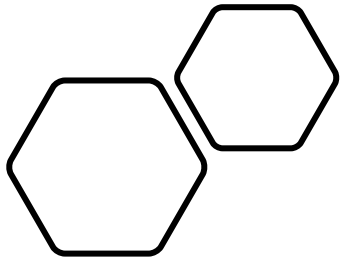
Curso 2023-2024



UD9: SVG, JSON y JavaScript

Lenguajes de marcas y sistemas de gestión de información


JavaScript

¿Qué es JavaScript?

- JavaScript (JS) es un potente lenguaje de programación de scripts que puede aplicarse sobre documentos HTML.
- La principal función es aplicar **control** a estructuras y apariencias.
- Con JS podemos hacer infinidad de **pequeños añadidos** a nuestra página web, o usarlo de forma avanzada para crear **grandes funcionalidades**.
- JS es un lenguaje de programación en el lado del **cliente**. No obstante, con **node.js** podemos usarlo como lenguaje de programación en el lado del servidor.

¿Qué es JavaScript?

- JS, por sí mismo permite realizar gran cantidad de programas y funcionalidades, no obstante…
- Existen muchas **bibliotecas y APIs** que nos permiten realizar tareas y funciones ya establecidas.
- La selección de la biblioteca o API que necesitemos es una parte fundamental a la hora de realizar la planificación de una nueva funcionalidad o aplicación.

Primeros pasos

- Comentarios:
  - Los comentarios en JS se realizan con /**/ o bien con //:

```
/* Esto es un comentario, que puede abarcar varias líneas. */
// Esto también es un comentario, pero solo de una línea.
```

- Las líneas de código deberían acabar con punto y coma (;).
  - En algunos casos no es estrictamente necesario, pero sí recomendable, ya que ayuda a la interpretación automática del código.
  - Hay también algunos casos en los que no deberíamos acabar con **; (por ejemplo, al abrir o cerrar llaves {}).**

Primeros pasos

- Salida de texto por la **consola de JS** (es necesario abrirla en el navegador desde las herramientas para desarrolladores):

```
console.log(mensaje);
```

Variables

Variables

- Para almacenar los datos que vamos a usar durante la ejecución del script creamos **variables**.
- JS diferencia entre mayúsculas y minúsculas, por lo que la variable `a` es diferente a la variable `A`.
- Para usar una variable primero hay que declararla:
  - Usando **var**: `var nombreVariable`
  - Usando **let**: `let nombreVariable`
- Es absolutamente recomendable usar **let**, puesto que el ámbito de las variables declaradas con **var** puede ocasionar problemas si no se controla adecuadamente.

Variables

- Realmente no es **obligatorio** declarar las variables: podemos asignar valor a una variable **sin necesidad de declararla** previamente:

```
miNombre= 'Ninguno';
```

- El problema es que, si no declaramos las variables, éstas internamente se declaran **globalmente**, mientras que si usamos **var** o **let** se declaran en el **ámbito de ejecución en curso**.

Ámbito de las variables: ejemplo

```
function imprimirNombre(){
  console.log('Dentro de función, mi nombre es ' + miNombre);
}

function principal(){
  miNombre='George';
  imprimirNombre();
}

let miNombre= 'Jerry';

principal();

console.log('Hola, mi nombre es ' + miNombre);
```

Variables

- Otra diferencia es que las variables que declaramos **se “mueven” automáticamente antes de ejecutar cualquier otro código de ese bloque** mientras que una variable no declarada **no existe** antes de que el código que le asigna valor se ejecute.

- Por último, las variables declaradas son una propiedad no configurable (durante su ámbito de ejecución en curso), mientras que las variables no declaradas son configurables (y podemos actuar sobre ellas).

Variables

- La recomendación general es declarar siempre las variables.

- Elevación de variables (*Hoisting*):
  - Puesto que las variables declaradas se crean antes de ejecutar cualquier otro código, lo ideal es que ajustemos la declaración de las variables al inicio de su ámbito de ejecución para evitar lo que denominamos la “elevación de variables”, que es un comportamiento en el que parece que podemos usar una variable antes de declararla.

Variables

- Sintaxis:

```
var nombreVariable;
var nombreVariable = valor;
var nombreVariable1 = valor1, nombreVariable2 = valor2;

let nombreVariable;
let nombreVariable = valor;
let nombreVariable1 = valor1, nombreVariable2 = valor2;
```

Variables

- JS es un lenguaje **débilmente tipado** y dinámico, lo que quiere decir que al declarar una variable no la asociamos con un tipo de dato en concreto. Además, podemos asignarle un valor de cualquier tipo sin ningún problema.

```
let miVariable = 3;
miVariable = 'Cadena de texto';
```

Variables

- Tipos de datos en JS:
  - String (cadena de texto).

    ```
    let miVariable = 'DCLIENT';
    ```
  - Número.

    ```
    let miVariable = 3.4;
    ```
  - Booleano.

    ```
    let miVariable = false;
    ```
  - Array.

    ```
    let miVariable = [3, false, 'DCLIENT'];
    ```
  - Objeto.

Operadores

Operadores

- Son símbolos que operan sobre dos elementos (*inputs*) y produce un resultado (*output*).
- Pueden ser de naturaleza aritmética o lógica.
  - Operadores aritméticos
    - Adición (**+**): suman dos números **o concatenan dos cadenas de texto**.
    - Sustracción (**-**): resta el segundo operando del primero.
    - Multiplicación (******): multiplica los dos operandos.
    - División (***/***): divide el primero operando por el segundo.
    - Resto (**%**): devuelve el resto de la división de los operandos.
    - Potencia (**): eleva el primer operando al segundo.
    - Operador de asignación (**=**): asigna un valor a una variable.

Operadores

- Operadores lógicos (devuelven true/false)
  - Igualdad (**===**) (**==**): devuelve true si los operandos son iguales.
  - Diferencia (**!==**) (**!=**): devuelve true si los operandos son diferentes.
  - Negación (**!**): invierte un valor booleano true/false false/true.
  - Menor que (<): devuelve true si el primer operando es estrictamente menor que el segundo.
  - Mayor que (>): devuelve true si el primer operando es estrictamente mayor que el segundo.
  - Menor o igual que (<=): devuelve true si el primer operando es menor o igual que el segundo.
  - Mayor o igual que (>=): devuelve true si el primer operando es mayor o igual que el segundo.

Condicionales

Condicionales

- Un condicional es una estructura de control que ejecuta una serie de instrucciones en función de que una condición se cumpla o no.

- Sintaxis:

```
if (condición)
  Instrucción1;
else
  Instrucción2;
```

Condicionales

- En caso de que queramos ejecutar varias instrucciones en el supuesto de cumplirse (o no) una condición deberemos usar llaves para contener todo el código:

```
if (condición){
  Instrucción1;
  Instrucción2;
}
else{
  Instrucción3;
  Instrucción4;
}
```

Condicionales

- Podemos establecer una estructura de evaluación de condiciones compleja si anidamos varias sentencias if...else de la siguiente forma:

```
if (condición1)
  Instrucción1;
else if (condición2)
  Instrucción2;
else if (condición3)
  Instrucción3;
else
  InstrucciónAlternativa;
```

Condicionales

- En caso de que no queramos establecer instrucciones especiales para el caso de que no se cumpla la condición (else) usamos simplemente if:

```
if (condición1)
  Instrucción1;
```

Condicionales

- Otra forma de realizar un mecanismo de control en función de los posibles valores de una variable es con la construcción switch-case:

```
switch (expresión){
  case valor1:
    instruccion1;
    instruccion2;
    break;
  case valor2:
    instruccion3;
    break;
  default:
    instruccionPorDefecto;
}
```

Condicionales

- Cuando queremos establecer más de una condición tenemos dos opciones:
  - Anidar varios condicionales.
  - Relacionar varias condiciones con operadores lógicos:
    - AND lógico: **&&**
    - OR lógico: **||**
    - NOT lógico: **!**

Bucles

Bucles

- En JS, al igual que en otros lenguajes de programación, hay varias formas de indicar que queremos que un conjunto de instrucciones se repitan un cierto número de veces o hasta que se cumpla una determinada condición.

1. **while**
   - Con while indicamos que mientras se cumpla una condición se repetirá la instrucción (o instrucciones) contenidas en su interior. Es importante tener la certeza de que la condición debe poder dejar de cumplirse.
   - Sintaxis:

```
while(condición)
  instrucción;
```

```
while(condición){
  instrucción;
}
```

Bucles

2. do…while

- Mismo principio que con while, pero evalúa la condición al final de cada iteración del bucle, de forma que siempre se ejecutará al menos una vez.
- Sintaxis:

```
do
  instrucción;
while (condición);
```

```
do{
  instrucción;
}while (condición);
```

Bucles

3. **for:**
   - Al igual que while, permite crear un bucle.
   - De forma general, se incluyen 3 elementos en la declaración del for:
     1. Inicialización: una expresión que se ejecuta al inicio de cada iteración.
     2. Condición: la condición que vamos a exigir en cada iteración para continuar con el bucle.
     3. Terminación: una expresión que se ejecuta al final de cada iteración. Normalmente la usamos para incrementar un contador.
   - Sintaxis:

```
for (inicialización; condición; terminación)
  instrucción;
```

```
for (inicialización; condición; terminación){
  instrucción1;
  instrucción2;
}
```

Funciones

Funciones

- Las funciones son trozos de código que encapsulamos, de forma que podemos invocarlas en otras partes sin tener que escribir el código que contienen.
- Hay ciertas funciones ya realizadas e implementadas en el navegador.
- También podemos crear nuestras propias funciones.
- Sintaxis:

Creación:

```
function miFuncion(parámetro1, parámetro2){
  instrucciones;
  return resultado; /* ESTO ES OPCIONAL */
}
```

Invocación:

```
miFuncion(parámetro1, parámetro2);
```

Objetos

Objetos javascript

- Para definir un objeto en JS usamos la siguiente sintaxis:

```
let miObjeto= {
    atributo1: valor1,
    atributo2: valor2,
    //… resto de atributos
    metodo1: function(){
        //aquí realizo las acciones del método 1;
    }
};
```

Arrays

Arrays

- Los arrays o vectores son elementos que contienen diferentes valores, almacenados según posición.
- Hay varias formas de definirlos:
  1. Introducir los elementos uno a uno:

```
let miArray=new Array(); // Se desaconseja usar este constructor
// Es mejor usar let miArray=[];
miArray[0]=19;
miArray[1]=38;
miArray[2]=53;
console.log(miArray[0]+','+miArray[1]+','+miArray[2]); // 19,38,53
```

  2. Introducir varios elementos a la vez:

```
let otroArray=['hola',true,21];
```

Arrays

- Podemos ver los valores individuales dentro del array o bien podemos mostrar todo el contenido:

```
console.log(miArray[0]); // Imprime 19
console.log(miArray[1]); // Imprime 38
console.log(miArray[2]); // Imprime 53
console.log(miArray); // Imprime [ 19, 38, 53 ]
```

Arrays

- No solo podemos introducir diferentes tipos de datos en los arrays, también podemos introducir:
  - Objetos.
  - Funciones.
  - Otros arrays (de esta forma podemos crear matrices).
- Además, los arrays en JS son objetos (¡pero objetos especiales!), por lo que poseen:
  - Atributos
  - Métodos

Los arrays son objetos especiales porque sus elementos están indexados, si no tenemos cuidado y creamos el array como un objeto no tendremos acceso a muchos de sus métodos y atributos.

Arrays

- Para acceder a la dimensión o tamaño del array usamos la propiedad length:

```
console.log(miArray.length); // Imprime 3
```

- Así, para acceder al primer elemento del array usaremos:

```
miArray[0]; // Aunque primero deberíamos hacer una comprobación…
```

- Mientras que para acceder al último usamos:

```
miArray[miArray.length-1];
```

- Para recorrer todos los elementos de un array podemos:
  - Crear un bucle for.
  - Usar el método forEach.

Arrays

- Usando un bucle for:

```
for (let i=0; i<miArray.length; i++){
    console.log(miArray[i]);
}
```

- Usando el método forEach() de la clase Array:

```
function imprimirElementos(elementoActual){
  console.log('El elemento actual es:'+ elementoActual);
}

miArray.forEach(imprimirElementos);
```

Arrays

- Podemos añadir nuevos elementos a un array de varias formas:
  - Usando el método push:

```
let miArray=new Array();
miArray[0]=19;
miArray[1]=38;
miArray[3]=53;

miArray.push(45);

console.log(miArray);
//[ 19, 38, <1 empty item>, 53, 45 ]
```

Arrays

- Podemos añadir nuevos elementos a un array de varias formas:
  - Directamente en la posición que indiquemos:

```
let miArray=new Array();
miArray[0]=19;
miArray[1]=38;
miArray[3]=53;

miArray[miArray.length]=45;

console.log(miArray);
//[ 19, 38, <1 empty item>, 53, 45 ]
```

Arrays

- ¿Cómo puedo saber si una variable que recibo es un array?
  - Con el operador typeof de JS se puede obtener el tipo de una variable, por ejemplo:

```
let a = 5.7;
console.log(typeof a); //number
```

  - El problema es que si lo aplicamos a un array nos dice que es de tipo 'object', puesto que, en el fondo, un array es un objeto.
  - Podemos usar el método isArray() de la clase Array para comprobar si una variable es un array. Hay que tener en cuenta que los navegadores más antiguos pueden no soportar este método.

```
Array.isArray(miArray); //true si es un array, false si no lo es.
```

Integración HTML y JS

Integrar nuestro código en HTML

- Para que lo visto hasta ahora cobre verdadera utilidad, tenemos que ser capaces de interactuar con JS con algo más que la consola del navegador (console.log()).
- Lo primero es conseguir vincular el código JS con el documento HTML. Para esto hay dos opciones:
  - Incrustando nuestro código dentro de la etiqueta <script></script>:

```
<script type="text/javascript">
    alert('Hola Mundo');
</script>
```

  - Invocando un archivo .js externo:

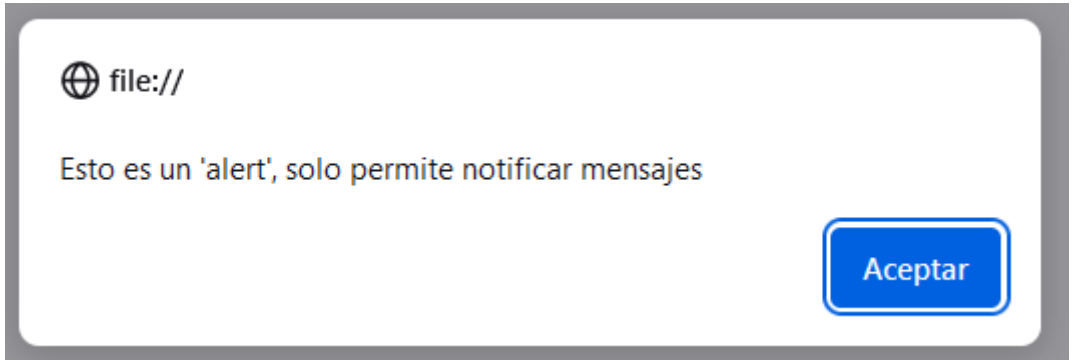
```
<script src="miScript.js"></script>
```

  - Podemos incluir tantas fuentes .js externas como queramos. Generalmente estas etiquetas irán dentro de <head>, pero no es estrictamente necesario.

Entrada y salida de texto

- alert: sirve para mostrar un mensaje de texto al usuario en una ventana emergente.
  - Sintaxis:

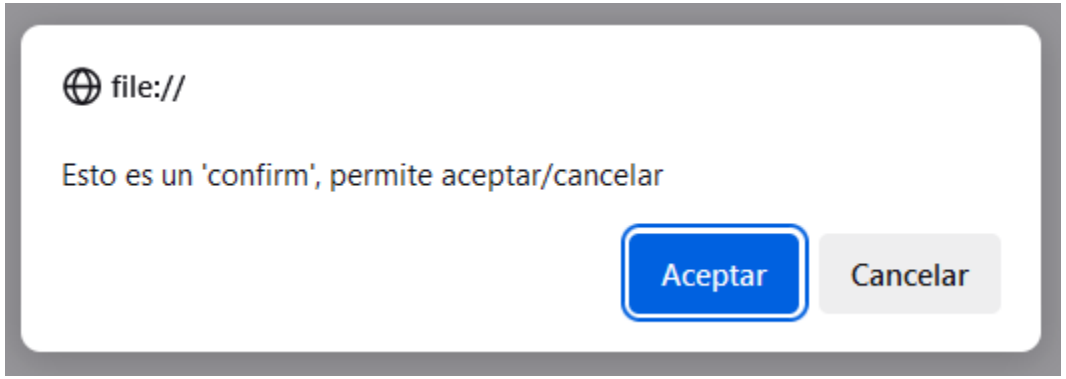
```
alert('mensaje para el usuario');
```



Entrada y salida de texto

- confirm: sirve para mostrar un mensaje de texto al usuario en una ventana emergente, que se puede aceptar o cancelar. La respuesta del usuario la podemos recoger en una variable.
  - Sintaxis:

    ```
    variable= confirm('mensaje para el usuario');
    ```
  - Se devuelve true si se pulsa en aceptar y false en caso contrario.

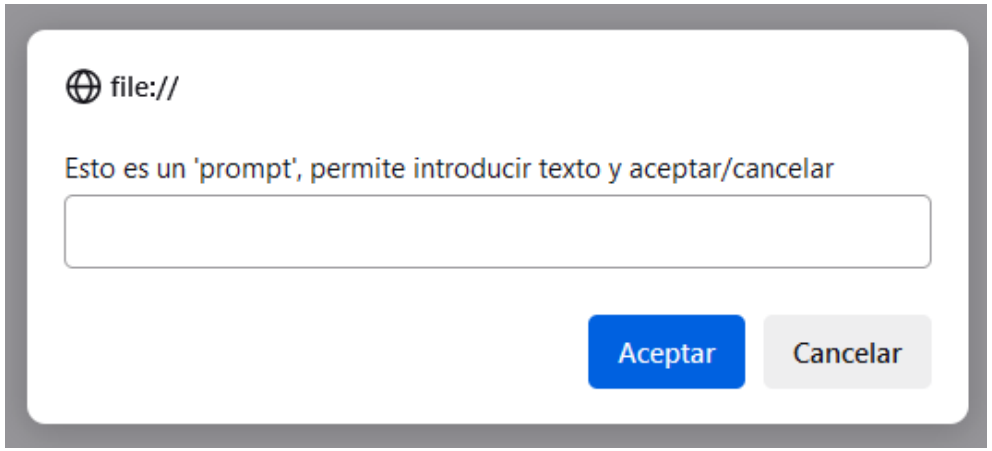


Entrada y salida de texto

- prompt: sirve para solicitar al usuario una entrada textual en una ventana emergente. Por defecto, la variable que recoge la entrada será un String.
  - Sintaxis:

    ```
    variable = prompt ('introduce algún texto');
    ```

  - Si el usuario no introduce nada:
    - Al pulsar en aceptar nos devuelve una cadena vacía.
    - Al pulsar en cancelar nos devuelve null.



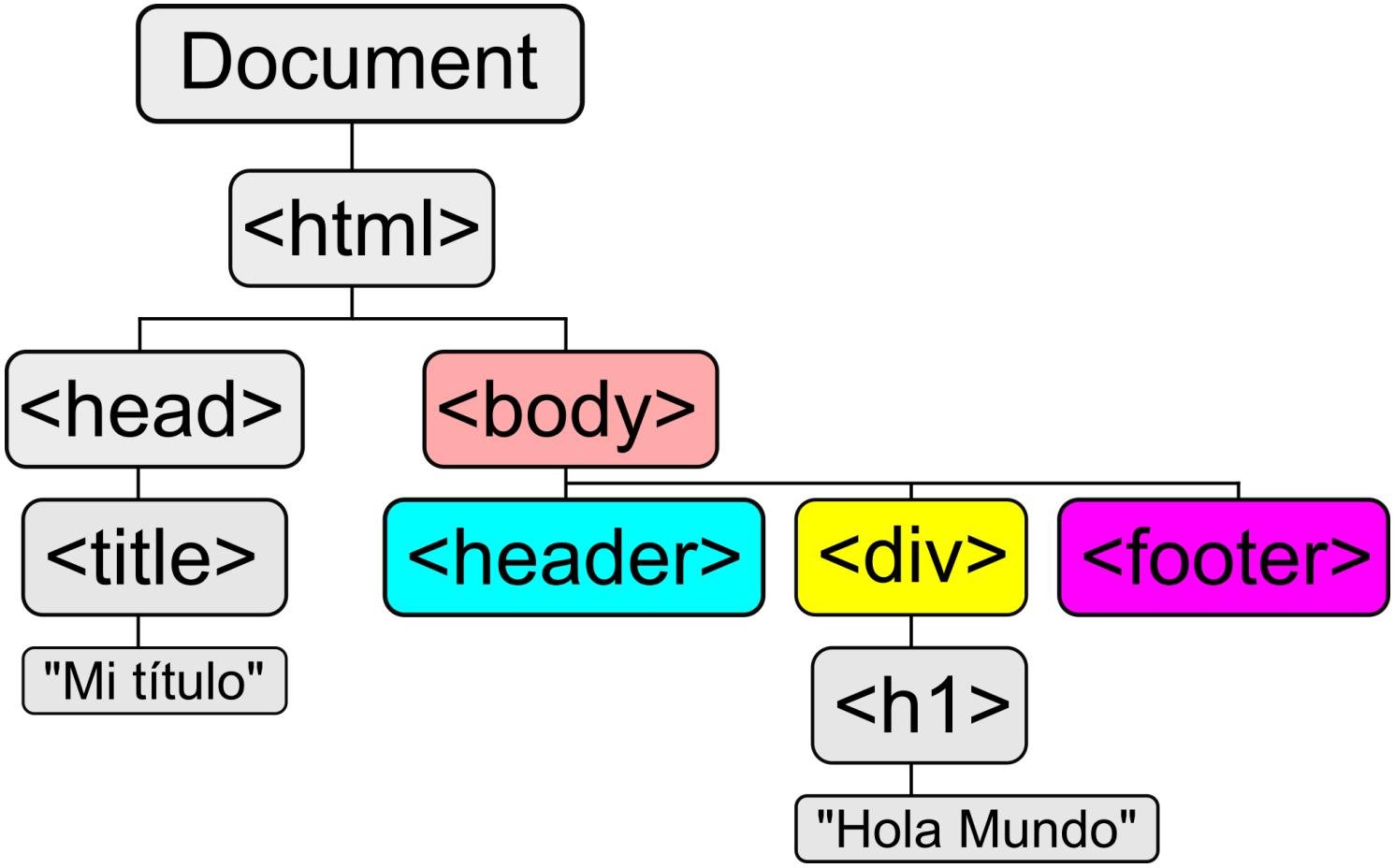
Introducción al DOM

Introducción al DOM

- DOM es el acrónimo de *Document Object Model*, es decir, modelo de objetos del documento.
- Nos proporciona una **forma estándar** de acceder a cualquier elemento del documento HTML y poder modificar sus contenidos y su aspecto.
- El DOM es un modelo de documento que se carga en el **navegador** y representa al propio documento como un árbol de nodos. Cada nodo es un elemento HTML.
- De forma que usando JS podemos acceder al DOM y, por tanto, obtener acceso a cualquier elemento HTML de la página.


<header>
<h1>Hola Mundo</h1>

<footer>


Document
<html>
<head>
<body>
<title>
<header>

<footer>
"Mi título"
<h1>
"Hola Mundo"

Introducción al DOM

- Por tanto, con el DOM podemos acceder a los elementos HTML desde JS como si de objetos se tratase. En realidad, el DOM define:
  - Los elementos HTML como **objetos**.
  - Los **atributos** de todos los elementos HTML.
  - Los **métodos** para acceder a todos los elementos HTML.
  - Los **eventos** para los elementos HTML.
- En resumen, el DOM nos proporciona una interfaz para acceder al documento HTML desde JS y poder leer y modificar tanto su contenido como su apariencia.

DOM HTML

- Por una parte, tenemos los **métodos**, que indican acciones que el objeto puede realizar o se pueden realizar sobre él, y por otra parte los **atributos**, que son variables o características de un objeto.

- Uno de los **métodos** que más vamos a usar es getElementById('ID'), que nos devuelve el elemento (objeto en este caso) HTML que tiene el identificador que le indicamos:

```
let miVariable= document.getElementById('ID_del_elemento');
```

DOM HTML

- Uno de los **atributos** que más usaremos es **innerHTML**, que nos permite obtener/modificar el **contenido textual** de elementos HTML.

```
mivariable.innerHTML = 'Hola Mundo';
```

- En caso de que el elemento sea, por ejemplo, un campo de texto <input type="text"> el atributo a usar para acceder al contenido es value:

```
mivariable.value = 'Hola Mundo';
```

DOM HTML

- Acceder a los elementos sobre los que queremos realizar alguna consulta o modificación:

```
document.getElementById('ID');
document.getElementsByTagName('etiqueta');
document.getElementsByClassName('clase');
document.querySelector('div.importante');
document.querySelectorAll('div.importante');
```

DOM HTML

- Modificar elementos HTML:

```
// Mediante modificación directa de atributos
elemento.innerHTML =  'Nuevo contenido';
elemento.attribute = 'Nuevo atributo HTML';
elemento.style.property = 'Nueva propiedad CSS';
// Mediante un método setter:
elemento.setAttribute(atributo, nuevoValor);
```

DOM HTML

- Para modificar el aspecto (en definitiva, el CSS) de los elementos usamos:

```
elemento.style.property = 'Nueva propiedad CSS';
```

- Ejemplo:

```
miVariable.style.backgroundColor = 'blue';
```

- La lista completa de propiedades CSS la podéis encontrar en:
  https://www.w3schools.com/jsref/dom_obj_style.asp

Eventos

Eventos

- Ya sabemos acceder a los elementos HTML y modificarlos.
- Lo que necesitamos ahora para poder establecer una verdadera interacción es poder responder a las acciones del usuario.
- Los **eventos** son la forma que tiene JS de detectar acciones del usuario y responder en consecuencia.
- Hay acciones que no están directamente relacionadas con las acciones del usuario (por ejemplo, que la página se cargue).
- Profundizaremos en los eventos en sesiones posteriores, de momento veremos los más destacados para poder empezar a trabajar con JS.

Eventos

- ¿Cómo establezco un evento?
  - Hay varias formas, siendo la más inmediata la escritura directa en la etiqueta HTML que va a reaccionar:

```
<input type="button" id="boton" value="evento" onclick="accion()"/>
//Otra forma
document.getElementById('id').onclick = function() {
    alert('Hola!');
}
//Otra forma…
document.getElementById('id').onclick = nombreFuncion;
//Y otra forma más…
elemento = document.getElementById('id').addEventListener('click', nombre
Funcion);
```

Eventos

- Algunos eventos útiles:
  - *onclick*: se realiza click sobre el elemento.
  - *onload*: se carga la página.
  - *onchange*: se realiza cualquier cambio sobre el elemento y se cambia el foco.
  - *onmouseover*: el puntero entra en el área del elemento.
  - *onomouseout*: el puntero del ratón ha salido del área del elemento.
  - *onfocus*: el elemento está seleccionado.
  - *onblur*: el elemento deja de estar seleccionado.

Uso del DOM para creación de elementos

Creación de nodos

1. Crear un nuevo elemento.

```
elem= document.createElement('elementoHTML');
```

2. Añadirle contenido, bien con la propiedad innerHTML o bien con:

```
texto= document.createTextNode('texto');
elem.appendChild(texto);
```

3. Agregar atributos al elemento (si se considera necesario):

```
elem.setAttribute('atributoHTML',valor_atributo);
```

4. Agregar el elemento al documento. En este caso hay varias opciones. Si lo que queremos es simplemente añadir el nodo como último hijo de <body>usamos:

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

Seleccionar la posición de inserción

- Cuando creamos un nuevo elemento, generalmente no es para añadirlo al final del documento como hijo de éste, sino que queremos insertarlo en una posición en concreto, en relación con el resto de elementos del documento. En este aspecto, puede ser:
  - Hijo del documento, pero insertado antes de otros hijos.
  - Hijo de un elemento, insertado como último hijo.
  - Hijo de un elemento, insertado como primer hijo.
  - Hijo de un elemento, insertado antes de otros hijos, pero no como primero.
- Hay varios métodos que nos permiten establecer la posición de inserción de un nuevo nodo.

Seleccionar la posición de inserción

- Lo primero es obtener el nodo respecto al cual vamos a insertar el nuevo nodo, puede ser hijo o hermano de éste. Para este fin, usamos alguno de los métodos de selección de elementos que ya conocemos:
  - `nodoReferencia = document.getElementById('id');`
- Una vez que tenemos la referencia, insertaremos el nodo bien como hijo o como hermano de éste. Si lo que queremos es añadirlo como hijo en la última posición usaremos:
  - `nodoReferencia.appendChild(nuevoNodo);`

Seleccionar la posición de inserción

- Si lo que queremos es insertar el nuevo nodo antes del nodo de referencia podemos usar:
  - `nodoReferencia.insertBefore(nuevoNodo);`
- El método `insertAdjacentElement(posición,nodo)` es el que más control nos va a proporcionar a la hora de insertar un elemento, puesto que nos permite determinar la posición relativa:
  - Insertar el nodo antes del nodo referencia: `'beforebegin'`
  - Insertar el nodo como primer hijo de nodo referencia: `'afterbegin'`
  - Insertar el nodo como último hijo de nodo referencia: `'beforeend'`
  - Insertar el nodo después del nodo referencia: `'afterend'`

Gestión de nodos

- Cuando queremos obtener información sobre la estructura de un nodo del DOM podemos usar los siguientes métodos y propiedades de los nodos:
- **childElementCount**: propiedad que contiene el número de nodos hijos.
- **childNodes**: colección de los nodos hijos, incluyendo nodos de texto.
- **children**: colección de los nodos hijos, excluyendo nodos de texto.
- **cloneNode**: crea una copia de un nodo.
- **firstChild**: contiene el primer nodo hijo.
- **lastChild**: contiene el último nodo hijo.
- **nextSibling**: contiene el siguiente nodo hermano.
- **parentNode**: contiene el nodo padre.

HTMLCollection

HTMLCollection: propiedades y métodos

- Las colecciones HTMLCollection tienen unas propiedades y métodos comunes:
  - Propiedad **length**: contiene el número de elementos de la colección.
  - Método **item(**'posición'**)**: retorna el elemento de la posición indicada.
  - Método **namedItem(**'id'**)**: retorna el elemento con el id HTML indicado.
- También es posible acceder al elemento de una posición con []:
  - miColeccion**[**3**]**: devuelve el cuarto elemento de la colección.

Las funciones setTimeout() y setInterval()

La función *setTimeout()*

- La función *setTimeout* es una función especial que espera un tiempo determinado (el que indicamos con su segundo argumento) y después realiza una acción (la que le indicamos en el primer argumento). El tiempo lo expresamos en milisegundos.
- Ejemplo de una función que mostrará un *alert* dentro de 3 segundos:

```
setTimeout(function(){ alert("Hola"); }, 3000);
```

La función *setInterval()*

- La función *setInterval* es igual a *setTimeout*, diferenciándose de ésta únicamente en que *setInterval* realiza el ciclo espera-acción de forma indefinida (*setTimeout* solo lo realiza una vez). Los argumentos que recibe son los mismos.
- Ejemplo de una función que muestra un *alert* cada 3 segundos:

```
setInterval(function(){ alert("Hola"); }, 3000);
```

¿Cómo detenemos la ejecución de *setInterval*?

- Cuando invocamos la función *setInterval*, ésta retorna un valor que podemos guardar en una variable. Este valor es un id del intervalo que hemos definido.
- Si guardamos este valor (id) en una variable podemos usarlo posteriormente como argumento de la función *clearInterval*, que detiene la ejecución del intervalo que le indiquemos.
- Ejemplo: detener la ejecución de un intervalo establecido previamente:

```
let miIntervalo = setInterval(function(){ alert("Hola"); }, 3000);
clearInterval(miIntervalo);
```

El objeto arguments

El objeto arguments

- El objeto especial *arguments* contiene una lista de los parámetros que se le proporcionan a una función, y se puede acceder desde dentro de ésta.

```
function imprimirParametros(a,b,c){
  console.log ('a: ' + arguments[0]);
  console.log ('b: ' + arguments[1]);
  console.log ('c: ' + arguments[2]);
}
```

El objeto arguments

- *arguments* no es un *Array*, pero podemos recorrerlo mediante índices numerados (empieza en el índice [0]) y saber su tamaño mediante la propiedad *length*.

```
function imprimirParametros(){
  for (let i=0;i<arguments.length;i++)
    console.log (i + ': ' + arguments[i]);
}
```

El objeto arguments

- Además, podemos usar el operador *typeof* con cada uno de los elementos de *arguments* para conocer qué tipo de dato contienen:

```
function imprimirTipoArgumentos(a,b,c){
  console.log ('a: ' + typeof arguments[0]);
  console.log ('b: ' + typeof arguments[1]);
  console.log ('c: ' + typeof arguments[2]);
}
```

Persistencia de datos

Persistencia de datos

- JS no está pensado para ser capaz de almacenar grandes cantidades de datos.
- Actualmente existen dos formas de almacenar datos en JS:
  - **Cookies:** adecuadas para almacenar pequeñas cantidades de datos (variables). Tradicionalmente se han usado para almacenar datos relativos a la identificación (login) de los usuarios.
  - **LocalStorage**: a medida que JS ha ido creciendo en uso se han implementado *LocalStorage* y *SessionStorage*, que permiten almacenar en torno a 5 MB de datos y son de acceso más sencillo.

Persistencia de datos

- `localStorage.setItem('nombre','Miguel');`
- `localStorage.getItem('nombre');`
- `localStorage.removeItem('nombre’);`

- `sessionStorage.setItem('nombre','Miguel');`
- `sessionStorage.getItem('nombre');`
- `sessionStorage.removeItem('nombre');`

Persistencia de datos

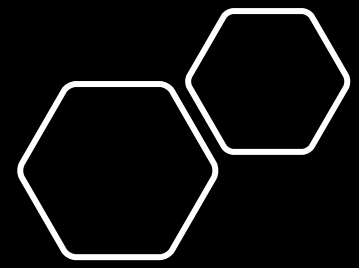
- En caso de no tener nada almacenado previamente para esa clave, el valor que retorna getItem será null.
- localStorage guarda los datos de forma persistente, mientras que sessionStorage los guarda mientras dura la sesión, lo que implica que, al cerrar el navegador o la pestaña, estos datos se pierdan.

Uso de objetos en JS

Objetos javascript

- Para definir un objeto en javascript usamos la siguiente sintaxis:

```
let miObjeto= {
    atributo1: valor1,
    atributo2: valor2,
    //resto de atributos
    metodo1: function(){
        //aquí realizo las acciones del método 1;
    }
};
```



Objetos en JS

- En JS podemos definir rápidamente un objeto que solo tenga variables (atributos) de la siguiente forma:

```
let silla = {patas:4,respaldo:true,acolchada:false,reclinable:false
,ajustable:true,altura:50};
```

Propiedad	Valor
patas	4
respaldo	Sí (true)
acolchada	No (false)
reclinable	No (false)
ajustable	Sí (true)
altura	50 (cm)

Objetos en JS

- Añadir métodos:
  - Los métodos en JS son funciones internas de los objetos que las contienen.
  - Hay que tener claro la forma en que JS trata a las funciones:

```
//Definición de la función
function saludar(){
  console.log("Hola");
}

//Invocación de la función
saludar(); //Hola
```

Objetos en JS

- Ya hemos usado funciones como setTimeout o setInterval, que como primer parámetro reciben una función, pero en este caso no añadimos los paréntesis ¿Por qué?

```
setTimeout(saludar(),500); //Error
setTimeout(saludar,500); //Correcto
```

- Porque usar los paréntesis después del nombre de la función, invoca a la función, por lo tanto…

Objetos en JS

```
setTimeout(saludar(),500); //Error
//Lo que realmente está haciendo esto es:
setTimeout("Hola",500);
```

```
setTimeout(saludar,500); //Correcto
//Para saber lo que hace, podemos usar:
console.log(saludar); // [Function: saludar]
//Por tanto, lo que realmente hace es:
setTimeout(function saludar(){console.log("Hola");},500);
```

Objetos en JS

- Otra característica de JS es que las funciones pueden tener nombre o ser anónimas:

```
setTimeout(function saludar(){console.log("Hola");},500);
//Es equivalente a:
setTimeout(function (){console.log("Hola");},500);
```

- Además, podemos guardar una función (con nombre o sin él) en una variable:

```
saludo1= function saludar(){console.log("Hola");};
saludo2= function (){console.log("Hola");};
setTimeout(saludo1,500); //Hola
setTimeout(saludo2,500); //Hola
```

Objetos en JS

- Si declaramos una función con nombre **en una variable**, no podremos acceder a ella después con el nombre de la función:

```
saludo1= function saludar(){console.log("Hola");};
setTimeout(saludo1,500); //Hola
setTimeout(saludar,500); //saludar is not defined
```

- Sí que podemos definir una función y luego asignarla a una variable sin mayores problemas:

```
function saludar(){console.log("Hola");}
let saludo1= saludar;
setTimeout(saludo1,500); //Hola
setTimeout(saludar,500); //Hola
```

Objetos en JS

- Entonces, ¿cómo definimos métodos en JS?
  - Hemos visto que podemos definir funciones anónimas y asociarlas a variables.
  - Las propiedades de los objetos son variables internas de éstos.
  - Por tanto, los métodos serán funciones anónimas internas de los objetos asignadas a variables, que serán el nombre del método.

```
let silla = {
  patas:4,
  respaldo:true,
  acolchada:false,
  reclinable:false,
  ajustable:true,
  altura:50,
  ajustar: function(alt){
    if (this.ajustable)
      this.altura=alt;
    else
      console.log("Esta silla no es ajustable");
  }
};

console.log(silla.altura); //50
silla.ajustar(70);
console.log(silla.altura); //70
```

Objetos en JS

Propiedad	Valor
patas	4
respaldo	Sí (true)
acolchada	No (false)
reclinable	No (false)
ajustable	Sí (true)
altura	50 (cm)
ajustar	`function(alt){ if (this.ajustable) this.altura=alt; else console.log("Esta silla no es ajustable"); }`

Objetos en JS

- Con esto podemos crear objetos, con atributos y métodos como si de variables complejas se tratase.
- ¿Y si necesitamos definir varios objetos del mismo tipo?
- En este caso tendremos que definir **constructores**.
- Los constructores nos permiten diseñar un “molde” para crear diferentes objetos con atributos y métodos iguales en estructura, pero que pueden tener valores diferentes.
- En los constructores podemos usar la palabra clave **this**, que hace referencia al objeto que ejecuta el código.

```
function Silla(legs, resp, acol, rec, ajus, height) {
  this.patas = legs;
  this.respaldo = resp;
  this.acolchada = acol;
  this.reclinable = rec;
  this.ajustable=ajus;
  this.altura=height;
  this.ajustar = function(alt){
    if (this.ajustable)
      this.altura=alt;
    else
      console.log("Esta silla no es ajustable");
  };
}

let unaSilla = new Silla(4,true,true,true,true,60);
console.log(unaSilla.altura); //60
unaSilla.ajustar(80);
console.log(unaSilla.altura); //80
```

Clases en JS

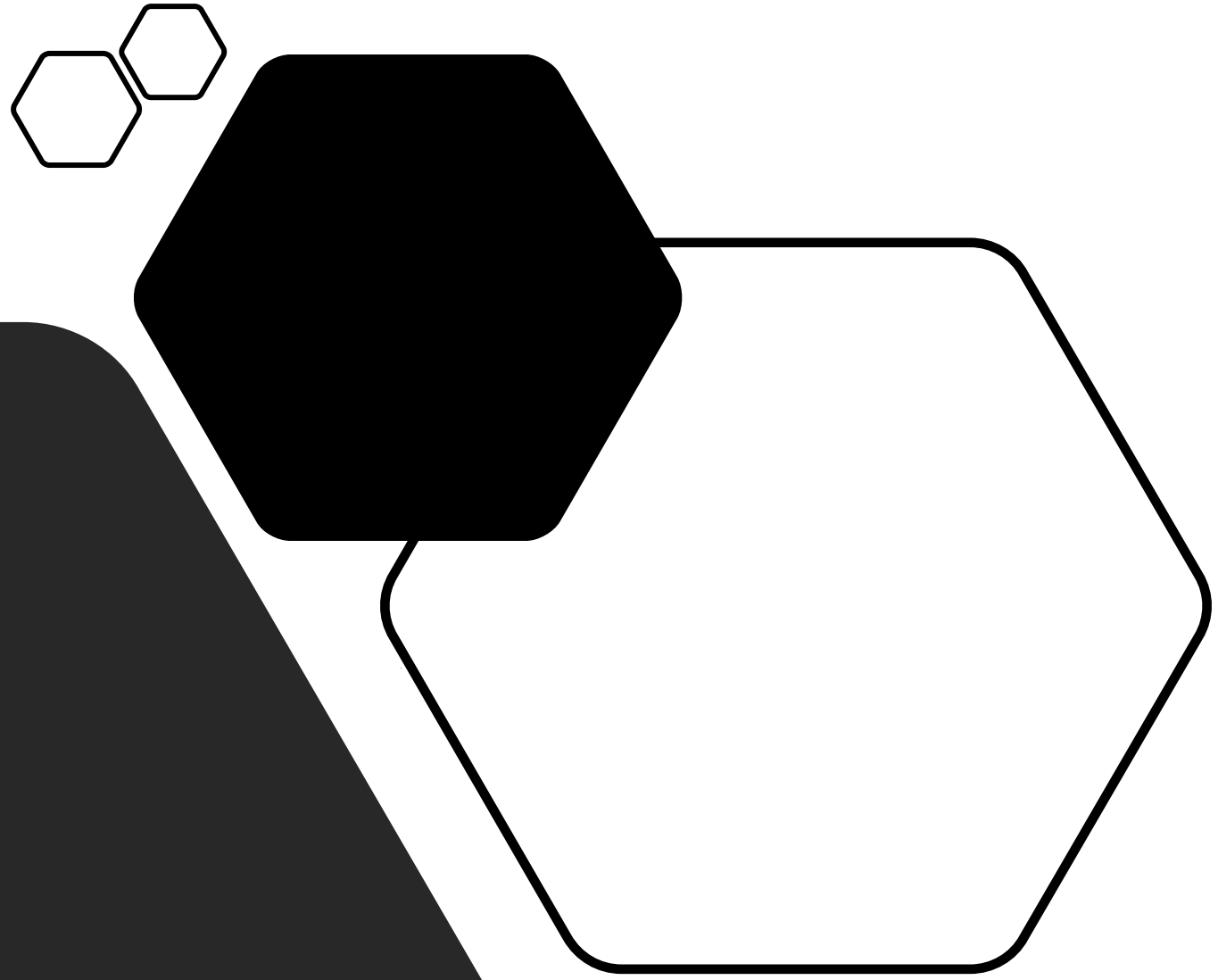
Clases en JS

- Desde junio de 2015, el estándar ECMAScript 6 de JS nos permite definir clases para representar las abstracciones de un objeto.
- Podemos elegir, por tanto, si queremos definir nuestras clases mediante constructores o mediante la creación de clases siguiendo el formato indicado en ECMAScript 6.

```
class Silla {
  constructor(legs, resp, acol, rec, ajus, height) {
    this.patas = legs;
    this.respaldo = resp;
    this.acolchada = acol;
    this.reclinable = rec;
    this.ajustable = ajus;
    this.altura = height;
    this.ajustar = function (alt) {
      if (this.ajustable)
        this.altura = alt;

      else
        console.log("Esta silla no es ajustable");
    };
  }
}
```

JSON



JSON

- El término lenguaje de marcas engloba a un conjunto amplio de lenguajes.
- JSON es algo diferente de los lenguajes de marcas vistos hasta ahora, principalmente por su sencillez.
  - No tiene etiquetas.
  - Está definido por una sintaxis muy simple, con un conjunto muy reducido de tipos de datos.
- JSON es el acrónimo de *JavaScript Object Notation*, y es un lenguaje de marcas muy sencillo, apropiado sobre todo para el intercambio de mensajes.

JSON

- La sintaxis de JSON es muy simple, y los documentos resultan más ligeros que en otros lenguajes como XML. Estas dos características son las razones de su éxito.
- Como punto negativo, no tiene en absoluto el nivel de estandarización de XML.
- Es muy fácil trabajar con JSON en JS, puesto que es su lenguaje de programación natural.
- Aparte de la validación sintáctica, JSON no dispone de forma nativa de mecanismos de validación análogos a los de XML.

JSON

- JS tiene una clase JSON, con métodos estáticos para convertir un objeto JSON en un String (por ejemplo, para enviarlo al servidor), así como para realizar la conversión de un String en un objeto JSON.

```
let stringJson = '{"titulo": "Los miserables","autor":"Victor Hugo"}';
let objetoJson = JSON.parse(stringJson);
console.log(objetoJson.titulo); // Los miserables
```

JSON

- Además de usarse como lenguaje de intercambio de información entre sistemas, JSON también se puede usar para almacenar información en bases de datos no relacionales.
- Los datos guardados en formato JSON generalmente ocupan menos espacio que en otros lenguajes de marcas.

Estructura y sintaxis JSON

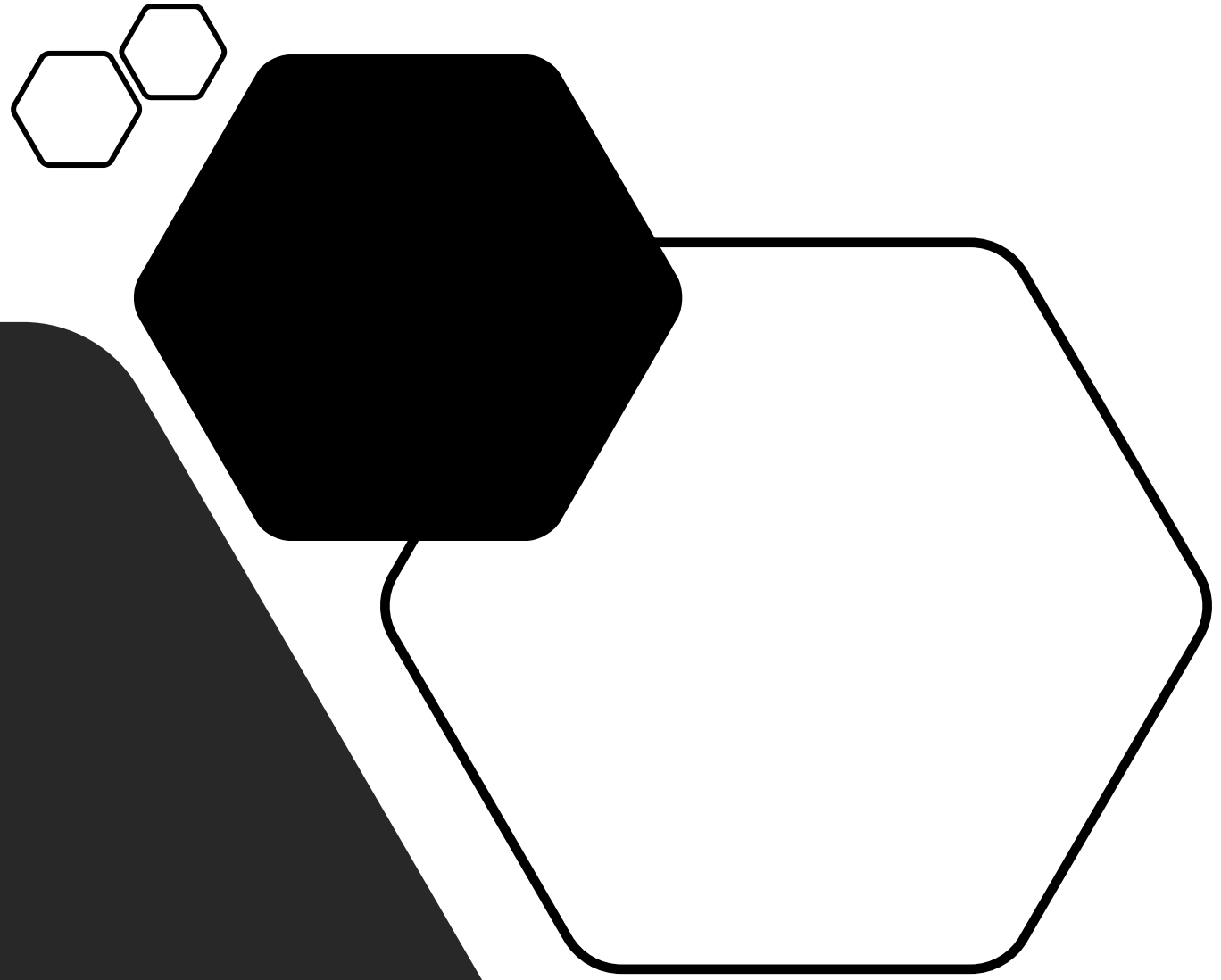
- La extensión de los ficheros es .json
- No existen los comentarios en JSON.
- La sintaxis de un objeto es la siguiente:
  - Están delimitados por llaves.
  - Contienen una lista de pares clave-valor, separados por comas.
  - Los nombres de las claves se separan de los valores por :
  - Los nombres de las claves se escriben entre comillas dobles.

Estructura y sintaxis JSON

- Los valores pueden ser:
  - Cadenas de caracteres (entre comillas dobles)
  - Números, tanto enteros como decimales.
  - Booleanos (true y false).
  - Objetos.
  - Arrays de objetos.
  - El valor null.
- Los arrays son colecciones de elementos, que pueden ser datos primitivos, objetos u otros arrays. La sintaxis es:
  - Están delimitados por corchetes [ ]
  - Los elementos están separados por comas.

```
{
  "grupo": "Daft Punk",
  "discos": [
    {
      "titulo": "Homework",
      "salida": "20-01-1997",
      "ventas": 2300000
    },
    {
      "titulo": "Discovery",
      "salida": "13-03-2001",
      "ventas": 3000000
    },
    {
      "titulo": "Human After All",
      "salida": "14-03-2005",
      "ventas": 1200000
    },
    {
      "titulo": "Random Access Memories",
      "salida": "17-05-2013",
      "ventas": 5000000
    }
  ]
}
```

SVG



SVG

- SVG son las siglas de *Scalable Vector Graphics*.
- Formato de representación gráfica bidimensional.
- Se basa en funciones matemáticas sobre un lenguaje de marcas para representación de imágenes.
- A mayores también permite la creación de animaciones.
- Basado en XML, pueden ser creadas y editadas mediante un editor de textos, aunque generalmente se usan programas de diseño específicos.

SVG

- Pueden ser modificados en tamaño, forma y color sin perder calidad.
- La extensión de archivo es .svg.
- Se puede editar en texto plano, mientras que para visualizarlo usamos un visor de imágenes o el propio navegador.

Uso de SVG: como imagen en HTML

- Al igual que otros tipos de imágenes, los SVG pueden ser añadidos a una web mediante la etiqueta <img>, funcionando así como una imagen más.

```
<img src="imagen.svg"/>
```

- También se pueden añadir mediante la etiqueta <object>, lo que permite referenciar el código del propio SVG como si estuviese creado directamente ahí, y así modificarlo o darle estilo mediante CSS.

```
<object type="image/svg+xml" data="imagen.svg">SVG no soportado</object>
```

Uso de SVG: como código en HTML

- Los SVG pueden ser embebidos y creados directamente dentro del HTML mediante la etiqueta *svg*:

```
<svg width="100" height="100">
    <circle cx="50" cy="50" r="40" fill="yellow"/>
</svg>
```

- En este ejemplo se crea un círculo amarillo de 40px de radio cuyo centro está en las coordenadas (50,50) en un contenedor de 100x100px.
- Al crearse en HTML, podemos referenciarlo como cualquier otro elemento y darle formato con CSS o manipularlo con JS.

Uso de SVG: con CSS

- Al crearse en HTML o importarlo como objeto, sus elementos pueden ser referenciados mediante CSS y aplicarle estilos.
- Las propiedades CSS para los elementos SVG difieren de las de HTML, ya que disponen de propiedades y valores especiales. Por ejemplo, para ponerle un color a un elemento SVG se utiliza la propiedad *fill*, no *background-color*. Para bordes es *stroke* en vez de *border-color*.
- Estas propiedades son las mismas, en la mayoría de los casos, que los atributos de las propias etiquetas SVG.

Etiquetas SVG

- SVG dispone de elementos con los que representar cualquier imagen.
- Pueden ser algo tan sencillo como una línea o un rectángulo o ser más complejos, como una elipsis o un camino multiforma.
- Todos los elementos se basan en coordenadas en el plano de dibujo, siendo la esquina superior izquierda el punto de referencia (0,0), aumentando positivamente hacia abajo y la derecha.
- El **formato de los elementos** suele indicarse en la propia etiqueta mediante el uso de atributos.

Etiquetas SVG: Línea

- El elemento más sencillo es la línea, que se define con <line>.
- Los atributos básicos son:
  - **x1, y1:** coordenadas x e y de origen.
  - **x2, y2:** las coordenadas x e y de destino.

```
<svg height="210" width="500">
    <line x1="0" y1="0" x2="200" y2="200" stroke="rgb(255,0,0)" stroke-width="3"/>
    Tu navegador no soporta SVG.
</svg>
```

- En este ejemplo se crea una línea roja de 3px de grosor desde el punto (0,0) hasta el (200,200)

Etiquetas SVG: Multilínea

- La etiqueta <polyline> define una línea con múltiples puntos.
- Los puntos se definen en el atributo *points*, que admite indefinidos pares de coordenadas separadas por espacios. Las coordenadas se estipulan en formato “x,y”.

```
<svg height="180" width="500">
   <polyline
      points="0,40 40,40 40,80 80,80 80,120 120,120 120,160"
      style="fill:white; stroke:red; stroke-width:4" />
   Tu navegador no soporta SVG.
</svg>
```

Etiquetas SVG: Rectángulo

- La etiqueta <rect> permite definir rectángulos con un tamaño predeterminado. Sus atributos son:
  - **x** e **y**: coordenadas de origen.
  - **width** y **height**: el ancho y el alto del rectángulo

```
<svg height="180" width="400">
   <rect x="50" y="20" width="150" height="150"
      style="fill:blue; stroke:pink; stroke-width:5; fill-opacity: 0.1;
stroke-opacity:0.9" />
   Tu navegador no soporta SVG.
</svg>
```

Etiquetas SVG: Polígono

- La etiqueta <polygon> permite definir polígonos con n lados estableciendo las coordenadas de sus vértices.
- Al igual que con *polyline*, se usa el atributo **points** para cada vértice. El polígono cierra automáticamente la figura, creando la línea entre el primer y último vértice.

```
<svg height="210" width="500">
   <polygon points="200,10 250,190 160,210"
      style="fill:lime; stroke:purple; stroke-width:1;" />
   Tu navegador no soporta SVG.
</svg>
```

Etiquetas SVG: Círculo

- La etiqueta <circle> permite crear círculos determinando las coordenadas de su centro y su radio.
- Sus atributos son:
  - **cx** y **cy**: coordenadas x e y del centro.
  - **r**: el radio del círculo.

```
<svg height="100" width="100">
   <circle cx="50" cy="50" r="40" fill="tomato" />
   Tu navegador no soporta SVG.
</svg>
```

Etiquetas SVG: Elipse

- La etiqueta <ellipse> permite representar elipses de una forma muy similar al círculo.
- Sus atributos son:
  - **cx** y **cy**: coordenadas x e y del centro.
  - **rx y ry**: el radio máximo en horizontal y en vertical.

```
<svg height="150" width="500">
   <ellipse cx="150" cy="80" rx="100" ry="50"
      style="fill:wheat;stroke:purple;stroke-width:2" />
   Tu navegador no soporta SVG.
</svg>
```

Etiquetas SVG: Texto

- En SVG también se puede dibujar texto mediante el elemento texto. Este elemento dispone de los atributos x e y para indicar la posición de inicio de texto y el contenido de la propia etiqueta es el texto a mostrar.
- Hay que tener en cuenta que las coordenadas son la línea base de escritura.

```
<svg height="60" width="200">
   <text x="5" y="15" transform="rotate(30 20,40)">Texto en SVG</text>
   Tu navegador no soporta SVG.
</svg>
```

Programas que trabajan con SVG

- Dada la complejidad de su creación mediante código, existen múltiples programas en el mercado diseñados expresamente para su creación y manipulación:
- **Adobe Illustrator**: de los más usados a nivel profesional. Diseñado en exclusiva para trabajar con SVG. Es de pago bajo suscripción mensual.
- **Corel Draw**: otro de los más usados a nivel profesional. Permite trabajar con varios formatos de imagen, incluido SVG. Es de pago bajo suscripción o pago único sin actualizaciones.
- **Inkscape**: ampliamente usado por la comunidad dado que tiene opción para MAC y GNU/Linux. Es gratuito y bajo licencia de software libre.