: "dni"});
}
```

Título III: Almacenamiento en BBDD

1. Para agregar información a nuestra BBDD, primero debemos recabar en variables la información que queremos almacenar.

2. Posteriormente, tenemos que crear un objeto transacción que será el devolvido por el método **transaction(nombre, modo)** aplicado sobre el objeto BBDD que obtuvimos en el punto 2 del título I.
Este método recibe dos parámetros: un array con el nombre del almacén al que queremos acceder y el modo en el que queremos abrirlo.
*Existen tres modos de abrir una BBDD: **“readonly”, “writeonly”, “readwrite”.** (Este último es el más usado).
3. Después, almacenamos en una variable el objeto transacción al que le aplicamos el método **objectStore(nombre)** , en el que volvemos a indicar el nombre del almacén.
4. Almacenamos en el almacén un objeto que creamos dentro del método **add** , donde guardamos los datos en modo de **campo:valor** , indicando en **campo** el nombre del campo de clase, como si estos se crearan dinámicamente.
Todo esto lo guardamos en una variable.
A esta variable le podemos añadir un evento de tipo **“success”**, el cual se ejecutará si el almacenamiento se completó correctamente.

```
var nombre, apellido, anos, dni;
código;

var transaccion = bbdd.transaction([“ciudadanos”], “readwrite”);
var almacen = transaccion.objectStore(“ciudadanos”);
var agregar = almacen.add({nombre: nombre, apellidos: apellido,
edad: anos, dni: dni});
agregar.addEventListener(“success”, funcion, false);
```

Título IV: Lectura de BBDD

1. Para leer la BBDD, creamos un objeto transacción que será el devolvido por el método **transaction(nombre, modo)** aplicado sobre el objeto BBDD que obtuvimos en el punto 2 del título I.

Este método recibe dos parámetros: un array con el nombre del almacén al que queremos acceder y el modo en el que queremos abrirlo.
*Existen tres modos de abrir una BBDD: **“readonly”, “writeonly”, “readwrite”.** (Este último es el más usado, aunque en este caso conviene usar el primero).
2. Después, almacenamos en una variable el objeto transacción al que le aplicamos el método **objectStore(nombre)** , en el que volvemos a indicar el nombre del almacén.
3. Creamos un cursor con el método **openCursor()** sobre el objeto creado en el punto 2.
4. Declaramos que si la creación del objeto anterior se realizó exitosamente (ponemos a la escucha de un evento de tipo “**success**”), creamos una función que muestre o nos devuelva los datos.
5. Creamos una función oyente que recibe un parámetro. Dentro de esta función, volvemos a crear el objeto cursor.
6. Creamos un bucle que se ejecute mientras que el cursor tenga valores.
7. Leemos el almacén en la posición actual del cursor con nomenclatura de puntos (indicando el nombre del cursor **.value.** el nombre del campo al que queremos acceder).
8. Pasamos al siguiente valor del cursor con el método **continue()**.

```
var transaccion = bbdd.transaction([“ciudadanos”], “readonly”);
var almacen = transaccion.objectStore(“ciudadanos”);
var cursor = almacen.openCursor();
cursor.addEventListener(“success”, mostrar, false);

function mostrar(e){
        var cursor = e.target.result;
        var resultado;
        while(cursor){
```

```
            resultado += cursor.value.nombre + " " +
cursor.value.apellidos + ", " + cursor.value.edad + " con DNI: " +
cursor.value.dni;

            cursor.continue();
        }
}
```

Capítulo VII: File

La API File nos permite interactuar con los archivos del usuario en local procesando información de los archivos.
La API File Directories & System nos permite crear un sistema de archivos.
La API File Writer nos permite escribir dentro de archivos.
Todas estas APIs pertenecen a la API File.

Título I: Teoría

Para leer los archivos del usuario, usaremos una interfaz llamada **FileReader,** perteneciente a la API File, la cual nos devuelve un objeto con varios métodos que nos permiten obtener la información del archivo en concreto:

readAsText(archivo)

Intenta interpretar cada byte del archivo como texto, codificado en utf-8.
Podemos hacer que use otro codificado que pueda mostrar tildes, como el iso-8859-1, este se le indicamos como segundo parámetro: **readAsText(archivo, “iso-8859-1”);**

readAsBinaryString(archivo)

Devuelve la información de cada byte como una sucesión de números entre 0 y 255. Útil para mover archivos.

readAsDataUrl(archivo)

Devuelve el archivo como una cadena de tipo url codificada en base 64. Útil para codificar información y moverla codificada o para visualizar imágenes.

readAsArrayBuffer(archivo)

Devuelve los datos en una especie de array con un buffer en binario.

También necesitaremos un lector para leer la información devuelta por cada uno de los métodos.

Título II: Obtener objeto File

Una forma de obtener un objeto File es usando un **input** de HTML de **type="file"**, al que le aplicamos un evento **"change"** donde accederemos a la característica **files**:

```
boton = document.getElementById("selector");
boton.addEventListener("change", seleccionado, false);

function seleccionado(e){
	var archivos = e.target.files;
	var archivo = archivos[0];
}
```

Título III: Lectura de un objeto File

Una vez que ya tengamos nuestro objeto File, podemos leerlo con los métodos vistos en el Título I, creando un lector con la interfaz **FileReader**.
Posteriormente, hemos de hacer que nuestro lector desencadene un evento **"load"**.
Este evento hará que se muestre el contenido del archivo:

```
var lector = new FileReader();
lector.readAsText(archivo, "iso-8859-1");
lector.addEventListener("load", mostrar, false);

function mostrar(e){
	var contenido = e.target.result;
```

```
    alert("El contenido del archivo es: " + contenido);
}
```

Lectura de imágenes

Si el archivo seleccionado es una imagen (consultar Título IV), al leerla con **readAsText()** nos mostrará un texto sin sentido, por lo que para visualizar la imagen, debemos usar el método **readAsDataUrl()** .

```
var lector = new FileReader();
lector.readAsDataUrl(archivo);
lector.addEventListener("load", mostrarImg, false);

function mostrarImg(e){
    var contenido = e.target.result;
    contenedor.innerHTML = "<img src="" + contenido + "">
}
```

Título IV: Propiedades de archivos

Los objetos de tipo File tienen unas propiedades que nos permiten obtener información sobre los archivos:

type

Devuelve el tipo de archivo.

*Para comparar con algo:

```
if(archivo.type.match(/image/)){
    alert("Es una imagen");
}
```

name

Devuelve el nombre del archivo.

size

Devuelve el tamaño del archivo en bytes.

fullPath

Devuelve la ruta del archivo.

Título V: Creación de archivos “virtuales”

Para crear archivos y directorios, necesitaremos de la API Files Directories & System.
Debemos seguir una serie de pasos para crear archivos y directorios:

pasos

1. Petición al navegador para solicitar acceso al disco duro (ya que por defecto se encuentra desactivado) (*).
2. Creamos el sistema de archivos, el cual sólo accesible desde la aplicación, ya que es una especie de sistema de archivos virtual.
3. Creamos o abrimos todos los archivos o directorios deseados (si no existe, se creará, si existe, se abrirá).
4. Ya podemos realizar diferentes acciones como listar, mover, copiar, eliminar los archivos y directorios o incluso escribir y agregar contenido a estos con la API FIleWriter.

*Existen dos tipos de espacio en el disco: los temporales, que se conservan hasta que la pestaña de la página web es cerrada; y los persistentes, que se conservan hasta que el usuario los elimina.

creación

1. Utilizamos el método **requestQuota(espacio, funcion)** sobre el objeto **navigator.tipoNavegadorPersistentStorage** para un tipo de espacio persistente o **navigator.tipoNavegadorTemporaryStorage** para un tipo de espacio temporal (*).

El método recibe dos parámetros: el espacio en bytes que ocupará en nuestro disco duro (con 5MB va bien) y la función a la que queremos que llame si el proceso del método se ejecuta con éxito.
2. Creamos el sistema de archivos dentro de la función a la que llama el método **requestQuiota** del punto 1. Para ello, indicamos el método **tipoNavegadorRequestFileSystem(tipoEspacio, espacio, exito, error)** sobre el objeto **window**.
El método recibe cuatro parámetros:

- **tipoEspacio** : El tipo de espacio que queremos crear (PERSISTENT o TEMPORARY)
- **espacio** : El espacio en bytes que ocupará en nuestro disco duro (el mismo que en el punto 1).
- **exito :** La función a la que llamará si se ejecuta con éxito (a la que se le pasa un objeto **FileSystem**).
- **error :** La función a la que llamará si algo falla (a la que se le pasa un objeto error (ver Capítulo IV, Título II, errores)).

2.1. Creamos el sistema de archivos guardando la propiedad **root** del objeto **FileSystem** creado en el punto 2 en una variable para usar esta variable como espacio.
3. Para crear el archivo, ejecutamos el método **getFile(nombre, obj, exito, error)** sobre la variable espacio creada en el punto 2.1.
Esta función recibe cuatro parámertos:

- **nombre :** El nombre del archivo a crear.
- **obj :** Un objeto que creamos en el proceso para que si el archivo no existe, lo cree y si existe que no de fallo.
  *Si en vez de crear un archivo, queremos abrirlo, especificar **null**.
- **exito :** La función a la que llamará si se ejecuta con éxito (a la que se le pasa el objeto **File** creado).
- **error :** La función a la que llamará si algo falla (a la que se le pasa un objeto error (ver Capítulo IV, Título II, errores)).

*3. Para crear un directorio, especificar **getDirectory()** en lugar de **getFile()** (no cambia nada más, especificar mismo parámetros).

*En tipoNavegador , indicar la abreviatura correspondiente al navegador a utilizar (ver Apuntes CSS, Capítulo II, Título II, Tipos de degradado, especificar navegador):
navigator.webkitPersistentStorage ,
navigator.mozPersistentStorage …

```
navigator.webkitPersistentStorage.requestQuota(5*1024*1024, acceso);

function acceso(){
	window.webkitRequestFileSystem(PERSISTENT, 5*1024*1024, crearSist, error);
}

function crearSist(sistema){
	espacio = sistema.root;
}

function crear(){
	espacio.getFile(nombreArchivo, {create: true, exclusive: false}, exito, error);
}

function exito(){alert("Archivo creado exitosamente");}
function error(){alert("Algo ha fallado");}
```

El resto del apartado Files fue saltado.
(vídeos 57-62).

Capítulo VIII: Ajax

La tecnología Ajax (Asynchronous JAvascript Xml) , nos permite realizar peticiones al servidor y que las respuestas actualicen la página web (no crear otra página, sino cambiar algún componente).
Google Maps es un ejemplo de Ajax.

Ventajas

- Permite mostrar nuevo contenido HTML sin recargar la página.
- Permite enviar un formulario y mostrar una respuesta inmediata.
- Permite hacer login en una página sin abandonarla.
- Permite mostrar resultados automáticos de encuentras.
- Permite buscar información en una BBDD y ver los resultados inmediatamente.

Elementos en Ajax

- Navegador: Donde el usuario interactúa.
- Servidor web: Que ofrece las respuestas del navegador.
- JavaScript: Que procesa las respuestas del servidor.
- Objeto XMLHttpRequest (XHR): Puente utilizado para enviar información al servidor y recibirla.

Pasos para enviar petición y procesar el resultado

1. Crear instancia de **XMLHttpRequest** (constructor default).
2. Utilizar el método **open()** de **XHR** para especificar qué petición se enviará (GET o POST) y dónde irá dentro del servidor, especificando la página alojada en el servidor..
    2.1. **GET**: La información de la petición se envía utilizando la URL del navegador como medio.
    Se utiliza para recibir y procesar respuestas del servidor.
    2.2. **POST**: Utilizado para actualizar información del servidor (actualizar BBDD). Se utiliza el método **send()** de **XHR**.
3. Crear una función JavaScript que gestione la respuesta.

3.1. Modificar el DOM (componentes HTML).

4. Recibir la respuesta , que puede estar formada por diferentes piezas de información:

- Estado de la respuesta (código).
- Texto de respuesta.
- XML de respuesta.
- HTML de respuesta.
- JSON de respuesta.

get

get envía la información usando la barra de URL, por lo que la información enviada es visible para el usuario.

*No utilizar **get** cuando se estén usando contraseñas, ya que el usuario las verá.

get se suele utilizar con consultas, ya que sólo podemos enviar 2500 caracteres máximo.

post

post envía la información de forma transparente, por lo que el usuario nunca ve que la información es enviada. Utilizar con contraseñas.

post se suele utilizar para realizar cambios en el servidor.

Título I: Consulta de archivo HTML con JQuery

load(archivoHtml)

load() es una función perteneciente a JQuery que nos simplifica bastante la tecnología Ajax, siempre y cuando solicitemos un archivo HTML.

load() carga el archivo html especificado en **archivoHtml** dentro del componente HTML sobre el que se ejecuta este método:

$("#contenedor").load("archivo.html");

*Este método no funciona en local, sólo con protocolos http o https , por lo que para probar la página, tendremos que subirla a un servidor (junto con el archivo que indicamos en **archivoHtml**).

**Podemos indicar que no nos cargue la página entera, sino sólo lo que haya dentro de un div en específico (llamado contendorP):
$("#contenedor").load("archivo.html #contendorP");

Título II: Consultas a servidor con JQuery

Para realizar una consulta con JQuery, primero tenemos que crear el literal donde irán los datos y posteriormente usar el método **get()** o **post()** para enviar y procesar los datos.

$.get(url, datos, procesa)

Donde **url** es la página PHP (u otras tecnologías) que procesa los datos enviados en **datos** (dentro de un literal) y **procesa** es la función JavaScript que procesa la respuesta de la página PHP (este parámetro es opcional). Esta función recibe como parámetro los datos que el servidor devuelve.

$.get("pagina.php", datos, procesa);
function procesa(devuelta){codigo;}
(ver Título VI)

$.post(url, datos, procesa)

Donde **url** es la página PHP (u otras tecnologías) que procesa los datos enviados en **datos** (dentro de un literal) y **procesa** es la función JavaScript que procesa la respuesta de la página PHP (este parámetro es opcional). Esta función recibe como parámetro los datos que el servidor devuelve.

$.post("pagina.php", datos, procesa);
function procesa(devuelta){código;}
(ver Título VI)

$.ajax($parametros)

Puede usarse como **$.get()** y como **$.post()** . Permite especificar múltiples parámetros.

Dentro de él , indicar unos parámetros dentro de unos corchetes en formato **clave:valor** :

```
$.ajax({
        url: "pagina.php",
        type: "POST",
        data: datos,
        success: function(e){alert(e);}
})
```

(ver Título VI)

Título III: Excepciones en consultas con JQuery

Para capturar estos errores, tenemos que indicar la función **error()** detrás de la función **get()** , **post()** , **load()** o la que estemos usando en ese momento:

```
$.get("pagina.php", datos, funcion).error(fallo);
function fallo(){alert("Algo ha fallado");}
```

error(funcion)

Ejecuta la función **funcion** especificada en el caso de que ocurra un error.

Título IV: Objetos JSON

JSON es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera un formato independiente del lenguaje.

Para crear un objeto JSON, tenemos que crear un objeto JavaScript, indicando entre corchetes un formato de **clave:valor** y separar estos formatos por comas. Dentro de un objeto JSON podemos indicar arrays (estructura similar al JSON pero con corchetes) u otros objetos JSON:

```
json = {
        “nombre” : “Juan”,
        “edad” : 18,
        “apellido” : “García”,
        “hermanos” : true,
        “coches” : [“Renault”, “Nissan”],
        “casa” : {“duplex” : false, “piso” : true}
}
```

*Si los datos están dentro de un formulario y estamos dentro de un evento del formulario, podemos ahorrarnos todo esto e incluir la función:

```
var datos = $(this).serialize();
```

Que enviará los datos con el valor de cada campo de texto y el nombre será el atributo HTML **name** de cada campo.

Acceder a JSON

Para leer un objeto JSON , primero tenemos que convertirlo a un objeto JavaScript con la función **JSON.parse(json)** y luego simplemente indicar el objeto JSON y seguir la nomenclatura del punto con sus valores:
(la función recibe un archivo json):

```
function leer(json){
        var obj = JSON.parse(json);

        obj.nombre;             //Juan
        obj.hermanos;           //true
        obj.coches[1];          //Nissan
```

```
    obj.casa.duplex;        //false
}
```

Esta parte del curso HTML fue saltada (vídeos 63-68).
La parte del curso de JavaScript Google Maps también (vídeo 75)

Capítulo IX: Web Worker

La API Web Worker nos permite crear programación multihilo o multiproceso.

Para crear una solución multiproceso, tenemos que crear dos archivos .js distintos, que trabajarán a la vez. El secundario de estos dos es el llamado **worker**.
Estos dos se comunican mediante mensajes.
Cuando el principal necesita del **worker**, este envía un mensaje al **worker**. el worker lo procesa y se lo devuelve.
*Los mensajes siempre deben ser de tipo texto.

**Sólo es necesario indicar el archivo principal en el archivo HTML, ya que el worker se especifica en este archivo principal.

Título I: Enviar y recibir mensajes al worker

1. Para crear un worker, tenemos que crear un objeto **Worker** dentro del archivo .js principal.
Esto lo hacemos con el constructor **new Worker(file)** , al que le indicamos en **file** la ruta del archivo .js del código del worker.

trabajador = new Worker(“archivo.js”);

2. Posteriormente, ponemos al objeto worker a la escucha de un evento de tipo **“message”** , que será el que se ejecuta cuando el fichero recibe un mensaje de otro fichero.
Creamos una función que se ejecutará cuando el archivo principal requiera del worker.
Creamos la función oyente del message la cual recibe un parámetro, que se corresponde con el mensaje enviado por el worker.

```
trabajador.addEventListener("message", recibir, false);

function enviar(mensaje){
        trabajador.postMessage(mensaje);
}

function recibir(e){
        alert(e.data);
}
```

Título II: Recibir y enviar mensajes desde el worker

1. Lo primero, es capturar el mensaje enviado por el archivo principal. Esto lo hacemos con un evento de tipo **"message"**.
Posteriormente, indicamos que procese el mensaje.

```
addEventListener("message", recibir, false);

function recibir(e){
        mensaje = e.data;
        procesar(mensaje);
}
```

2. Después de que el worker procese el mensaje, tenemos que devolver el mensaje. Esto lo hacemos indicando el método **postMessage**:

```
function procesar(mensaje){
        código;

        postMessage(mensaje);
}
```

Título III: SharedWorker

Podemos crear SharedWorkers, que son Workers compartidos por diferentes páginas HTML.
Esto nos permite que un principal envíe un mensaje al worker y que todos los que estén conectados al SharedWorder reciban el mensaje y que cada uno lo trate de forma independiente.

1. La creación del trabajador es idéntica, salvo que ahora la clase es **SharedWorder**:

trabajador = new SharedWorker("archivo.js");

2. Al igual que en el título I, lo ponemos a la escucha de un evento de tipo **"message"** pero sobre la propiedad **port** , ya que tratamos de conectarnos a él por un puerto. Y hacemos que empiece el envío de mensajes.

```
trabajador.port.addEventListener("message", recibir, false);
trabajador.port.start();

function enviar(mensaje){
    trabajador.port.postMessage(mensaje);
}

function recibir(mensaje){
    alert(mensaje.data);
}
```

---

3. Para recibir el mensaje desde el worker, lo ponemos a la escucha de un evento de tipo **"connect"** , que se ejecutará cuando un archivo se

conecta a él mediante un puerto. Este evento devuelve un array de una posición con el puerto de los archivos de los diferentes documentos HTML, por lo que para identificar a cada archivo HTML, almacenamos en un array todos los puertos.
Ponemos a este puerto a la escucha de otra función:

```
puertos = new Array();
addEventListener("connect", conectar, false);

function conectar(e){
        puertos.push(e.ports[0]);
        e.ports[0].onmessage = enviar;
}

function enviar(mensaje){
        msg = mensaje.data;
        código;     //Este código manipula el mensaje que se envía.

        for(var i=0; i<puertos.length; i++){
                puertos[i].postMessage(mensaje.msg);
        }
}
```

Capítulo X: History

La API History nos permite acceder al historial para moverse con las flechas de adelante y atrás.
Si la página en la que deseamos ir hacia atrás es la misma página que dónde se cometieron los cambios pero simplemente se actualizó, la pulsar atrás saldremos de esta, por lo que manipulando el historial con la API History, conseguiremos establecer cuál es la página anterior a esa.
Estas son llamadas "url falsas", las cuales nos permiten navegar por la información previamente cargada.
Estas las podemos crear con los métodos **pushState()** o **replaceState()**.

back()

Equivalente a pulsar el botón de atrás.

forward()

Equivalente a pulsar el botón de alante.

go(pasos)

Va hacia atrás tantas veces como números negativos haya (-3 = back(); back(); back()) y hacia delante tantos números positivos haya (2 = forward(); forward()).

length

Devuelve el número de las páginas que han sido visitadas.

window.history.pushState(estado, titulo, url)

Crea una nueva url falsa y la pone en la barra de url.
Se puede prescindir de los dos primeros (indicar **null**)(no recomendable).

- **estado :** "id" que identifica cada página (entero).

- **titulo :** No relevante, especificar **null** .
- **url :** Especificar la terminación de la url (pagina2.html por ejemplo).

window.history.pushState(null, null, “pagina2.html”);

window.history.replaceState(estado, titulo, url)

Establece el estado de la página actual.

- **estado :** “id” que identifica cada página (entero).
- **titulo :** No relevante, especificar **null** .
- **url :** Especificar la terminación de la url (no es necesario indicarla).

window.history.replaceState(null, null);

state

Aplicar sobre el objeto que devuelve los eventos de tipo “popstate”.
Devuelve el estado de la página (un entero).

window.addEventListener(“popstate”, funcion, bool)

Se dispara cada vez que las url falsas cambian.
Devuelve un objeto a la función oyente, al que se le puede aplicar la propiedad **state**.

Capítulo XI: API Offline

La API Offline nos permite trabajar offline en el navegador con la ayuda de la caché para que cuando recuperemos la conexión esta se actualice.
Esto lo hacemos con la ayuda de un archivo manifiesto, con extensión **.manifest** .
Para que cuando la conexión con el servidor se corte, dispongamos de este archivo para terminar nuestro trabajo y no depender del servidor web.

Esta parte fue saltada (vídeo 73).

Capítulo XII: JQuery

JQuery es una librería de JavaScript que nos ayuda en la creación de tareas frecuentes a la hora de programar como:

- Seleccionar elementos HTML.
- Agregar nuevo contenido a HTML de forma dinámica.
- Ocultar y mostrar elementos HTML de una web.
- Validar formularios.
- Creación de efectos visuales vistosos.

Las librerías Prototype, Mootools, Dojo y Yahoo User Interface nos ofrecen funciones similares.

*Todos los signos de dólar "$" vistos a continuación se pueden sustituir por **JQuery** , aunque es recomendable indicar el dólar por eficacia a la hora de programar.

Librería

En JavaScript, una librería es un documento .js lleno de funciones que nos ofrecen diferentes funcionalidades.
Las librerías van evolucionando, añadiendo y eliminando funcionalidades.

Ventajas

- JQuery es una librería pequeña (sobre ~100 kB).
- Es "CSS friendly", necesitamos CSS para usar esta librería.
- Ha sido ampliamente testeada.
- Es gratis.
- Tiene una gran comunidad de soporte.
- Presenta numerosos plugins que realizan las tareas más frecuentes.

Agregar JQuery a página web

Para agregar JQuery a nuestra página web, añadimos la siguiente etiqueta dentro del **head** de nuestro archivo HTML después del código CSS:

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

Después de agregar nuestra librería JQuery, tenemos que agregar nuestro código en otra etiqueta **script** , la cual tendrá el siguiente código:

```
$(document).ready(function(){
    //Aquí va todo el código de la página web.
});
```

O su versión abreviada:

```
$(function(){
    //código
})
```

Lo que hace la siguiente función es esperar a que todos los elementos HTML estén cargados antes de ejecutar la primera línea de JavaScript.
Es equivalente a (usar el que sea):

```
window.addEventListener("load", function(){
    //código
}, false);
```

DOM

DOM o Document Object Model , hace que el navegador recuerde una jerarquía entre todos los elementos HTML y que memorice las propiedades de cada elemento HTML.
En el DOM , a cada etiqueta se le llama nodo.
El DOM está establecido por w3c.

Título I: Objeto JQuery

Podemos crear un objeto JQuery identificando componentes HTML desde código JavaScript.
Para crear este objeto, especificaremos el símbolo del dólar, unos paréntesis, unas comillas y un selector CSS que seleccione los componentes HTML que deseemos. Todo esto lo almacenaremos en una variable:

var arrayBotones = $(“.botones”);
var componentePrincipal = $(“.principal”);
var parrafos = $(“p”);
var negritas = $(“p strong”);
…

*Los objetos creados son objetos JQuery, por lo que no pueden usar métodos de otro tipo de objetos, como el método **addEventListener()**.
Todos los métodos vistos a continuación sólo son aplicables sobre objetos **JQuery .

next(selector)

Funciona de forma similar a un objeto JQuery con un selector descendiente.
Hace referencia a un componente hijo especificado en **selector** (mediante selectores de id, clases o etiqueta) que se encuentra dentro del componente padre sobre el que se ejecuta este método.
Resulta muy útil cuando no sabemos cual es el componente padre, ya que este será un **this** :

$(this).next(“p”);

Con esto, hacemos referencia a la etiqueta(s) p que estén dentro del componente HTML actual.

width(x)

Hace que el componente sobre el que se ejecuta el método tenga **x** píxeles de ancho. Especificar un número entero.
*Si se especifica sin height, establece un alto automático y lógico.

height(x)

Hace que el componente sobre el que se ejecuta el método tenga **x** píxeles de alto. Especificar un número entero.
*Si se especifica sin width, establece un ancho automático y lógico.

color(c)

Hace que el componente sobre el que se ejecuta el método tenga en el atributo CSS **color** el valor especificado en **c** .

val(v)

Hace que el componente sobre el que se ejecuta el método tenga en el atributo HTML **value** el valor especificado en **v** .
*Si no se especifica ningún parámetro, devuelve el valor del atributo **value** , con lo que podemos manipular formularios.

focus()

El componente sobre el que se ejecuta recibe el foco.

Título II: Características especiales

JQuery nos provee de algunas características especiales para facilitar nuestro código:

Loops automáticos

Podemos ejecutar ciertas funciones para un grupo de componentes sin usar bucles, ya que los usará JQuery de forma automática.
Queremos esconder a todos los párrafos de nuestra página cuando algo suceda:

$(“p”).hide();

Funciones encadenadas

Podemos ejecutar funciones en cadena que se ejecutarán una detrás de otra sobre un componente.
Queremos dotar a un label con id=texto de propiedades CSS y que haga un fade-in cuando algo suceda:
$(“#texto”).width(500).height(500).text(“Hola!”).color(“#0FF”).hide().fadeIn(2000);

*Si el código nos parece demasiado largo, lo podemos dividir en varias líneas:

```
$(“#texto”).width(500)
.height(500)
.text(“Hola!”)
.color(“#0FF”)
.hide()
.fadeIn(2000);
```

Título III: Agregar contenido HTML

Disponemos de 6 funciones que nos otorga JQuery para agregar contenido a nuestra página web desde código JavaScript:

html()

Devuelve el código HTML que encontramos dentro del componente al que se le aplica:
(HTML): ** Hola, como estas Juan **
(JavaScript): **alert($(“p”).html());**
Salida: Hola, como estas Juan

html(codHtml)

Borra todo el código HTML que hay dentro del componente al que se le aplica y lo sustituye por el código especificado en **codHtml** :
(HTML): ** Hola, como estas Juan **
(JavaScript):
$("p").html("Hola, como estas Alberto");

append(codHtml)

Realiza la misma función que **html(codHtml)** pero no borra el código, sino que añade el código HTML **codHtml** al final del último código HTML dentro del contenedor sobre el que se aplica esta función.
(HTML): ** Hola, como estas Juan **
(JavaScript):
$("p").append(", espero que estés bien");

prepend(codHtml)

Realiza la misma función que **append(codHtml)** pero en vez de añadir el código al final del componente sobre el que se aplica esta función, lo añade al principio.

before(codHtml)

Realiza la misma función que **prepend(codHtml)** , pero en vez de añadir el código dentro del componente sobre el que se aplica, el código se añade antes del componente sobre el que se aplica.

after(codHtml)

Realiza la misma función que **append(codHtml)** pero en vez de añadir el código dentro del componente sobre el que se aplica, el código se coloca fuera del componente sobre el que se aplica y después de este.

text() / text(texto)

Funciona igual que **html() / html(codHtml)** pero no es capaz de usar etiquetas HTML, sólo devuelve o modifica el texto de las etiquetas.

Título IV: Eliminar y reemplazar contenido

remove()

Elimina el componente HTML sobre el que se aplica.

replaceWith(codHtml)

Elimina el componente HTML sobre el que se aplica y coloca en su lugar el código HTML especificado en **codHtml** .

Título V: Cambiar componente HTML de clase

Podemos cambiar las clases a las que pertenecen los componentes HTML para así dotar de un estilo personalizado a ciertas clases desde código CSS o cualquier otra funcionalidad.
*Los componentes HTML pueden pertenecer a varias clases a la vez:
** **

addClass(clase)

Añade el atributo class = **clase** al componente sobre el que se ejecuta el método.

removeClass(clase)

Elimina el atributo class = **clase** del componente sobre el que se ejecuta el método.

toggleClass(clase)

Añade el atributo class = **clase** al componente sobre el que se ejecuta el método si el componente no tenía el atributo class = **clase** o elimina el atributo class = **clase** del componente sobre el que se ejecuta si el componente tenía el atributo class = **clase** .

Título VI: Leer y Cambiar atributos CSS

Para leer y cambiar atributos de CSS, necesitaremos usar la función **css()** , la cual tiene tres sobrecargas diferentes.
Todas estas sobrecargas se ejecutan sobre la referencia JavaScript del componente HTML al que queremos acceder.

css(propiedad)

Devuelve el valor de la propiedad CSS indicada en **propiedad** :
var ancho = componente.css("width");

css(propiedad, valor)

Establece a la propiedad CSS indicada en **propiedad** el valor **valor**:
componente.css("width", "500px");

css({propiedad:valor, propiedad:valor, …})

Establece las propiedades y los valores CSS en formato **propiedad:valor** . Se pueden especificar tantas propiedades a la vez como se desee:
componente.css({"width":"500px", "height":"200px", "color":"#000"});

Título VII: Leer, Cambiar y eliminar atributos HTML

Para leer, cambiar y eliminar atributos HTML desde JavaScript, disponemos de diferentes métodos que ejecutaremos sobre los diferentes objetos **JQuery** :

attr(atributo)

Devuelve el valor del atributo HTML especificado en **atributo** .

attr(atributo, valor)

Establece el valor **valor** al atributo HTML **atributo** :

componente.attr(“src”, “http://ruta.jpg”);

removeAttr(atributo);

Elimina el atributo HTML especificado en **atributo** .

Título VIII: Función each(funcion)

Establece que por cada componente especificado (componente(s)) sobre el que se ejecuta, ejecutará una vez la función especificada en **funcion.** Estas funciones suelen ser funciones anónimas.
Queremos añadir ciertas propiedades a todos los párrafos de nuestra página:

```
$(“p”).each(function(){
        $(this).color(“#F00”).background(“#FFF”);
});
```

*En este caso, con **this** hacemos referencia al objeto **p** que estamos “analizando” en ese momento.
Esto lo podríamos hacer sin la función **each() , debido a lo explicado en Título II, Loops automáticos ; aunque para otros casos (como en los que tengamos que especificar condicionales), la función **each()** es de vital importancia.

Título IX: Eventos

Para añadir una función a un componente HTML debemos crear una referencia a ese componente con un objeto JQuery (ver Título I) y posteriormente ejecutar sobre él un método específico (*) , donde especificamos la función que se ejecuta cuando se desencadena el evento (**).

De esta forma, si queremos añadir un evento cuando se pulsa sobre cualquier botón:

function pulsar(){<u>código</u>;}
$(“button”).click(pulsar);

*El nombre de los métodos para añadir eventos se corresponde con el tipo de evento que queramos añadir (ver Capítulo I, Título III).
**Podemos especificar una referencia a una función como en el ejemplo o bien crear una función anónima.
***En el caso de que creemos una función anónima, podemos hacer referencia al objeto **$(this)** , el cual hará referencia al elemento sobre el que estamos ejecutando el método del evento:
****Todas estas funciones oyentes pueden recibir el parámetro del evento siempre y cuando reciban un parámetro.

```
$(“button”).click(function(){
        $(this).text(“Ya no me pulses m&aacute;s”);
});
```

hover(funcionIn, funcionOut)

Se encarga de manejar el evento **mouseover** de forma más sencilla:
Cuando el ratón se introduce en el componente HTML sobre el que se ejecuta el método, se ejecuta la función **funcionIn** y cuando este sale del componente, la función **funcionOut** .
Funciona de forma similar a los métodos vistos en este título.
Es como combinar las funciones **mouseover** y **mouseout** en una.

toggle(funcionOn, funcionOff)

Se encarga de manejar el evento **click** de forma más sencilla:
Cuando el ratón hace un click por primera vez sobre el componente HTML sobre el que se ejecuta el método, se ejecuta la función **funcionOn** . Cuando se realice el siguiente click , se ejecutará la función **funcionOff** .
Funciona de forma similar a los métodos vistos en este título.

Es como añadir una función de On/Off a cualquier botón de forma sencilla.

bind(tipoEvento, param, funcion)

Similar al método **addEventListener()** pero para objetos JQuery. Añade la función **funcion** como oyente del evento (o eventos, en cuyo caso separar con un espacio) de tipo **tipoEvento** (ver Capítulo I, Título III) y le pasa los parámetros **param** :

```
$("button").bind("click", num, llamar);
function llamar(e){
        alert(e.data);
}
```

En este ejemplo, **e.data** es igual a la variable **num** .

Título X: Efectos

Disponemos de diferentes funciones que nos ofrece la librería JQuery para crear efectos y animaciones:

hide()

Esconde el componente sobre el que se ejecuta el método, haciéndolo invisible.

show()

Muestra el componente sobre el que se ejecuta el método, haciéndolo visible.

toggle()

Alterna entre **hide()** y **show()** dependiendo del estado del componente. Dota al componente de un efecto de redimensionamiento.

fadeIn(ms)

Hace que el elemento sobre el que se aplica este método vaya apareciendo poco a poco y que esta aparición tarde los milisegundos especificados en **ms** (convidar con **fadeOut()** o con **hide()**).
*Si no se especifican parámetros, se ejecuta con un tiempo default.
Es equivalente a **show() pero con retraso.

fadeTo(ms, opacity)

Realiza la misma función que **fadeIn(ms)** , pero en vez de llegar al 100% de opacidad (**opacity = 1**) , se queda en el número especificado en opacity (el cual debe estar entre 0 y 1, siendo 0 totalmente transparente).

fadeOut(ms)

Hace que el elemento sobre el que se aplica este método vaya desapareciendo poco a poco y que este desvanecimiento tarde los milisegundos especificados en **ms**.
*Si no se especifican parámetros, se ejecuta con un tiempo default.
Es equivalente a **hide() pero con retraso.

fadeToggle(ms)

Alterna entre **fadeIn(ms)** y **fadeOut(ms)** dependiendo del estado del componente.
*Si no se especifican parámetros, se ejecuta con un tiempo default.

slideDown(ms)

Hace que aparezca el elemento sobre el que se aplica con un efecto de deslizamiento desde arriba hasta abajo.
El efecto total tarda **ms** milisegundos.
*Si no se especifican parámetros, se ejecuta con un tiempo default.

slideUp(ms)

Hace que desaparezca el elemento sobre el que se aplica con un efecto de deslizamiento desde abajo a arriba.
El efecto total tarda **ms** milisegundos.
*Si no se especifican parámetros, se ejecuta con un tiempo default.

slideToggle(ms)

Alterna entre **slideDown()** y **slideUp()** .
*Si no se especifican parámetros, se ejecuta con un tiempo default.

Título XI: Animaciones

JQuery nos permite animar componentes HTML utilizando código similar a CSS.
Para ello, usaremos el método **animate()** y las propiedades CSS que queramos usar para animar a nuestro componente.

*Sólo se podrán usar aquellas propiedades CSS que reciban valores numéricos, a excepción de los colores hexadecimales.
Las propiedades CSS se mantienen constantes, a excepción de las formadas por varias palabras y un guión, en cuyo caso se usará la nomenclatura **camelCase :

width → width **margin-left → marginLeft**

animate(propiedades, tiempo)

Le da al componente CSS sobre el que se aplica las propiedades CSS especificadas en **propiedades** (las cuales suelen ser un incremento o decremento de variable numéricas) en un formato de **propiedad:valor** dentro de unos corchetes.
Puede recibir un segundo parámetro opcional que indica el tiempo que tarda en completar la animación el milisegundos:

componente.animate({marginLeft:"+=150px"}, 1000);

En este ejemplo, el componente **componente** se alejará 150px de la izquierda (irá hacia la derecha) y tardará 1 segundo en hacerlo.

*Si el valor especificado lleva algún dígito que no sea numérico, especificar el valor entre comillas, de lo contrario no hace falta.
**Permite recibir un tercer parámetro, el cual es una referencia a una función que se ejecutará una vez terminada la animación.
***Entre los milisegundos y la función también podemos indicar otro parámetro que hace referencia a un efecto que tendrá la animación, aunque para esto es necesario importar **jquery easing plugin** .

stop()

No procesa la cola de eventos de un evento (si se pulsa 100 veces mientras ocurre una animación se anima una única vez).
Para usarlo, especificarlo delate del método **animate()**:
componente.stop().animate({width:"200px"}, 1000);

delay(ms)

Establece un tiempo de delay en milisegundos.
Útil en medio de funciones encadenadas:
componente.fadeIn(1000).delay(5000).fadeOut(1000);
Aparece durante un segundo, se queda estático durante 5 y desaparece en un segundo.
*No detiene el hilo de ejecución.

Título XII: Plugins

Se pueden descargar diferentes "extensiones" de librerías JQuery que nos permiten realizar diferentes funciones.
En Google existen multitud de páginas en las que descargar estos plugins.

Añadir plugins

Para añadir plugins, tenemos que hacer lo mismo que vimos en **Agregar JQuery a página web** , indicando los plugins después de la librería JQuery.

Título XIII: Preloader de imágenes

Con imágenes de rollover (etiquetas **img** que cambian de imagen al pasar el ratón por encima), se puede dar el caso de que el navegador tarde mucho en cargar la segunda imagen y que el efecto se vea interrumpido.
Para evitar esto, podemos usar un preloader el cual no muestra ninguna imagen hasta que ambas estén cargadas en la caché.

Para crear un preloader, es tan simple como crear un objeto **Image** en vez de especificar las urls e indicar la propiedad **src** de este objeto para especificar la url:

(Código sin preloader):
$(“#imagen”).attr(“src”, ruta);

(Código con preloader):
var imagen = new Image();
imagen.src = ruta;
$(“#imagen”).attr(“src”, imagen.src);

Título XIV: Imágenes con FancyBox

FancyBox es una librería que proporciona un visor de imágenes que deja el navegador en segundo plano, mostrando una imagen como principal atractivo.

Para usar esta librería, o nos descargamos el archivo y lo importamos como url local o hacemos referencia a él dentro de un servidor como url global.

fancybox(propiedades)

Se ejecuta sobre cada imagen que queremos agregar al fancybox.
Recibe unas propiedades que se establecen en formato **clave:valor** , dentro de unos corchetes.
Se pueden especificar tantas propiedades como se desee, separadas por comas.
Todas las propiedades.

Una vez especificado este método, ya tenemos nuestra galería.

*Podemos añadir etiquetas **<iframe>** dentro de un fancybox. Para ello, consultar vídeo 55.