

Parte 2

El navegador: Documentos, Eventos e Interfaces

JS

Ilya Kantor

Hecho el 13 de junio de 2021

La última versión de este tutorial está en <https://es.javascript.info>.

Trabajamos constantemente para mejorar el tutorial. Si encuentra algún error, por favor escríbanos a [nuestro github](#).

- Documento
 - Entorno del navegador, especificaciones
 - Árbol del Modelo de Objetos del Documento (DOM)
 - Recorriendo el DOM
 - Buscar: getElement*, querySelector*
 - Propiedades del nodo: tipo, etiqueta y contenido
 - Atributos y propiedades
 - Modifying the document
 - Estilos y clases
 - Elementos de tamaño y desplazamiento
 - Tamaño de ventana y desplazamiento
 - Coordenadas
- Introducción a los eventos
 - Introducción a los eventos en el navegador
 - Propagación y captura
 - Delegación de eventos
 - Acciones predeterminadas del navegador
 - Envío de eventos personalizados
- Eventos en la UI
 - Eventos del Mouse
 - Moviendo el mouse: mouseover/out, mouseenter/leave
 - Arrastrar y Soltar con eventos del ratón
 - Eventos de puntero
 - Keyboard: keydown and keyup
 - Desplazamiento
- Formularios y controles
 - Propiedades y Métodos de Formularios
 - Focusing: focus/blur
 - Eventos: change, input, cut, copy, paste
 - Formularios: evento y método submit
- El documento y carga de recursos
 - Page: DOMContentLoaded, load, beforeunload, unload
 - Scripts: async, defer
 - Carga de recursos: onload y onerror
- Temas diversos
 - Mutation observer
 - Selection y Range
 - Loop de eventos: microtarefas y macrotareas

Aprenderemos a manejar la página del navegador: agregar elementos, manipular su tamaño y posición, crear interfaces dinámicamente e interactuar con el visitante.

Documento

Aquí aprenderemos a manipular una página web usando JavaScript.

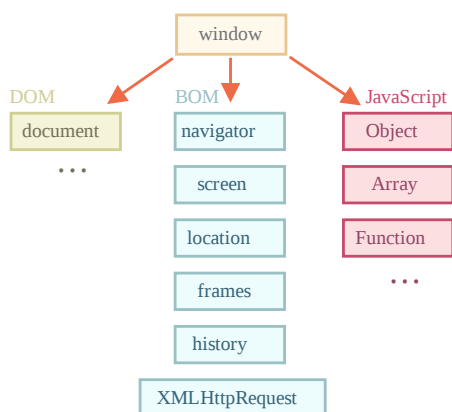
Entorno del navegador, especificaciones

El lenguaje JavaScript fue creado inicialmente para los navegadores web. Desde entonces, ha evolucionado y se ha convertido en un lenguaje con muchos usos y plataformas.

Una plataforma puede ser un navegador, un servidor web u otro *host*, incluso una máquina de café “inteligente”, si puede ejecutar JavaScript. Cada uno de ellos proporciona una funcionalidad específica de la plataforma. La especificación de JavaScript llama esto como *entorno de host* (host environment).

Un entorno host proporciona objetos específicos de la plataforma y funciones adicionales al núcleo del lenguaje. Los navegadores web proporcionan un medio para controlar las páginas web. Node.JS proporciona características del lado del servidor, y así sucesivamente.

Aquí tienes una vista general de lo que tenemos cuando JavaScript se ejecuta en un navegador web:



Hay un objeto “raíz” llamado `window`. Tiene dos roles:

1. Primero, es un objeto global para el código JavaScript, como se describe en el capítulo [Objeto global](#).
2. En segundo lugar, representa la “ventana del navegador” y proporciona métodos para controlarla.

Por ejemplo, aquí lo usamos como un objeto global:

```
function sayHi() {
  alert("Hola");
}

// Las funciones globales son métodos del objeto global:
window.sayHi();
```

Y aquí lo usamos como una ventana del navegador, para ver la altura de la ventana:

```
alert(window.innerHeight); // altura interior de la ventana
```

Hay más métodos y propiedades específicos de `window`, los cubriremos más adelante.

Modelo de Objetos del Documento (DOM)

El objeto `document` da acceso al contenido de la página. Con él podemos cambiar o crear cualquier cosa en la página.

Por ejemplo:

```
// cambiar el color de fondo a rojo
document.body.style.background = "red";

// deshacer el cambio después de 1 segundo
setTimeout(() => document.body.style.background = "", 1000);
```

Aquí usamos `document.body.style`, pero hay muchos, muchos más. Las propiedades y métodos se describen en la especificación: [DOM Living Standard](#).

i DOM no es solo para navegadores

La especificación DOM explica la estructura de un documento y proporciona objetos para manipularlo. Hay instrumentos que no son del navegador que también usan DOM.

Por ejemplo, los scripts del lado del servidor que descargan páginas HTML y las procesan, también pueden usar DOM. Sin embargo, pueden admitir solo una parte de la especificación.

i CSSOM para los estilos

También hay una especificación separada, [CSS Object Model \(CSSOM\)](#) para las reglas y hojas de estilo CSS, que explica cómo se representan como objetos y cómo leerlos y escribirlos.

CSSOM se usa junto con DOM cuando modificamos las reglas de estilo para el documento. Sin embargo, en la práctica, rara vez se requiere CSSOM, porque rara vez necesitamos modificar las reglas CSS de JavaScript (generalmente solo agregamos / eliminamos clases CSS, no modificamos sus reglas CSS), pero eso también es posible.

BOM (parte de la especificación HTML)

El Modelo de Objetos del Navegador (BOM) son objetos adicionales proporcionados por el navegador (entorno host) para trabajar con todo excepto el documento.

Por ejemplo:

- El objeto [navigator](#), proporciona información sobre el navegador y el sistema operativo. Hay muchas propiedades, pero las dos más conocidas son: `navigator.userAgent` – sobre el navegador actual, y `navigator.platform` – sobre la plataforma (puede ayudar a diferenciar entre Windows/Linux/Mac, etc.).
- El objeto [location](#), nos permite leer la URL actual y puede redirigir el navegador a uno nuevo.

Aquí vemos cómo podemos usar el objeto `location`:

```
alert(location.href); // muestra la URL actual
if (confirm("Go to wikipedia?")) {
  location.href = "https://wikipedia.org"; // redirigir el navegador a otra URL
}
```

Las funciones `alert/confirm/prompt` también forman parte de BOM: no están directamente relacionadas con el documento, sino que representan métodos de comunicación puros con el usuario.

i Especificación de HTML

BOM es la parte general de la especificación de [HTML specification](#).

Sí, oíste bien. La especificación HTML en <https://html.spec.whatwg.org> no solo trata sobre el “lenguaje HTML” (etiquetas, atributos), sino que también cubre un montón de objetos, métodos y extensiones DOM específicas del navegador. Eso es “HTML en términos generales”.

Resumen

Hablando de estándares, tenemos:

La especificación del DOM

Describe la estructura del documento, las manipulaciones y los eventos, consulte <https://dom.spec.whatwg.org> .

La especificación del CSSOM

Describe las hojas de estilo y las reglas de estilo, las manipulaciones con ellas y su vinculación a los documentos, consulte <https://www.w3.org/TR/cssom-1/> .

La especificación del HTML

Describe el lenguaje HTML (por ejemplo, etiquetas) y también el BOM (modelo de objeto del navegador) – varias funciones del navegador: `setTimeout` , `alert` , `location` , etc., consulte <https://html.spec.whatwg.org> . Toma la especificación DOM y la extiende con muchas propiedades y métodos adicionales.

Ahora nos concentraremos en aprender el DOM, porque juega un papel central en la interfaz de usuario.

Ten en cuenta los enlaces anteriores, ya que hay tantas cosas que aprender que es imposible cubrir y recordar todo.

Cuando desees leer sobre una propiedad o un método, el manual de Mozilla en <https://developer.mozilla.org/es/search> es un buen recurso, pero leer las especificaciones correspondientes puede ser mejor: es más complejo y hay más para leer, pero hará que su conocimiento de los fundamentos sea sólido y completo.

Árbol del Modelo de Objetos del Documento (DOM)

La estructura de un documento HTML son las etiquetas.

Según el Modelo de Objetos del Documento (DOM), cada etiqueta HTML es un objeto. Las etiquetas anidadas son llamadas “hijas” de la etiqueta que las contiene. El texto dentro de una etiqueta también es un objeto.

Todos estos objetos son accesibles empleando JavaScript, y podemos usarlos para modificar la página.

Por ejemplo, `document.body` es el objeto que representa la etiqueta `<body>` .

Ejecutar el siguiente código hará que el `<body>` sea de color rojo durante 3 segundos:

```
document.body.style.background = 'red'; // establece un color de fondo rojo

setTimeout(() => document.body.style.background = '', 3000); // volver atrás
```

En el caso anterior usamos `style.background` para cambiar el color de fondo del `document.body` , sin embargo existen muchas otras propiedades, tales como:

- `innerHTML` – contenido HTML del nodo.
- `offsetWidth` – ancho del nodo (en píxeles).
- ..., etc.

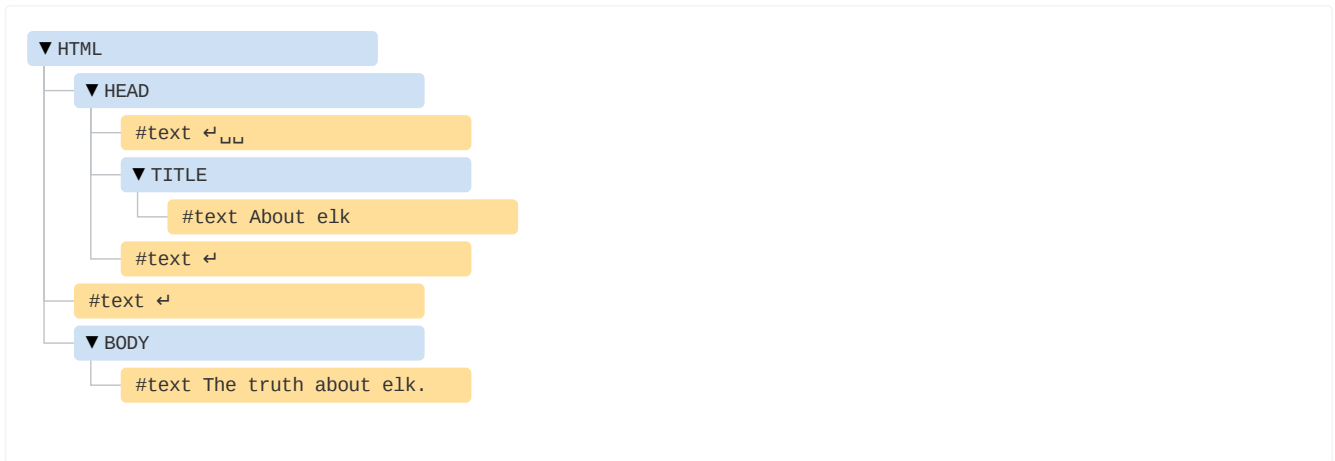
Más adelante, aprenderemos otras formas de manipular el DOM, pero primero necesitamos conocer su estructura.

Un ejemplo del DOM

Comencemos con el siguiente documento simple:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>About elk</title>
</head>
<body>
  The truth about elk.
</body>
</html>
```

El DOM representa HTML como una estructura de árbol de etiquetas. A continuación podemos ver como se muestra:



Cada nodo del árbol es un objeto.

Las etiquetas son *nodos de elementos* (o solo elementos) y forman la estructura del árbol: `<html>` está ubicado en la raíz del documento, por lo tanto, `<head>` y `<body>` son sus hijos, etc.

El texto dentro de los elementos forma *nodos de texto*, etiquetados como `#text`. Un nodo de texto contiene solo una cadena. Esta puede no tener hijos y siempre es una hoja del árbol.

Por ejemplo, la etiqueta `<title>` tiene el texto `"About elk"`.

Hay que tener en cuenta los caracteres especiales en nodos de texto:

- una línea nueva: `↵` (en JavaScript se emplea `\n` para obtener este resultado)
- un espacio:

Los espacios y líneas nuevas son caracteres totalmente válidos, al igual que letras y dígitos. Ellos forman nodos de texto y se convierten en parte del DOM. Así, por ejemplo, en el caso de arriba la etiqueta `<head>` contiene algunos espacios antes de la etiqueta `<title>`, entonces ese texto se convierte en el nodo `#text`, que contiene una nueva línea y solo algunos espacios.

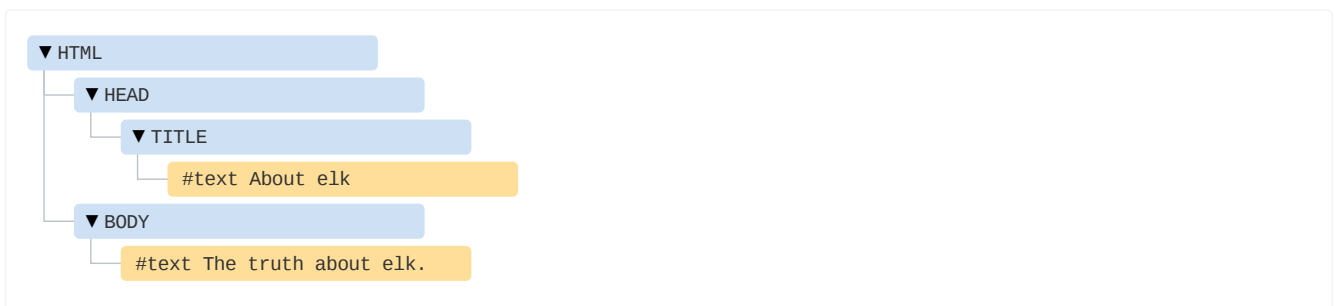
Hay solo dos excepciones de nivel superior:

1. Los espacios y líneas nuevas antes de la etiqueta `<head>` son ignorados por razones históricas.
2. Si colocamos algo después de la etiqueta `</body>`, automáticamente se sitúa dentro de `body`, al final, ya que, la especificación HTML necesita que todo el contenido esté dentro de la etiqueta `<body>`, no puede haber espacios después de esta.

En otros casos todo es sencillo – si hay espacios (como cualquier carácter) en el documento, se convierten en nodos de texto en el DOM, y si los eliminamos, entonces no habrá ninguno.

En el siguiente ejemplo, no hay nodos de texto con espacios en blanco:

```
<!DOCTYPE HTML>
<html><head><title>About elk</title></head><body>The truth about elk.</body></html>
```



i Los espacios al inicio/final de la cadena y los nodos de texto que solo contienen espacios en blanco, por lo general, están ocultos en las herramientas

Las herramientas del navegador (las veremos más adelante) que trabajan con DOM usualmente no muestran espacios al inicio/final del texto y nodos de texto vacíos (saltos de línea) entre etiquetas.

De esta manera, las herramientas para desarrolladores ahorran espacio en la pantalla.

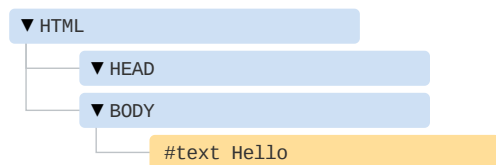
En otras representaciones del DOM, las omitiremos cuando sean irrelevantes. Tales espacios generalmente no afectan la forma en la cual el documento es mostrado.

Autocorrección

Si el navegador encuentra HTML mal escrito, lo corrige automáticamente al construir el DOM.

Por ejemplo, la etiqueta superior siempre será `<html>`. Incluso si no existe en el documento, ésta existirá en el DOM, puesto que, el navegador la creará. Sucede lo mismo con la etiqueta `<body>`.

Como ejemplo de esto, si el archivo HTML es la palabra "Hello", el navegador lo envolverá con las etiquetas `<html>` y `<body>`, y añadirá la etiqueta `<head>` la cual es requerida, basado en esto, el DOM resultante será:

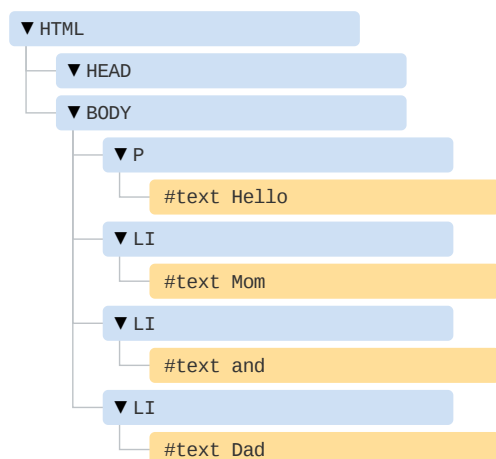


Al generar el DOM, los navegadores procesan automáticamente los errores en el documento, cierran etiquetas, etc.

Un documento sin etiquetas de cierre:

```
<p>Hello
<li>Mom
<li>and
<li>Dad
```

...se convertirá en un DOM normal a medida que el navegador lee las etiquetas y compone las partes faltantes:



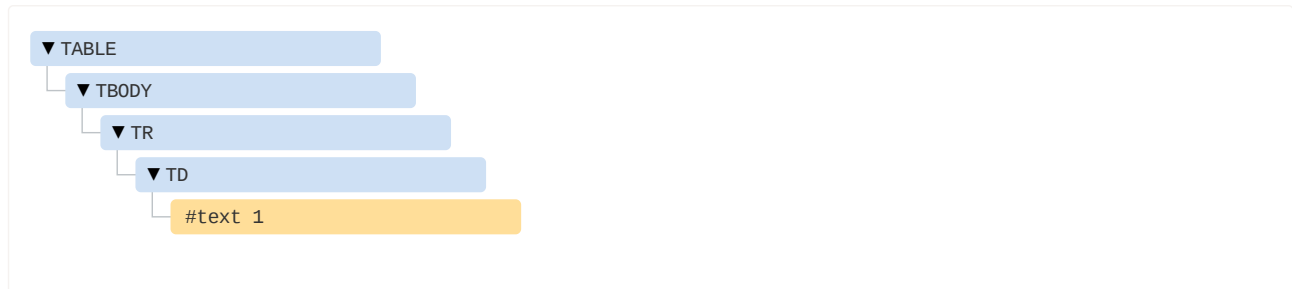
⚠ Las tablas siempre tienen la etiqueta `<tbody>`

Un caso especial interesante son las tablas. De acuerdo a la especificación DOM deben tener la etiqueta `<tbody>`, sin embargo el texto HTML puede omitirla: el navegador crea automáticamente la etiqueta `<tbody>` en el DOM.

Para el HTML:

```
<table id="table"><tr><td>1</td></tr></table>
```

La estructura del DOM será:



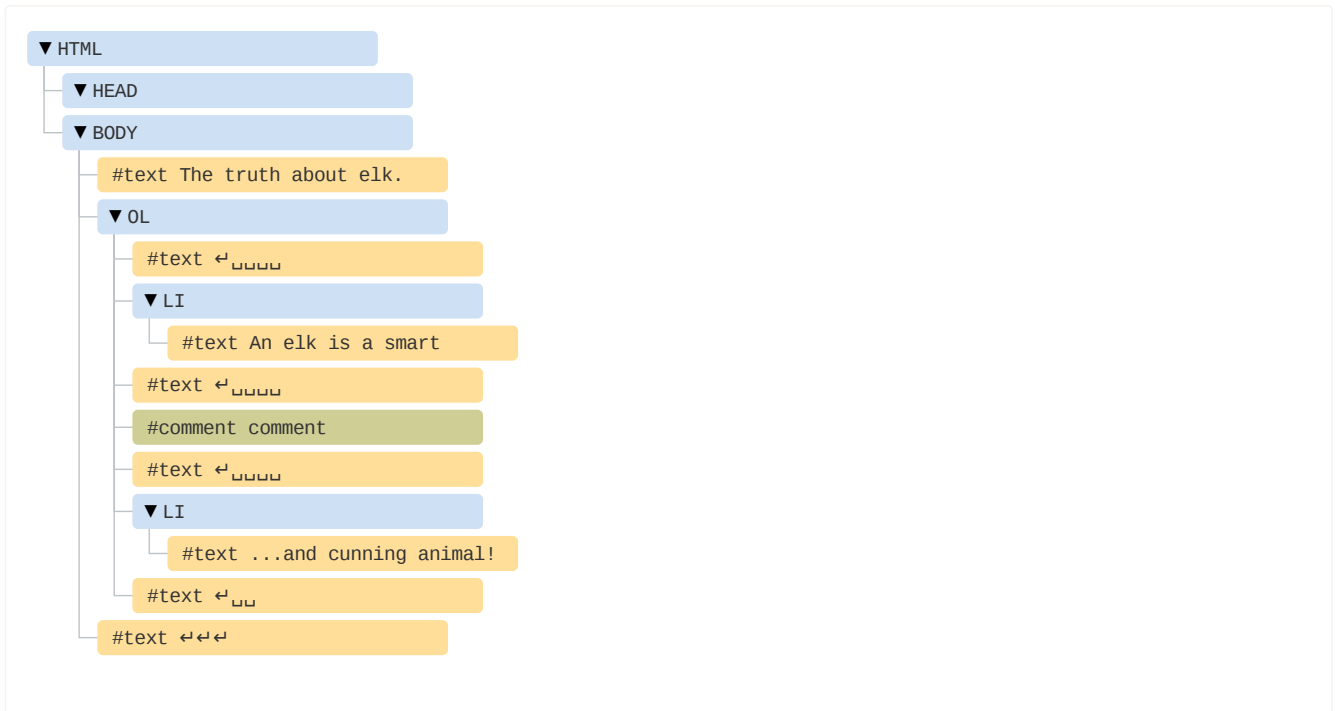
¿Lo ves? La etiqueta `<tbody>` apareció de la nada. Debemos tener esto en cuenta al trabajar con tablas para evitar sorpresas.

Otros tipos de nodos

Existen otros tipos de nodos además de elementos y nodos de texto.

Por ejemplo, los comentarios:

```
<!DOCTYPE HTML>
<html>
<body>
  The truth about elk.
  <ol>
    <li>An elk is a smart</li>
    <!-- comentario -->
    <li>...y el astuto animal!</li>
  </ol>
</body>
</html>
```

Aquí podemos ver un nuevo tipo de nodo de árbol – *nodo de comentario*, etiquetado como `#comment`, entre dos nodos de texto.

Podemos pensar – ¿Por qué se agrega un comentario al DOM? Esto no afecta la representación de ninguna manera. Pero hay una regla – si algo está en el código HTML, entonces también debe estar en el árbol DOM.

Todo en HTML, incluso los comentarios, se convierte en parte del DOM.

Hasta la declaración `<!DOCTYPE . . .>` al principio del HTML es un nodo del DOM. Su ubicación en el DOM es justo antes de la etiqueta `<html>`. No vamos a tocar ese nodo, por esa razón ni siquiera lo dibujamos en diagramas, pero esta ahí.

El objeto `document` que representa todo el documento es también, formalmente, un nodo DOM.

Hay [12 tipos de nodos](#). En la práctica generalmente trabajamos con 4 de ellos:

1. `document` – el “punto de entrada” en el DOM.
2. nodos de elementos – Etiquetas-HTML, los bloques de construcción del árbol.
3. nodos de texto – contienen texto.
4. comentarios – a veces podemos colocar información allí, no se mostrará, pero JS puede leerla desde el DOM.

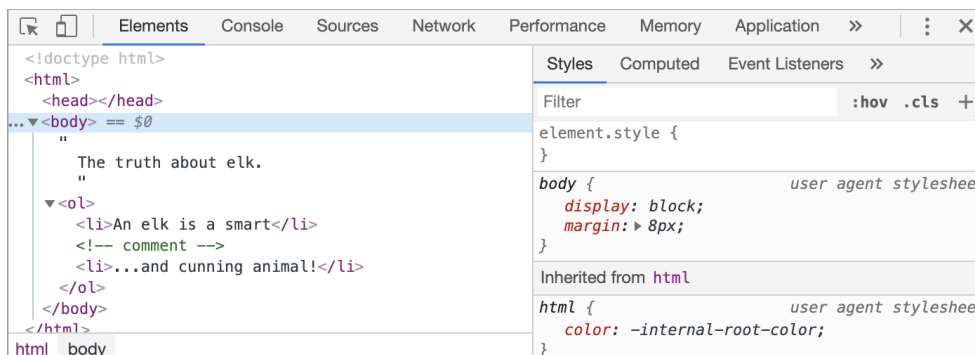
Véalo usted mismo

Para ver la estructura del DOM en tiempo real, intente [Live DOM Viewer](#). Simplemente escriba el documento, y se mostrará como un DOM al instante.

Otra forma de explorar el DOM es usando la herramienta para desarrolladores del navegador. En realidad, eso es lo que usamos cuando estamos desarrollando.


Para hacerlo, abra la página web [elk.html](#), active las herramientas para desarrolladores del navegador y cambie la pestaña a elementos.

Debe verse así:

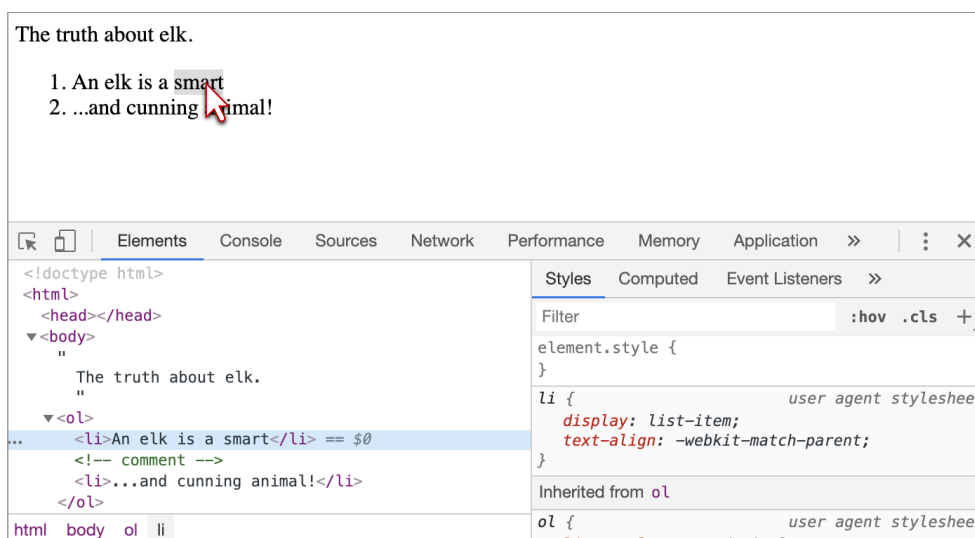


Puedes ver el DOM, hacer clic sobre los elementos, ver sus detalles, etc.

Tenga en cuenta que la estructura DOM en la herramienta para desarrolladores está simplificada. Los nodos de texto se muestran como texto. Y absolutamente no hay nodos de texto con espacios en blanco. Esto está bien, porque la mayoría de las veces nos interesan los nodos de elementos.

Hacer clic en el botón  ubicado en la esquina superior izquierda nos permite elegir un nodo desde la página web utilizando un “mouse” (u otros dispositivos de puntero) e “inspeccionar” (desplazarse hasta él en la pestaña elementos). Esto funciona muy bien cuando tenemos una página HTML enorme (y el DOM correspondiente es enorme) y nos gustaría ver la posición de un elemento en particular.

Otra forma de realizarlo sería hacer clic derecho en la página web y en el menú contextual elegir la opción “Inspeccionar Elemento”.



En la parte derecha de las herramientas encontramos las siguientes sub-pestañas:

- **Styles** – podemos ver CSS aplicado al elemento actual regla por regla, incluidas las reglas integradas (gris). Casi todo puede ser editado en el lugar, incluyendo las dimensiones/márgenes/relleno de la siguiente caja.
- **Computed** – nos permite ver cada propiedad CSS aplicada al elemento: para cada propiedad podemos ver la regla que la provee (incluida la herencia CSS y demás).
- **Event Listeners** – nos ayuda a ver los listener de eventos adjuntos a elementos del DOM (los cubriremos en la siguiente parte del tutorial).
- ...,etc.

La manera de estudiarlos es haciendo clic en ellos. Casi todos los valores son editables en el lugar.

Interacción con la consola

A medida que trabajamos el DOM, también podemos querer aplicarle JavaScript. Al igual que: obtener un nodo y ejecutar algún código para modificarlo, para ver el resultado. Aquí hay algunos consejos para desplazarse entre la pestaña elementos y la consola.

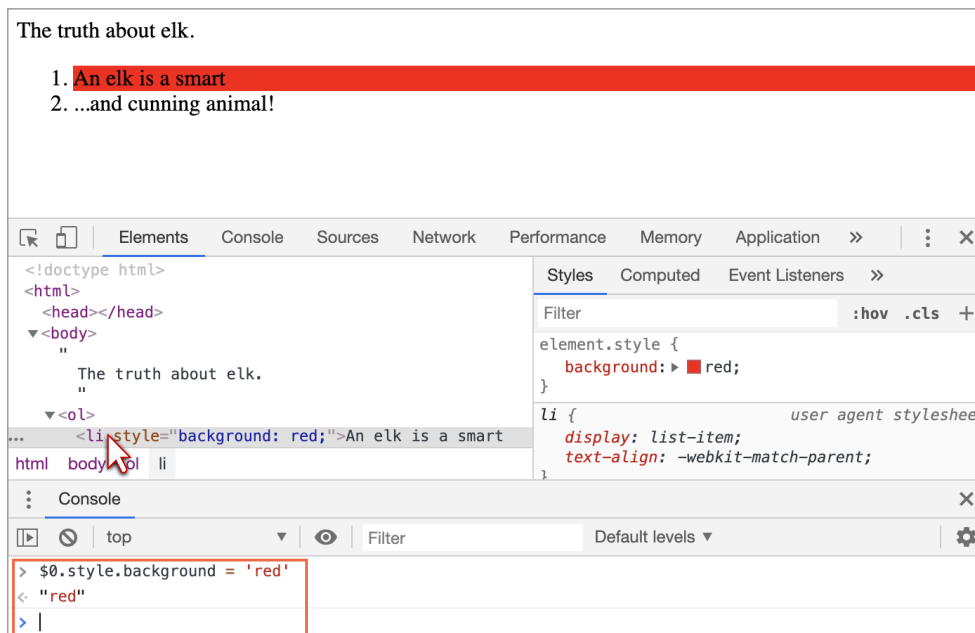
Para empezar:

1. Seleccione el primer elemento `` en la pestaña elementos.

2. Presiona la tecla `Esc` – esto abrirá la consola justo debajo de la pestaña de elementos.

Ahora el último elemento seleccionado esta disponible como `$0`, el seleccionado previamente es `$1`, etc.

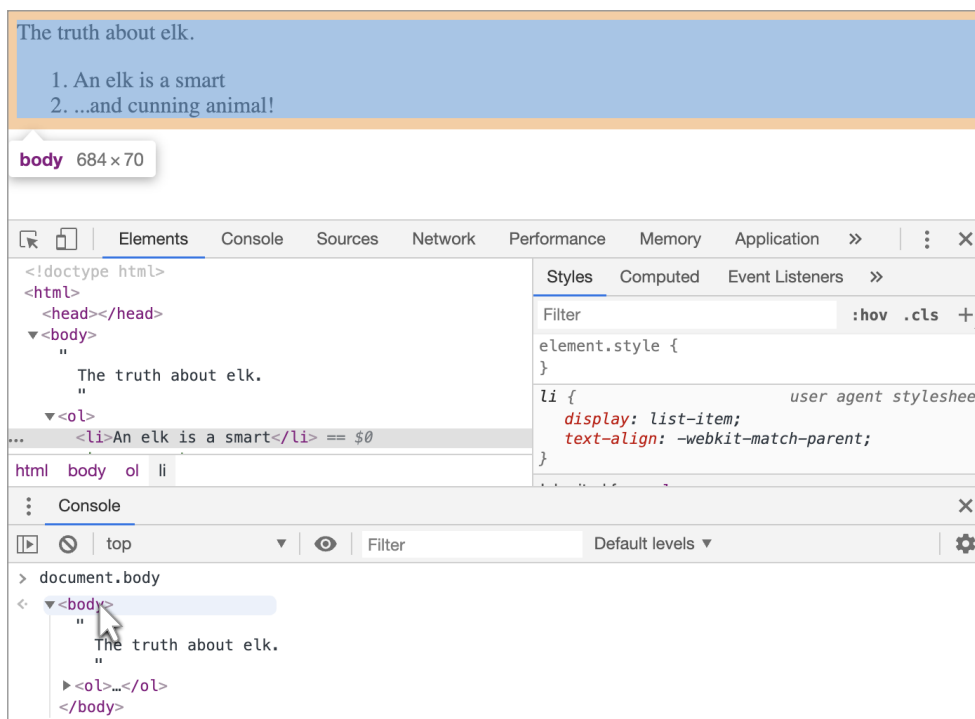
Podemos ejecutar comandos en ellos. Por ejemplo, `$0.style.background = 'red'` hace que el elemento de la lista seleccionado sea rojo, algo así:



Así es como en la consola, se obtiene un nodo de los elementos.

También hay un camino de regreso. Si hay una variable que hace referencia a un nodo del DOM, usamos el comando `inspect (node)` en la consola para verlo en el panel de elementos.

O simplemente podemos generar el nodo del DOM en la consola y explorar en el lugar, así como `document.body` a continuación:



Desde luego, eso es para propósitos de depuración del curso. A partir del siguiente capítulo accederemos y modificaremos el DOM usando JavaScript.

Las herramientas para desarrolladores del navegador son de mucha ayuda en el desarrollo: podemos explorar el DOM, probar cosas y ver que sale mal.

Resumen

Un documento HTML/XML está representado dentro del navegador como un árbol de nodos (DOM).

- Las etiquetas se convierten en nodos de elemento y forman la estructura.
- El texto se convierte en nodos de texto.
- ...etc, todos los elementos de HTML tienen su lugar en el DOM, incluso los comentarios.

Podemos utilizar las herramientas para desarrolladores para inspeccionar el DOM y modificarlo manualmente.

Aquí cubrimos los conceptos básicos, las acciones más importantes y más utilizadas, para comenzar. Hay una extensa documentación acerca de las herramientas para desarrolladores de Chrome en <https://developers.google.com/web/tools/chrome-devtools> . La mejor forma de aprender a usar las herramientas es hacer clic en ellas, leer los menús: la mayoría de las opciones son obvias. Más adelante, cuando tenga conocimiento general sobre ellas, lea los documentos y elija el resto.

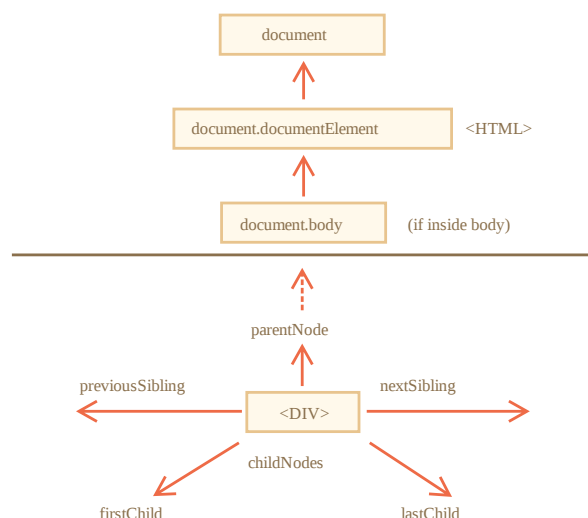
Los nodos del DOM tienen propiedades y métodos que nos permiten desplazarnos entre ellos, modificarlos, movernos por la página, y más. Empezaremos a realizar todo esto en los siguientes capítulos.

Recorriendo el DOM

El DOM nos permite hacer cualquier cosa con sus elementos y contenidos, pero lo primero que tenemos que hacer es llegar al objeto correspondiente del DOM.

Todas las operaciones en el DOM comienzan con el objeto `document` . Este es el principal “punto de entrada” al DOM. Desde ahí podremos acceder a cualquier nodo.

Esta imagen representa los enlaces que nos permiten viajar a través de los nodos del DOM:



Vamos a analizarlos con más detalle.

En la parte superior: `documentElement` y `body`

Los tres nodos superiores están disponibles como propiedades de `document` :

`<html>` = `document.documentElement`

El nodo superior del documento es `document.documentElement` . Este es el nodo del DOM para la etiqueta `<html>` .

`<body>` = `document.body`

Otro nodo muy utilizado es el elemento `<body>` – `document.body` .

`<head>` = `document.head`

La etiqueta `<head>` está disponible como `document.head` .

⚠ Hay una trampa: `document.body` puede ser `null`

Un script no puede acceder a un elemento que no existe en el momento de su ejecución.

Por ejemplo, si un script está dentro de `<head>`, entonces `document.body` no está disponible, porque el navegador no lo ha leído aún.

Entonces, en el siguiente ejemplo `alert` muestra `null`:

```
<html>

<head>
  <script>
    alert( "From HEAD: " + document.body ); // null, no hay <body> aún
  </script>
</head>

<body>

  <script>
    alert( "From BODY: " + document.body ); // HTMLBodyElement, ahora existe
  </script>

</body>
</html>
```

ℹ En el mundo del DOM `null` significa “no existe”

En el DOM, el valor `null` significa que “no existe” o “no hay tal nodo”.

Hijos: `childNodes`, `firstChild`, `lastChild`

Existen dos términos que vamos a utilizar de ahora en adelante:

- **Nodos hijos (`childNodes`)** – elementos que son hijos directos, es decir sus descendientes inmediatos. Por ejemplo, `<head>` y `<body>` son hijos del elemento `<html>`.
- **Descendientes** – todos los elementos anidados de un elemento dado, incluyendo los hijos, sus hijos y así sucesivamente.

Por ejemplo, aquí `<body>` tiene de hijos `<div>` y `` (y unos pocos nodos de texto en blanco):

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>
      <b>Information</b>
    </li>
  </ul>
</body>
</html>
```

...Y los descendientes de `<body>` no son solo los hijos `<div>`, `` sino también elementos anidados más profundamente, como `` (un hijo de ``) o `` (un hijo de ``) – el subárbol entero.

La colección `childNodes` enumera todos los nodos hijos, incluidos los nodos de texto.

El ejemplo inferior muestra todos los hijos de `document.body`:

```
<html>
<body>
  <div>Begin</div>
```

```

<ul>
  <li>Information</li>
</ul>

<div>End</div>

<script>
  for (let i = 0; i < document.body.childNodes.length; i++) {
    alert( document.body.childNodes[i] ); // Texto, DIV, Texto, UL, ..., SCRIPT
  }
</script>
...más cosas...
</body>
</html>

```

Por favor observa un interesante detalle aquí. Si ejecutamos el ejemplo anterior, el último elemento que se muestra es `<script>`. De hecho, el documento tiene más cosas debajo, pero en el momento de ejecución del script el navegador todavía no lo ha leído, por lo que el script no lo ve.

Las propiedades `firstChild` y `lastChild` dan acceso rápido al primer y al último hijo.

Son solo atajos. Si existieran nodos hijos, la respuesta siguiente sería siempre verdadera:

```

elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

```

También hay una función especial `elem.hasChildNodes()` para comprobar si hay algunos nodos hijos.

Colecciones del DOM

Como podemos ver, `childNodes` parece un array. Pero realmente no es un array, sino más bien una *colección* – un objeto especial iterable, simil-array.

Hay dos importantes consecuencias de esto:

1. Podemos usar `for...of` para iterar sobre él:

```

for (let node of document.body.childNodes) {
  alert(node); // enseña todos los nodos de la colección
}

```

Eso es porque es iterable (proporciona la propiedad `Symbol.iterator`, como se requiere).

2. Los métodos de Array no funcionan, porque no es un array:

```

alert(document.body.childNodes.filter); // undefined (¡No hay método filter!)

```

La primera consecuencia es agradable. La segunda es tolerable, porque podemos usar `Array.from` para crear un array “real” desde la colección si es que queremos usar métodos del array:

```

alert( Array.from(document.body.childNodes).filter ); // función

```

⚠ Las colecciones DOM son solo de lectura

Las colecciones DOM, incluso más-- *todas* las propiedades de navegación enumeradas en este capítulo son sólo de lectura.

No podemos reemplazar a un hijo por otro elemento asignándolo así `childNodes[i] = ...`.

Cambiar el DOM necesita otros métodos. Los veremos en el siguiente capítulo.

⚠ Las colecciones del DOM están vivas

Casi todas las colecciones del DOM, salvo algunas excepciones, están *vivas*. En otras palabras, reflejan el estado actual del DOM.

Si mantenemos una referencia a `elem.childNodes`, y añadimos o quitamos nodos del DOM, entonces estos nodos aparecen en la colección automáticamente.

⚠ No uses `for...in` para recorrer colecciones

Las colecciones son iterables usando `for...of`. Algunas veces las personas tratan de utilizar `for...in` para eso.

Por favor, no lo hagas. El bucle `for...in` itera sobre todas las propiedades enumerables. Y las colecciones tienen unas propiedades “extra” raramente usadas que normalmente no queremos obtener:

```
<body>
<script>
  // enseña 0, 1, longitud, item, valores y más cosas.
  for (let prop in document.body.childNodes) alert(prop);
</script>
</body>
```

Hermanos y el padre

Los *hermanos* son nodos que son hijos del mismo padre.

Por ejemplo, aquí `<head>` y `<body>` son hermanos:

```
<html>
  <head>...</head><body>...</body>
</html>
```

- `<body>` se dice que es el hermano “siguiente” o a la “derecha” de `<head>`,
- `<head>` se dice que es el hermano “anterior” o a la “izquierda” de `<body>`.

El hermano siguiente está en la propiedad `nextSibling` y el anterior – en `previousSibling`.

El padre está disponible en `parentNode`.

Por ejemplo:

```
// el padre de <body> es <html>
alert( document.body.parentNode === document.documentElement ); // verdadero

// después de <head> va <body>
alert( document.head.nextSibling ); // HTMLBodyElement

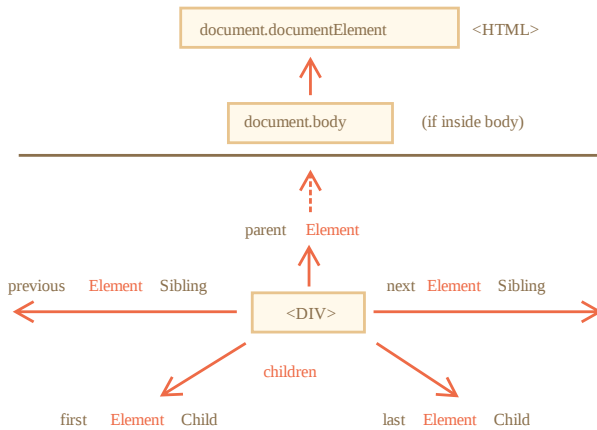
// antes de <body> va <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

Navegación solo por elementos

Las propiedades de navegación enumeradas abajo se refieren a *todos* los nodos. Por ejemplo, en `childNodes` podemos ver nodos de texto, nodos elementos; y si existen, incluso los nodos de comentarios.

Pero para muchas tareas no queremos los nodos de texto o comentarios. Queremos manipular el nodo que representa las etiquetas y formularios de la estructura de la página.

Así que vamos a ver más enlaces de navegación que solo tienen en cuenta los *elementos nodos*:



Los enlaces son similares a los de arriba, solo que tienen dentro la palabra `Element` :

- `children` – solo esos hijos que tienen el elemento nodo.
- `firstElementChild`, `lastElementChild` – el primer y el último elemento hijo.
- `previousElementSibling`, `nextElementSibling` – elementos vecinos.
- `parentElement` – elemento padre.

i ¿Por qué `parentElement` ? ¿Puede el padre *no* ser un elemento?

La propiedad `parentElement` devuelve el “elemento” padre, mientras `parentNode` devuelve “cualquier nodo” padre. Estas propiedades son normalmente las mismas: ambas seleccionan el padre.

Con la excepción de `document.documentElement` :

```
alert( document.documentElement.parentNode ); // documento
alert( document.documentElement.parentElement ); // null
```

La razón es que el nodo raíz `document.documentElement` (`<html>`) tiene a `document` como su padre. Pero `document` no es un elemento nodo, por lo que `parentNode` lo devuelve y `parentElement` no lo hace.

Este detalle puede ser útil cuando queramos navegar hacia arriba desde cualquier elemento `elem` al `<html>`, pero no hacia el `document` :

```
while(elem = elem.parentElement) { // sube hasta <html>
  alert( elem );
}
```

Vamos a modificar uno de los ejemplos de arriba: reemplaza `childNodes` por `children`. Ahora enseña solo elementos:

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
```



```
</body>
</html>
```

Más enlaces: tablas

Hasta ahora hemos descrito las propiedades de navegación básicas.

Ciertos tipos de elementos del DOM pueden tener propiedades adicionales, específicas de su tipo, por conveniencia.

Las tablas son un gran ejemplo de ello, y representan un particular caso importante:

El elemento **<table>** soporta estas propiedades (añadidas a las que hemos dado anteriormente):

- `table.rows` – la colección de elementos `<tr>` de la tabla.
- `table.caption/tHead/tFoot` – referencias a los elementos `<caption>`, `<thead>`, `<tfoot>`.
- `table.tBodies` – la colección de elementos `<tbody>` (pueden ser muchos según el estándar pero siempre habrá al menos uno – aunque no esté en el HTML el navegador lo pondrá en el DOM).

<thead>, **<tfoot>**, **<tbody>** estos elementos proporcionan las propiedades de las `filas`.

- `tbody.rows` – la colección dentro de `<tr>`.

<tr>:

- `tr.cells` – la colección de celdas `<td>` y `<th>` dentro del `<tr>` dado.
- `tr.sectionRowIndex` – la posición (índice) del `<tr>` dado dentro del `<thead>/<tbody>/<tfoot>` adjunto.
- `tr.rowIndex` – el número de `<tr>` en la tabla en su conjunto (incluyendo todas las filas de una tabla).

<td> and <th>:

- `td.cellIndex` – el número de celdas dentro del adjunto `<tr>`.

Un ejemplo de uso:

```
<table id="table">
  <tr>
    <td>one</td><td>two</td>
  </tr>
  <tr>
    <td>three</td><td>four</td>
  </tr>
</table>

<script>
  // seleccionar td con "dos" (primera fila, segunda columna)
  let td = table.rows[0].cells[1];
  td.style.backgroundColor = "red"; // destacarlo
</script>
```

La especificación: [tabular data](#).

También hay propiedades de navegación adicionales para los formularios HTML. Las veremos más adelante cuando empecemos a trabajar con los formularios.

Resumen

Dado un nodo del DOM, podemos ir a sus inmediatos vecinos utilizando las propiedades de navegación.

Hay dos conjuntos principales de ellas:

- Para todos los nodos: `parentNode`, `childNodes`, `firstChild`, `lastChild`, `previousSibling`, `nextSibling`.
- Para los nodos elementos: `parentElement`, `children`, `firstElementChild`, `lastElementChild`, `previousElementSibling`, `nextElementSibling`.

Algunos tipos de elementos del DOM, por ejemplo las tablas, proveen propiedades adicionales y colecciones para acceder a su contenido.

✓ Tareas

DOM children

importancia: 5

Mira esta página:

```
<html>
<body>
  <div>Users:</div>
  <ul>
    <li>John</li>
    <li>Pete</li>
  </ul>
</body>
</html>
```

Para cada una de las siguientes preguntas, da al menos una forma de cómo acceder a ellos:

- ¿El nodo `<div>` del DOM?
- ¿El nodo `` del DOM?
- El segundo `` (con Pete)?

[A solución](#)

La pregunta de los hermanos

importancia: 5

Si `elem` – es un elemento nodo arbitrario del DOM...

- ¿Es cierto que `elem.lastChild.nextSibling` siempre es `null`?
- ¿Es cierto que `elem.children[0].previousSibling` siempre es `null`?

[A solución](#)

Seleccionar todas las celdas diagonales

importancia: 5

Escribe el código para pintar todas las celdas diagonales de rojo.

Necesitarás obtener todas las `<td>` de la `<table>` y pintarlas usando el código:

```
// td debe ser la referencia a la celda de la tabla
td.style.backgroundColor = 'red';
```

El resultado debe ser:

1:1	2:1	3:1	4:1	5:1
1:2	2:2	3:2	4:2	5:2
1:3	2:3	3:3	4:3	5:3
1:4	2:4	3:4	4:4	5:4
1:5	2:5	3:5	4:5	5:5

[Abrir un entorno controlado para la tarea.](#) ↗

Buscar: getElement*, querySelector*

Las propiedades de navegación del DOM son ideales cuando los elementos están cerca unos de otros. Pero, ¿y si no lo están? ¿Cómo obtener un elemento arbitrario de la página?

Para estos casos existen métodos de búsqueda adicionales.

document.getElementById o sólo id

Si un elemento tiene el atributo `id`, podemos obtener el elemento usando el método `document.getElementById(id)`, sin importar dónde se encuentre.

Por ejemplo:

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // obtener el elemento
  let elem = document.getElementById('elem');

  // hacer que su fondo sea rojo
  elem.style.background = 'red';
</script>
```

Existe además una variable global nombrada por el `id` que hace referencia al elemento:

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // elem es una referencia al elemento del DOM con id="elem"
  elem.style.background = 'red';

  // id="elem-content" tiene un guión en su interior, por lo que no puede ser un nombre de variable
  // ...pero podemos acceder a él usando corchetes: window['elem-content']
</script>
```

...Esto es a menos que declaremos una variable de JavaScript con el mismo nombre, entonces ésta tiene prioridad:

```
<div id="elem"></div>

<script>
  let elem = 5; // ahora elem es 5, no una referencia a <div id="elem">

  alert(elem); // 5
</script>
```

⚠ Por favor, no utilice variables globales nombradas por id para acceder a los elementos

Este comportamiento se encuentra descrito [en la especificación](#) , por lo que es una especie de estándar. Pero está soportado principalmente para compatibilidad.

El navegador intenta ayudarnos mezclando espacios de nombres (*namespaces*) de JS y DOM. Esto está bien para los scripts simples, incrustados en HTML, pero generalmente no es una buena práctica. Puede haber conflictos de nombres. Además, cuando uno lee el código de JS y no tiene el HTML a la vista, no es obvio de dónde viene la variable.

Aquí en el tutorial usamos `id` para referirnos directamente a un elemento por brevedad, cuando es obvio de dónde viene el elemento.

En la vida real `document.getElementById` es el método preferente.

i El `id` debe ser único

El `id` debe ser único. Sólo puede haber en todo el documento un elemento con un `id` determinado.

Si hay múltiples elementos con el mismo `id`, entonces el comportamiento de los métodos que lo usan es impredecible, por ejemplo `document.getElementById` puede devolver cualquiera de esos elementos al azar. Así que, por favor, sigan la regla y mantengan el `id` único.

⚠ Sólo `document.getElementById`, no `anyElem.getElementById`

El método `getElementById` sólo puede ser llamado en el objeto `document`. Busca el `id` dado en todo el documento.

querySelectorAll

Sin duda el método más versátil, `elem.querySelectorAll(css)` devuelve todos los elementos dentro de `elem` que coinciden con el selector CSS dado.

Aquí buscamos todos los elementos `` que son los últimos hijos:

```
<ul>
  <li>La</li>
  <li>prueba</li>
</ul>
<ul>
  <li>ha</li>
  <li>pasado</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "prueba", "pasado"
  }
</script>
```

Este método es muy poderoso, porque se puede utilizar cualquier selector de CSS.

i También se pueden usar pseudoclases

Las pseudoclases como `:hover` (cuando el cursor sobrevuela el elemento) y `:active` (cuando hace clic con el botón principal) también son soportadas. Por ejemplo, `document.querySelectorAll(':hover')` devolverá una colección de elementos sobre los que el puntero hace hover en ese momento (en orden de anidación: desde el más exterior `<html>` hasta el más anidado).

querySelector

La llamada a `elem.querySelector(css)` devuelve el primer elemento para el selector CSS dado.

En otras palabras, el resultado es el mismo que `elem.querySelectorAll(css)[0]`, pero este último busca *todos* los elementos y elige uno, mientras que `elem.querySelector` sólo busca uno. Así que es más rápido y también más corto de escribir.

matches

Los métodos anteriores consistían en buscar en el DOM.

El `elem.matches(css)` [↗](#) no busca nada, sólo comprueba si el `elem` coincide con el selector CSS dado. Devuelve `true` o `false`.

Este método es útil cuando estamos iterando sobre elementos (como en un array) y tratando de filtrar los que nos interesan.

Por ejemplo:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // puede ser cualquier colección en lugar de document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("La referencia del archivo: " + elem.href );
    }
  }
</script>
```

closest

Los *ancestros* de un elemento son: el padre, el padre del padre, su padre y así sucesivamente. Todos los ancestros juntos forman la cadena de padres desde el elemento hasta la cima.

El método `elem.closest(css)` busca el ancestro más cercano que coincide con el selector CSS. El propio `elem` también se incluye en la búsqueda.

En otras palabras, el método `closest` sube del elemento y comprueba cada uno de los padres. Si coincide con el selector, entonces la búsqueda se detiene y devuelve dicho ancestro.

Por ejemplo:

```
<h1>Contenido</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Capítulo 1</li>
    <li class="chapter">Capítulo 1</li>
  </ul>
</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (porque h1 no es un ancestro)
</script>
```

getElementsBy*

También hay otros métodos que permiten buscar nodos por una etiqueta, una clase, etc.

Hoy en día, son en su mayoría historia, ya que `querySelector` es más poderoso y corto de escribir.

Aquí los cubrimos principalmente por completar el temario, aunque todavía se pueden encontrar en scripts antiguos.

- `elem.getElementsByTagName(tag)` busca elementos con la etiqueta dada y devuelve una colección con ellos. El parámetro `tag` también puede ser un asterisco `"*"` para "cualquier etiqueta".
- `elem.getElementsByClassName(className)` devuelve elementos con la clase dada.
- `document.getElementsByName(name)` devuelve elementos con el atributo `name` dado, en todo el documento. Muy raramente usado.

Por ejemplo:

```
// obtener todos los divs del documento
let divs = document.getElementsByTagName('div');
```

Para encontrar todas las etiquetas `input` dentro de una tabla:

```
<table id="table">
  <tr>
    <td>Su edad:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> menos de 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> de 18 a 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> más de 60
      </label>
    </td>
  </tr>
</table>

<script>
  let inputs = table.getElementsByTagName('input');

  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

⚠ ¡No olvides la letra "s" !

Los desarrolladores novatos a veces olvidan la letra `"s"`. Esto es, intentan llamar a `getElementByTagName` en vez de a `getElementsByTagName`.

La letra `"s"` no se encuentra en `getElementById` porque devuelve sólo un elemento. But `getElementsByTagName` devuelve una colección de elementos, de ahí que tenga la `"s"`.

⚠ ¡Devuelve una colección, no un elemento!

Otro error muy extendido entre los desarrolladores novatos es escribir:

```
// no funciona
document.getElementsByTagName('input').value = 5;
```

Esto no funcionará, porque toma una *colección* de inputs y le asigna el valor a ella en lugar de a los elementos dentro de ella.

En dicho caso, deberíamos iterar sobre la colección o conseguir un elemento por su índice y luego asignarlo así:

```
// debería funcionar (si hay un input)
document.getElementsByTagName('input')[0].value = 5;
```

Buscando elementos `.article`:

```
<form name="my-form">
  <div class="article">Artículo</div>
  <div class="long article">Artículo largo</div>
</form>

<script>
  // encontrar por atributo de nombre
  let form = document.getElementsByName('my-form')[0];

  // encontrar por clase dentro del formulario
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, encontró dos elementos con la clase "article"
</script>
```

Colecciones vivas

Todos los métodos `"getElementsBy*"` devuelven una colección *viva* (*live collection*). Tales colecciones siempre reflejan el estado actual del documento y se “auto-actualizan” cuando cambia.

En el siguiente ejemplo, hay dos scripts.

1. El primero crea una referencia a la colección de `<div>`. Por ahora, su longitud es `1`.
2. El segundo script se ejecuta después de que el navegador se encuentre con otro `<div>`, por lo que su longitud es de `2`.

```
<div>Primer div</div>

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>

<div>Segundo div</div>

<script>
  alert(divs.length); // 2
</script>
```

Por el contrario, `querySelectorAll` devuelve una colección *estática*. Es como un array de elementos fijos.

Si lo utilizamos en lugar de `getElementsByTagName`, entonces ambos scripts dan como resultado `1`:

```
<div>Primer div</div>
```

```

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>

<div>Segundo div</div>

<script>
  alert(divs.length); // 1
</script>

```

Ahora podemos ver fácilmente la diferencia. La colección estática no aumentó después de la aparición de un nuevo `div` en el documento.

Resumen

Hay 6 métodos principales para buscar nodos en el DOM:

Método	Busca por...	¿Puede llamar a un elemento?	¿Vivo?
<code>querySelector</code>	selector CSS	✓	-
<code>querySelectorAll</code>	selector CSS	✓	-
<code>getElementById</code>	<code>id</code>	-	-
<code>getElementsByName</code>	<code>name</code>	-	✓
<code>getElementsByTagName</code>	etiqueta o <code>'*'</code>	✓	✓
<code>getElementsByClassName</code>	<code>class</code>	✓	✓

Los más utilizados son `querySelector` y `querySelectorAll`, pero `getElementBy*` puede ser de ayuda esporádicamente o encontrarse en scripts antiguos.

Aparte de eso:

- Existe `elem.matches(css)` para comprobar si `elem` coincide con el selector CSS dado.
- Existe `elem.closest(css)` para buscar el ancestro más cercano que coincida con el selector CSS dado. El propio `elem` también se comprueba.

Y mencionemos un método más para comprobar la relación hijo-padre, ya que a veces es útil:

- `elemA.contains(elemB)` devuelve `true` si `elemB` está dentro de `elemA` (un descendiente de `elemA`) o cuando `elemA==elemB`.

✓ Tareas

Buscar elementos

importancia: 4

Aquí está el documento con la tabla y el formulario.

¿Cómo encontrar?...

1. La tabla con `id="age-table"`.
2. Todos los elementos `label` dentro de la tabla (debería haber 3).
3. El primer `td` en la tabla (con la palabra "Age").
4. El `form` con `name="search"`.
5. El primer `input` en ese formulario.
6. El último `input` en ese formulario.

Abra la página [table.html](#) en una ventana separada y haga uso de las herramientas del navegador.

[A solución](#)

Propiedades del nodo: tipo, etiqueta y contenido

Echemos un mirada más en profundidad a los nodos DOM.

En este capítulo veremos más sobre cuáles son y aprenderemos sus propiedades más utilizadas.

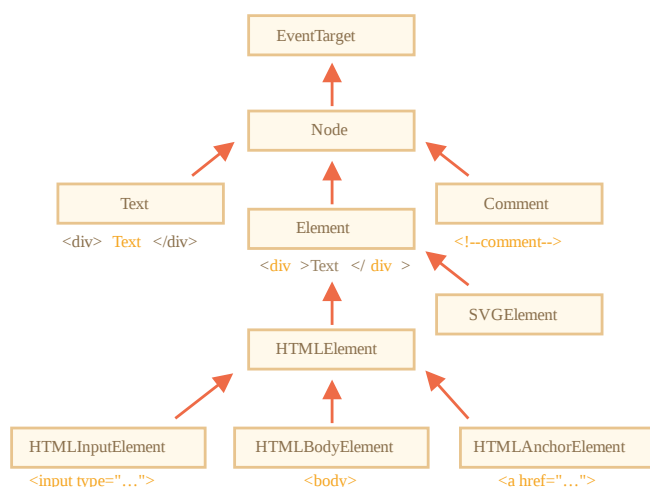
Clases de nodo DOM

Los diferentes nodos DOM pueden tener diferentes propiedades. Por ejemplo, un nodo de elemento correspondiente a la etiqueta `<a>` tiene propiedades relacionadas con el enlace, y el correspondiente a `<input>` tiene propiedades relacionadas con la entrada y así sucesivamente. Los nodos de texto no son lo mismo que los nodos de elementos. Pero también hay propiedades y métodos comunes entre todos ellos, porque todas las clases de nodos DOM forman una única jerarquía.

Cada nodo DOM pertenece a la clase nativa correspondiente.

La raíz de la jerarquía es [EventTarget](#), que es heredada por [Node](#), y otros nodos DOM heredan de él.

Aquí está la imagen, con las explicaciones a continuación:



Las clases son:

- [EventTarget](#) – es la clase raíz “abstracta”. Los objetos de esa clase nunca se crean. Sirve como base, por lo que todos los nodos DOM soportan los llamados “eventos”, los estudiaremos más adelante.
- [Node](#) – también es una clase “abstracta”, que sirve como base para los nodos DOM. Proporciona la funcionalidad del árbol principal: `parentNode`, `nextSibling`, `childNodes` y así sucesivamente (son getters). Los objetos de la clase `Node` nunca se crean. Pero hay clases de nodos concretas que heredan de él, a saber: `Text` para nodos de texto, `Element` para nodos de elementos y otros más exóticos como `Comment` para nodos de comentarios.
- [Element](#) – es una clase base para elementos DOM. Proporciona navegación a nivel de elemento como `nextElementSibling`, `children` y métodos de búsqueda como `getElementsByTagName`, `querySelector`. Un navegador admite no solo HTML, sino también XML y SVG. La clase `Element` sirve como base para clases más específicas: `SVGElement`, `XMLElement` y `HTMLElement`.
- [HTMLElement](#) – es finalmente la clase básica para todos los elementos HTML. Es heredado por elementos HTML concretos:
 - [HTMLInputElement](#) – la clase para elementos `<input>`,
 - [HTMLBodyElement](#) – la clase para los elementos `<body>`,
 - [HTMLAnchorElement](#) – la clase para elementos `<a>`,
 - ...y así sucesivamente, cada etiqueta tiene su propia clase que puede proporcionar propiedades y métodos específicos.

Entonces, el conjunto completo de propiedades y métodos de un nodo dado viene como resultado de la herencia.

Por ejemplo, consideremos el objeto DOM para un elemento `<input>`. Pertenece a la clase [HTMLInputElement](#).

Obtiene propiedades y métodos como una superposición de (enumerados en orden de herencia):

- `HTMLInputElement` – esta clase proporciona propiedades específicas de entrada,
- `HTMLElement` – proporciona métodos de elementos HTML comunes (y getters/setters),
- `Element` – proporciona métodos de elementos genéricos,
- `Node` – proporciona propiedades comunes del nodo DOM,
- `EventTarget` – da el apoyo para eventos (a cubrir),
- ...y finalmente hereda de `Object`, por lo que también están disponibles métodos de “objeto simple” como `hasOwnProperty`.

Para ver el nombre de la clase del nodo DOM, podemos recordar que un objeto generalmente tiene la propiedad `constructor`. Hace referencia al constructor de la clase, y `constructor.name` es su nombre:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

...O podemos simplemente usar `toString`:

```
alert( document.body ); // [object HTMLBodyElement]
```

También podemos usar `instanceof` para verificar la herencia:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

Como podemos ver, los nodos DOM son objetos regulares de JavaScript. Usan clases basadas en prototipos para la herencia.

Eso también es fácil de ver al generar un elemento con `console.dir(elem)` en un navegador. Allí, en la consola, puede ver `HTMLElement.prototype`, `Element.prototype` y así sucesivamente.

`console.dir(elem)` versus `console.log(elem)`


La mayoría de los navegadores admiten dos comandos en sus herramientas de desarrollo: `console.log` y `console.dir`. Envían sus argumentos a la consola. Para los objetos JavaScript, estos comandos suelen hacer lo mismo.

Pero para los elementos DOM son diferentes:

- `console.log(elem)` muestra el árbol DOM del elemento.
- `console.dir(elem)` muestra el elemento como un objeto DOM, es bueno para explorar sus propiedades.

Inténtalo en `document.body`.

IDL en la especificación

En la especificación, las clases DOM no se describen mediante JavaScript, sino con un [Lenguaje de descripción de interfaz](#)  (IDL) especial, que suele ser fácil de entender.

En IDL, todas las propiedades están precedidas por sus tipos. Por ejemplo, `DOMString`, `boolean` y así sucesivamente.

Aquí hay un extracto, con comentarios:

```
// Definir HTMLInputElement
// Los dos puntos ":" significan que HTMLInputElement hereda de HTMLElement
interface HTMLInputElement: HTMLElement {
    // aquí van las propiedades y métodos de los elementos <input>


    // "DOMString" significa que el valor de una propiedad es un string
    attribute DOMString accept;
    attribute DOMString alt;
    attribute DOMString autocomplete;
    attribute DOMString value;

    // Propiedad de valor booleano (true/false)
    attribute boolean autofocus;
    ...
    // ahora el método: "void" significa que el método no devuelve ningún valor
    void select();
    ...
}
```

La propiedad “nodeType”

La propiedad `nodeType` proporciona una forma “anticuada” más de obtener el “tipo” de un nodo DOM.

Tiene un valor numérico:

- `elem.nodeType == 1` para nodos de elementos,
- `elem.nodeType == 3` para nodos de texto,
- `elem.nodeType == 9` para el objeto de documento,
- hay algunos otros valores en [la especificación](#) .

Por ejemplo:

```
<body>
<script>
let elem = document.body;

// vamos a examinar lo que es
alert(elem.nodeType); // 1 => elemento

// Y el primer hijo es...
alert(elem.firstChild.nodeType); // 3 => texto

// para el objeto de tipo documento, el tipo es 9
alert( document.nodeType ); // 9
</script>
</body>
```

En los scripts modernos, podemos usar `instanceof` y otras pruebas basadas en clases para ver el tipo de nodo, pero a veces `nodeType` puede ser más simple. Solo podemos leer `nodeType`, no cambiarlo.

Tag: nodeName y tagName

Dado un nodo DOM, podemos leer su nombre de etiqueta en las propiedades de `nodeName` o `tagName`:

Por ejemplo:

```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

¿Hay alguna diferencia entre `tagName` y `nodeName` ?

Claro, la diferencia se refleja en sus nombres, pero de hecho es un poco sutil.

- La propiedad `tagName` existe solo para los nodos `Element` .
- El `nodeName` se define para cualquier `Node` :
 - para los elementos, significa lo mismo que `tagName` .
 - para otros tipos de nodo (texto, comentario, etc.) tiene una cadena con el tipo de nodo.

En otras palabras, `tagName` solo es compatible con los nodos de elementos (ya que se origina en la clase `Element`), mientras que `nodeName` puede decir algo sobre otros tipos de nodos.

Por ejemplo, comparemos `tagName` y `nodeName` para `document` y un nodo de comentario:

```
<body><!-- comentario -->

<script>
  // para comentarios
  alert( document.body.firstChild.tagName ); // undefined (no es un elemento)
  alert( document.body.firstChild.nodeName ); // #comment

  // para documentos
  alert( document.tagName ); // undefined (no es un elemento)
  alert( document.nodeName ); // #document
</script>
</body>
```

Si solo tratamos con elementos, entonces podemos usar tanto `tagName` como `nodeName` – no hay diferencia.

El nombre de la etiqueta siempre está en mayúsculas, excepto en el modo XML

El navegador tiene dos modos de procesar documentos: HTML y XML. Por lo general, el modo HTML se usa para páginas web. El modo XML está habilitado cuando el navegador recibe un documento XML con el encabezado: `Content-Type: application/xml+xhtml` .

En el modo HTML, `tagName/nodeName` siempre está en mayúsculas: es `BODY` ya sea para `<body>` o `<BoDy>` .

En el modo XML, el caso se mantiene “tal cual”. Hoy en día, el modo XML rara vez se usa.

innerHTML: los contenidos

La propiedad `innerHTML` [↗](#) permite obtener el HTML dentro del elemento como un string.

También podemos modificarlo. Así que es una de las formas más poderosas de cambiar la página.

El ejemplo muestra el contenido de `document.body` y luego lo reemplaza por completo:

```
<body>
  <p>Un párrafo</p>
  <div>Un div</div>

  <script>
    alert( document.body.innerHTML ); // leer el contenido actual
    document.body.innerHTML = 'El nuevo BODY!'; // reemplazar
  </script>

</body>
```

Podemos intentar insertar HTML no válido, el navegador corregirá nuestros errores:

```
<body>

<script>
  document.body.innerHTML = '<b>prueba'; // olvidé cerrar la etiqueta
  alert( document.body.innerHTML ); // <b>prueba</b> (arreglado)
</script>

</body>
```

i Los scripts no se ejecutan

Si `innerHTML` inserta una etiqueta `<script>` en el documento, se convierte en parte de HTML, pero no se ejecuta.

Cuidado: “innerHTML+=” hace una sobrescritura completa

Podemos agregar HTML a un elemento usando `elem.innerHTML+="more html"`.

Así:

```
chatDiv.innerHTML += "<div>Hola<img src='smile.gif' /> !</div>";
chatDiv.innerHTML += "¿Cómo vas?";
```

Pero debemos tener mucho cuidado al hacerlo, porque lo que está sucediendo *no* es una adición, sino una sobrescritura completa.

Técnicamente, estas dos líneas hacen lo mismo:

```
elem.innerHTML += "...";
// es una forma más corta de escribir:
elem.innerHTML = elem.innerHTML + "..."
```

En otras palabras, `innerHTML+=` hace esto:

1. Se elimina el contenido antiguo.
2. En su lugar, se escribe el nuevo `innerHTML` (una concatenación del antiguo y el nuevo).

Como el contenido se “pone a cero” y se reescribe desde cero, todas las imágenes y otros recursos se volverán a cargar..

En el ejemplo de `chatDiv` arriba, la línea `chatDiv.innerHTML+="¿Cómo va?"` recrea el contenido HTML y recarga `smile.gif` (con la esperanza de que esté en caché). Si `chatDiv` tiene muchos otros textos e imágenes, entonces la recarga se vuelve claramente visible.

También hay otros efectos secundarios. Por ejemplo, si el texto existente se seleccionó con el mouse, la mayoría de los navegadores eliminarán la selección al reescribir `innerHTML`. Y si había un `<input>` con un texto ingresado por el visitante, entonces el texto será eliminado. Y así.

Afortunadamente, hay otras formas de agregar HTML además de `innerHTML`, y las estudiaremos pronto.

outerHTML: HTML completo del elemento

La propiedad `outerHTML` contiene el HTML completo del elemento. Eso es como `innerHTML` más el elemento en sí.

He aquí un ejemplo:

```
<div id="elem">Hola <b>Mundo</b></div>

<script>
```

```
alert(elem.outerHTML); // <div id="elem">Hola <b>Mundo</b></div>
</script>
```

Cuidado: a diferencia de `innerHTML`, escribir en `outerHTML` no cambia el elemento. En cambio, lo reemplaza en el DOM.

Sí, suena extraño, y es extraño, por eso hacemos una nota aparte al respecto aquí. Echa un vistazo.

Considera el ejemplo:

```
<div>¡Hola, mundo!</div>

<script>
  let div = document.querySelector('div');

  // reemplaza div.outerHTML con <p>...</p>
  div.outerHTML = '<p>Un nuevo elemento</p>'; // (*)

  // ¡Guauu! ¡'div' sigue siendo el mismo!
  alert(div.outerHTML); // <div>¡Hola, mundo!</div> (**)
</script>
```

Parece realmente extraño, ¿verdad?

En la línea (*) reemplazamos `div` con `<p>Un nuevo elemento</p>`. En el documento externo (el DOM) podemos ver el nuevo contenido en lugar del `<div>`. Pero, como podemos ver en la línea (**), ¡el valor de la antigua variable `div` no ha cambiado!

La asignación `outerHTML` no modifica el elemento DOM (el objeto al que hace referencia, en este caso, la variable 'div'), pero lo elimina del DOM e inserta el nuevo HTML en su lugar.

Entonces, lo que sucedió en `div.outerHTML=...` es:

- `div` fue eliminado del documento.
- Otro fragmento de HTML `<p>Un nuevo elemento</p>` se insertó en su lugar.
- `div` todavía tiene su antiguo valor. El nuevo HTML no se guardó en ninguna variable.

Es muy fácil cometer un error aquí: modificar `div.outerHTML` y luego continuar trabajando con `div` como si tuviera el nuevo contenido. Pero no es así. Esto es correcto para `innerHTML`, pero no para `outerHTML`.

Podemos escribir en `elem.outerHTML`, pero debemos tener en cuenta que no cambia el elemento en el que estamos escribiendo ('elem'). En su lugar, coloca el nuevo HTML en su lugar. Podemos obtener referencias a los nuevos elementos consultando el DOM.

nodeValue/data: contenido del nodo de texto

La propiedad `innerHTML` solo es válida para los nodos de elementos.

Otros tipos de nodos, como los nodos de texto, tienen su contraparte: propiedades `nodeValue` y `data`. Estas dos son casi iguales para uso práctico, solo hay pequeñas diferencias de especificación. Entonces usaremos `data`, porque es más corto.

Un ejemplo de lectura del contenido de un nodo de texto y un comentario:

```
<body>
  Hola
  <!-- Comentario -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hola

    let comment = text.nextSibling;
    alert(comment.data); // Comentario
  </script>
</body>
```

Para los nodos de texto podemos imaginar una razón para leerlos o modificarlos, pero ¿por qué comentarios?

A veces, los desarrolladores incorporan información o instrucciones de plantilla en HTML, así:

```
<!-- if isAdmin -->
  <div>¡Bienvenido, administrador!</div>
<!-- /if -->
```

...Entonces JavaScript puede leerlo desde la propiedad `data` y procesar las instrucciones integradas.

textContent: texto puro

El `textContent` proporciona acceso al *texto* dentro del elemento: solo texto, menos todas las `<tags>`.

Por ejemplo:

```
<div id="news">
  <h1>¡Titular!</h1>
  <p>¡Los marcianos atacan a la gente!</p>
</div>

<script>
  // ¡Titular! ¡Los marcianos atacan a la gente!
  alert(news.textContent);
</script>
```

Como podemos ver, solo se devuelve texto, como si todas las `<etiquetas>` fueran recortadas, pero el texto en ellas permaneció.

En la práctica, rara vez se necesita leer este tipo de texto.

Escribir en `textContent` es mucho más útil, porque permite escribir texto de “forma segura”.

Digamos que tenemos un string arbitrario, por ejemplo, ingresado por un usuario, y queremos mostrarlo.

- Con `innerHTML` lo tendremos insertado “como HTML”, con todas las etiquetas HTML.
- Con `textContent` lo tendremos insertado “como texto”, todos los símbolos se tratan literalmente.

Compara los dos:

```
<div id="elem1"></div>
<div id="elem2"></div>

<script>
  let name = prompt("¿Cuál es tu nombre?", "<b>¡Winnie-Pooh!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

1. El primer `<div>` obtiene el nombre “como HTML”: todas las etiquetas se convierten en etiquetas, por lo que vemos el nombre en negrita.
2. El segundo `<div>` obtiene el nombre “como texto”, así que literalmente vemos `¡Winnie-Pooh!`.

En la mayoría de los casos, esperamos el texto de un usuario y queremos tratarlo como texto. No queremos HTML inesperado en nuestro sitio. Una asignación a `textContent` hace exactamente eso.

La propiedad “hidden”

El atributo “hidden” y la propiedad DOM especifican si el elemento es visible o no.

Podemos usarlo en HTML o asignarlo usando JavaScript, así:

```

<div>Ambos divs a continuación están ocultos</div>

<div hidden>Con el atributo "hidden"</div>

<div id="elem">JavaScript asignó la propiedad "hidden"</div>

<script>
  elem.hidden = true;
</script>

```

Técnicamente, `hidden` funciona igual que `style="display:none"`. Pero es más corto de escribir.

Aquí hay un elemento parpadeante:

```

<div id="elem">Un elemento parpadeante</div>

<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>

```

Más propiedades

Los elementos DOM también tienen propiedades adicionales, en particular aquellas que dependen de la clase:

- `value` – el valor para `<input>`, `<select>` y `<textarea>` (`HTMLInputElement`, `HTMLSelectElement` ...).
- `href` – el “href” para `` (`HTMLAnchorElement`).
- `id` – el valor del atributo “id”, para todos los elementos (`HTMLElement`).
- ...y mucho más...

Por ejemplo:

```

<input type="text" id="elem" value="value">

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // value
</script>

```

La mayoría de los atributos HTML estándar tienen la propiedad DOM correspondiente, y podemos acceder a ella así.

Si queremos conocer la lista completa de propiedades admitidas para una clase determinada, podemos encontrarlas en la especificación. Por ejemplo, `HTMLInputElement` está documentado en <https://html.spec.whatwg.org/#htmlinputelement> .

O si nos gustaría obtenerlos rápidamente o estamos interesados en una especificación concreta del navegador, siempre podemos generar el elemento usando `console.dir(elem)` y leer las propiedades. O explora las “propiedades DOM” en la pestaña Elements de las herramientas de desarrollo del navegador.

Resumen

Cada nodo DOM pertenece a una determinada clase. Las clases forman una jerarquía. El conjunto completo de propiedades y métodos proviene de la herencia.

Las propiedades principales del nodo DOM son:

`nodeType`

Podemos usarla para ver si un nodo es un texto o un elemento. Tiene un valor numérico: `1` para elementos, `3` para nodos de texto y algunos otros para otros tipos de nodos. Solo lectura.

`nodeName/tagName`

Para los elementos, nombre de la etiqueta (en mayúsculas a menos que esté en modo XML). Para los nodos que no son elementos, `nodeName` describe lo que es. Solo lectura.

innerHTML

El contenido HTML del elemento. Puede modificarse.

outerHTML

El HTML completo del elemento. Una operación de escritura en `elem.outerHTML` no toca a `elem` en sí. En su lugar, se reemplaza con el nuevo HTML en el contexto externo.

nodeValue/data

El contenido de un nodo que no es un elemento (text, comment). Estos dos son casi iguales, usualmente usamos `data`. Puede modificarse.

textContent

El texto dentro del elemento: HTML menos todas las `<tags>`. Escribir en él coloca el texto dentro del elemento, con todos los caracteres especiales y etiquetas tratados exactamente como texto. Puede insertar de forma segura texto generado por el usuario y protegerse de inserciones HTML no deseadas.

hidden

Cuando se establece en `true`, hace lo mismo que CSS `display:none`.

Los nodos DOM también tienen otras propiedades dependiendo de su clase. Por ejemplo, los elementos `<input>` (`HTMLInputElement`) admiten `value`, `type`, mientras que los elementos `<a>` (`HTMLAnchorElement`) admiten `href`, etc. La mayoría de los atributos HTML estándar tienen una propiedad DOM correspondiente.

Sin embargo, los atributos HTML y las propiedades DOM no siempre son iguales, como veremos en el próximo capítulo.

Tareas

Contar los descendientes

importancia: 5

Hay un árbol estructurado como `ul/li` anidado.

Escribe el código que para cada `` muestra:

1. ¿Cuál es el texto dentro de él (sin el subárbol)?
2. El número de `` anidados: todos los descendientes, incluidos los profundamente anidados.

[Demo en nueva ventana](#) ↗

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

¿Qué hay en nodeType?

importancia: 5

¿Qué muestra el script?

```
<html>

<body>
  <script>
    alert(document.body.lastChild.nodeType);
  </script>
</body>
```

```
</html>
```

[A solución](#)

Etiqueta en comentario

importancia: 3

¿Qué muestra este código?

```
<script>
  let body = document.body;

  body.innerHTML = "<!--" + body.tagName + "-->";

  alert( body.firstChild.data ); // ¿qué hay aquí?
</script>
```

[A solución](#)

¿Dónde está el "document" en la jerarquía?

importancia: 4

¿A qué clase pertenece el `document` ?

¿Cuál es su lugar en la jerarquía DOM?

¿Hereda de `Node` o `Element` , o tal vez `HTMLElement` ?

[A solución](#)

Atributos y propiedades

Cuando el navegador carga la página, “lee” (o “parser”(analiza en inglés)) el HTML y genera objetos DOM a partir de él. Para los nodos de elementos, la mayoría de los atributos HTML estándar se convierten automáticamente en propiedades de los objetos DOM.

Por ejemplo, si la etiqueta es `<body id="page">` , entonces el objeto DOM tiene `body.id="page"` .

¡Pero el mapeo de propiedades y atributos no es uno a uno! En este capítulo, prestaremos atención para separar estas dos nociones, para ver cómo trabajar con ellos, cuándo son iguales y cuándo son diferentes.

Propiedades DOM

Ya hemos visto propiedades DOM integradas. Hay muchas. Pero técnicamente nadie nos limita, y si no hay suficientes, podemos agregar las nuestras.

Los nodos DOM son objetos JavaScript normales. Podemos alterarlos.

Por ejemplo, creemos una nueva propiedad en `document.body` :

```
document.body.myData = {
  name: 'Cesar',
  title: 'Emperador'
};

alert(document.body.myData.title); // Emperador
```

También podemos agregar un método:

```
document.body.sayTagName = function() {
    alert(this.tagName);
};

document.body.sayTagName(); // BODY (el valor de 'this' en el método es document.body)
```

También podemos modificar prototipos incorporados como `Element.prototype` y agregar nuevos métodos a todos los elementos:

```
Element.prototype.sayHi = function() {
    alert(`Hola, yo soy ${this.tagName}`);
};

document.documentElement.sayHi(); // Hola, yo soy HTML
document.body.sayHi(); // Hola, yo soy BODY
```

Por lo tanto, las propiedades y métodos DOM se comportan igual que los objetos JavaScript normales:

- Pueden tener cualquier valor.
- Distingue entre mayúsculas y minúsculas (escribir `elem.nodeType`, no es lo mismo que `elem.NoDeTyPe`).

Atributos HTML

En HTML, las etiquetas pueden tener atributos. Cuando el navegador analiza el HTML para crear objetos DOM para etiquetas, reconoce los atributos *estándar* y crea propiedades DOM a partir de ellos.

Entonces, cuando un elemento tiene `id` u otro atributo *estándar*, se crea la propiedad correspondiente. Pero eso no sucede si el atributo no es estándar.

Por ejemplo:

```
<body id="test" something="non-standard">
  <script>
    alert(document.body.id); // prueba
    // el atributo no estándar no produce una propiedad
    alert(document.body.something); // undefined
  </script>
</body>
```

Tenga en cuenta que un atributo estándar para un elemento puede ser desconocido para otro. Por ejemplo, `"type"` es estándar para `<input>` ([HTMLInputElement](#)), pero no para `<body>` ([HTMLBodyElement](#)). Los atributos estándar se describen en la especificación para la clase del elemento correspondiente.

Aquí podemos ver esto:

```
<body id="body" type="...">
  <input id="input" type="text">
  <script>
    alert(input.type); // text
    alert(body.type); // undefined: Propiedad DOM no creada, porque no es estándar
  </script>
</body>
```

Entonces, si un atributo no es estándar, no habrá una propiedad DOM para él. ¿Hay alguna manera de acceder a tales atributos?

Claro. Todos los atributos son accesibles usando los siguientes métodos:

- `elem.hasAttribute(nombre)` – comprueba si existe.
- `elem.getAttribute(nombre)` – obtiene el valor.
- `elem.setAttribute(nombre, valor)` – establece el valor.
- `elem.removeAttribute(nombre)` – elimina el atributo.

Estos métodos funcionan exactamente con lo que está escrito en HTML.

También se pueden leer todos los atributos usando `elem.attributes`: una colección de objetos que pertenecen a una clase integrada `Attr` [↗](#), con propiedades `nombre` y `valor`.

Aquí hay una demostración de la lectura de una propiedad no estándar:

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); // no estándar
  </script>
</body>
undefined
```

Los atributos HTML tienen las siguientes características:

- Su nombre no distingue entre mayúsculas y minúsculas (`id` es igual a `ID`).
- Sus valores son siempre strings.

Aquí hay una demostración extendida de cómo trabajar con atributos:

```
<body>
  <div id="elem" about="Elephant"></div>

  <script>
    alert( elem.getAttribute('About') ); // (1) 'Elephant', leyendo

    elem.setAttribute('Test', 123); // (2), escribiendo

    alert( elem.outerHTML ); // (3), ver si el atributo está en HTML (sí)

    for (let attr of elem.attributes) { // (4) listar todo
      alert( `${attr.name} = ${attr.value}` );
    }
  </script>
</body>
```

Tenga en cuenta:

1. `getAttribute('About')` – la primera letra está en mayúscula aquí, y en HTML todo está en minúscula. Pero eso no importa: los nombres de los atributos no distinguen entre mayúsculas y minúsculas.
2. Podemos asignar cualquier cosa a un atributo, pero se convierte en un string. Así que aquí tenemos `"123"` como valor.
3. Todos los atributos, incluidos los que configuramos, son visibles en `outerHTML`.
4. La colección `attributes` es iterable y tiene todos los atributos del elemento (estándar y no estándar) como objetos con propiedades `name` y `value`.

Sincronización de propiedad y atributo

Cuando cambia un atributo estándar, la propiedad correspondiente se actualiza automáticamente (con algunas excepciones) y viceversa.

En el ejemplo a continuación, `id` se modifica como un atributo, y también podemos ver la propiedad cambiada. Y luego lo mismo al revés:

```
<input>

<script>
  let input = document.querySelector('input');

  // atributo => propiedad
  input.setAttribute('id', 'id');
  alert(input.id); // id (actualizado)
```

```
// propiedad => atributo
input.id = 'newId';
alert(input.getAttribute('id')); // newId (actualizado)
</script>
```

Pero hay exclusiones, por ejemplo, `input.value` se sincroniza solo del atributo a la propiedad (atributo => propiedad), pero no de regreso:

```
<input>

<script>
  let input = document.querySelector('input');

  // atributo => propiedad
  input.setAttribute('value', 'text');
  alert(input.value); // text

  // NO propiedad => atributo
  input.value = 'newValue';
  alert(input.getAttribute('value')); // text (¡no actualizado!)
</script>
```

En el ejemplo anterior:

- Cambiar el atributo `value` actualiza la propiedad.
- Pero el cambio de propiedad no afecta al atributo.

Esa “característica” en realidad puede ser útil, porque las acciones del usuario pueden conducir a cambios de `value`, y luego, si queremos recuperar el valor “original” de HTML, está en el atributo.

Las propiedades DOM tienen tipo

Las propiedades DOM no siempre son strings. Por ejemplo, la propiedad `input.checked` (para casillas de verificación) es un booleano:

```
<input id="input" type="checkbox" checked> checkbox

<script>
  alert(input.getAttribute('checked')); // el valor del atributo es: string vacía
  alert(input.checked); // el valor de la propiedad es: true
</script>
```

Hay otros ejemplos. El atributo `style` es un string, pero la propiedad `style` es un objeto:

```
<div id="div" style="color:red;font-size:120%">Hola</div>

<script>
  // string
  alert(div.getAttribute('style')); // color:red;font-size:120%

  // object
  alert(div.style); // [object CSSStyleDeclaration]
  alert(div.style.color); // red
</script>
```

La mayoría de las propiedades son strings.

Muy raramente, incluso si un tipo de propiedad DOM es un string, puede diferir del atributo. Por ejemplo, la propiedad DOM `href` siempre es una URL *completa*, incluso si el atributo contiene una URL relativa o solo un `#hash`.

Aquí hay un ejemplo:

```

<a id="a" href="#hola">link</a>
<script>
  // atributo
  alert(a.getAttribute('href')); // #hola

  // propiedad
  alert(a.href ); // URL completa de http://site.com/page#hola
</script>

```

Si necesitamos el valor de `href` o cualquier otro atributo exactamente como está escrito en el HTML, podemos usar `getAttribute`.

Atributos no estándar, dataset

Cuando escribimos HTML, usamos muchos atributos estándar. Pero, ¿qué pasa con los no personalizados y personalizados? Primero, veamos si son útiles o no. ¿Para qué?

A veces, los atributos no estándar se utilizan para pasar datos personalizados de HTML a JavaScript, o para “marcar” elementos HTML para JavaScript.

Como esto:

```

<!-- marque el div para mostrar "nombre" aquí -->
<div show-info="nombre"></div>
<!-- y "edad" aquí -->
<div show-info="edad"></div>

<script>
  // el código encuentra un elemento con la marca y muestra lo que se solicita
  let user = {
    nombre: "Pete",
    edad: 25
  };

  for(let div of document.querySelectorAll('[show-info]')) {
    // inserta la información correspondiente en el campo
    let field = div.getAttribute('show-info');
    div.innerHTML = user[field]; // primero Pete en "nombre", luego 25 en "edad"
  }
</script>

```

También se pueden usar para diseñar un elemento.

Por ejemplo, aquí para el estado del pedido se usa el atributo `order-state`:

```

<style>
  /* los estilos se basan en el atributo personalizado "order-state" */
  .order[order-state="nuevo"] {
    color: green;
  }

  .order[order-state="pendiente"] {
    color: blue;
  }

  .order[order-state="cancelado"] {
    color: red;
  }
</style>

<div class="order" order-state="nuevo">
  Un nuevo pedido.
</div>

<div class="order" order-state="pendiente">
  Un pedido pendiente.
</div>

```

```
<div class="order" order-state="cancelado">
  Un pedido cancelado
</div>
```

¿Por qué sería preferible usar un atributo a tener clases como `.order-state-new`, `.order-state-pending`, `.order-state-cancelado`?

Porque un atributo es más conveniente de administrar. El estado se puede cambiar tan fácil como:

```
// un poco más simple que eliminar antiguos / agregar una nueva clase
div.setAttribute('order-state', 'canceled');
```

Pero puede haber un posible problema con los atributos personalizados. ¿Qué sucede si usamos un atributo no estándar para nuestros propósitos y luego el estándar lo introduce y hace que haga algo? El lenguaje HTML está vivo, crece y cada vez hay más atributos que aparecen para satisfacer las necesidades de los desarrolladores. Puede haber efectos inesperados en tal caso.

Para evitar conflictos, existen atributos `data-*` [↗](#).

Todos los atributos que comienzan con “data-” están reservados para el uso de los programadores. Están disponibles en la propiedad `dataset`.

Por ejemplo, si un `elem` tiene un atributo llamado `"data-about"`, está disponible como `elem.dataset.about`.

Como esto:

```
<body data-about="Elefante">
<script>
  alert(document.body.dataset.about); // Elefante
</script>
```

Los atributos de varias palabras como `data-order-state` se convierten en camel-case: `dataset.orderState`

Aquí hay un ejemplo reescrito de “estado del pedido”:

```
<style>
  .order[data-order-state="nuevo"] {
    color: green;
  }

  .order[data-order-state="pendiente"] {
    color: blue;
  }

  .order[data-order-state="cancelado"] {
    color: red;
  }
</style>

<div id="order" class="order" data-order-state="nuevo">
  Una nueva orden.
</div>

<script>
  // leer
  alert(order.dataset.orderState); // nuevo

  // modificar
  order.dataset.orderState = "pendiente"; // (*)
</script>
```

El uso de los atributos `data-*` es una forma válida y segura de pasar datos personalizados.

Tenga en cuenta que no solo podemos leer, sino también modificar los atributos de datos. Luego, CSS actualiza la vista en consecuencia: en el ejemplo anterior, la última línea `(*)` cambia el color a azul.

Resumen

- Atributos: es lo que está escrito en HTML.
- Propiedades: es lo que hay en los objetos DOM.

Una pequeña comparación:

	Propiedades	Atributos
Tipo	Cualquier valor, las propiedades estándar tienen tipos descritos en la especificación	Un string
Nombre	El nombre distingue entre mayúsculas y minúsculas	El nombre no distingue entre mayúsculas y minúsculas

Los métodos para trabajar con atributos son:

- `elem.hasAttribute(nombre)` – para comprobar si existe.
- `elem.getAttribute(nombre)` – para obtener el valor.
- `elem.setAttribute(nombre, valor)` – para dar un valor.
- `elem.removeAttribute(nombre)` – para eliminar el atributo.
- `elem.attributes` es una colección de todos los atributos.

Para la mayoría de las situaciones, es preferible usar las propiedades DOM. Deberíamos referirnos a los atributos solo cuando las propiedades DOM no nos convienen, cuando necesitamos exactamente atributos, por ejemplo:

- Necesitamos un atributo no estándar. Pero si comienza con `data-`, entonces deberíamos usar `dataset`.
- Queremos leer el valor “como está escrito” en HTML. El valor de la propiedad DOM puede ser diferente, por ejemplo, la propiedad `href` siempre es una URL completa, y es posible que queramos obtener el valor “original”.

✓ Tareas

Obtén en atributo

importancia: 5

Escribe el código para obtener el atributo `data-widget-name` del documento y leer su valor.

```
<!DOCTYPE html>
<html>
<body>

  <div data-widget-name="menu">Elige el genero</div>

  <script>
    /* Tu código */
  </script>
</body>
</html>
```

A solución

Haz los enlaces externos naranjas

importancia: 3

Haz todos los enlaces externos de color orange alterando su propiedad `style`.

Un link es externo si:

- Su `href` tiene `://`
- Pero no comienza con `http://internal.com`.

Ejemplo:


```

<a name="list">the list</a>
<ul>
  <li><a href="http://google.com">http://google.com</a></li>
  <li><a href="/tutorial">/tutorial.html</a></li>
  <li><a href="local/path">local/path</a></li>
  <li><a href="ftp://ftp.com/my.zip">ftp://ftp.com/my.zip</a></li>
  <li><a href="http://nodejs.org">http://nodejs.org</a></li>
  <li><a href="http://internal.com/test">http://internal.com/test</a></li>
</ul>

<script>
  // establecer un estilo para un enlace
  let link = document.querySelector('a');
  link.style.color = 'orange';
</script>

```

El resultado podría ser:

La lista:

- <http://google.com>
- </tutorial.html>
- <local/path>
- <ftp://ftp.com/my.zip>
- <http://nodejs.org>
- <http://internal.com/test>

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Modifying the document

DOM modification is the key to creating “live” pages.

Here we’ll see how to create new elements “on the fly” and modify the existing page content.

Example: show a message

Let’s demonstrate using an example. We’ll add a message on the page that looks nicer than `alert`.

Here’s how it will look:

```

<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert">
  <strong>Hi there!</strong> You've read an important message.
</div>

```

Hi there! You've read an important message.

That was the HTML example. Now let’s create the same `div` with JavaScript (assuming that the styles are in the HTML/CSS already).

Creating an element

To create DOM nodes, there are two methods:

`document.createElement(tag)`

Creates a new *element node* with the given tag:

```
let div = document.createElement('div');
```

`document.createTextNode(text)`

Creates a new *text node* with the given text:

```
let textNode = document.createTextNode('Here I am');
```

Most of the time we need to create element nodes, such as the `div` for the message.

Creating the message

Creating the message div takes 3 steps:

```
// 1. Create <div> element
let div = document.createElement('div');

// 2. Set its class to "alert"
div.className = "alert";

// 3. Fill it with the content
div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";
```

We've created the element. But as of now it's only in a variable named `div`, not in the page yet. So we can't see it.

Insertion methods

To make the `div` show up, we need to insert it somewhere into `document`. For instance, into `<body>` element, referenced by `document.body`.

There's a special method `append` for that: `document.body.append(div)`.

Here's the full code:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

  document.body.append(div);
</script>
```

Here we called `append` on `document.body`, but we can call `append` method on any other element, to put another element into it. For instance, we can append something to `<div>` by calling `div.append(anotherElement)`.

Here are more insertion methods, they specify different places where to insert:

- `node.append(...nodes or strings)` – append nodes or strings *at the end* of `node`,
- `node.prepend(...nodes or strings)` – insert nodes or strings *at the beginning* of `node`,
- `node.before(...nodes or strings)` – insert nodes or strings *before* `node`,
- `node.after(...nodes or strings)` – insert nodes or strings *after* `node`,
- `node.replaceWith(...nodes or strings)` – replaces `node` with the given nodes or strings.

Arguments of these methods are an arbitrary list of DOM nodes to insert, or text strings (that become text nodes automatically).

Let's see them in action.

Here's an example of using these methods to add items to a list and the text before/after it:

```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before'); // insert string "before" before <ol>
  ol.after('after'); // insert string "after" after <ol>

  let liFirst = document.createElement('li');
  liFirst.innerHTML = 'prepend';
  ol.prepend(liFirst); // insert liFirst at the beginning of <ol>

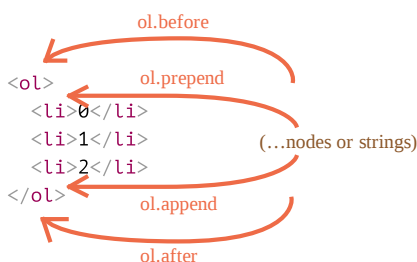
  let liLast = document.createElement('li');
  liLast.innerHTML = 'append';
  ol.append(liLast); // insert liLast at the end of <ol>
</script>
```

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

Here's a visual picture of what the methods do:



So the final list will be:

```
before
<ol id="ol">
  <li>prepend</li>
  <li>0</li>
  <li>1</li>
  <li>2</li>
  <li>append</li>
</ol>
after
```

As said, these methods can insert multiple nodes and text pieces in a single call.

For instance, here a string and an element are inserted:

```
<div id="div"></div>
<script>
  div.before('<p>Hello</p>', document.createElement('hr'));
</script>
```

Please note: the text is inserted “as text”, not “as HTML”, with proper escaping of characters such as `<`, `>`.

So the final HTML is:

```
&lt;p&gt;Hello&lt;/p&gt;
<hr>
<div id="div"></div>
```

In other words, strings are inserted in a safe way, like `elem.textContent` does it.

So, these methods can only be used to insert DOM nodes or text pieces.

But what if we'd like to insert an HTML string “as html”, with all tags and stuff working, in the same manner as `elem.innerHTML` does it?

insertAdjacentHTML/Text/Element

For that we can use another, pretty versatile method: `elem.insertAdjacentHTML(where, html)`.

The first parameter is a code word, specifying where to insert relative to `elem`. Must be one of the following:

- "beforebegin" – insert `html` immediately before `elem`,
- "afterbegin" – insert `html` into `elem`, at the beginning,
- "beforeend" – insert `html` into `elem`, at the end,
- "afterend" – insert `html` immediately after `elem`.

The second parameter is an HTML string, that is inserted “as HTML”.

For instance:

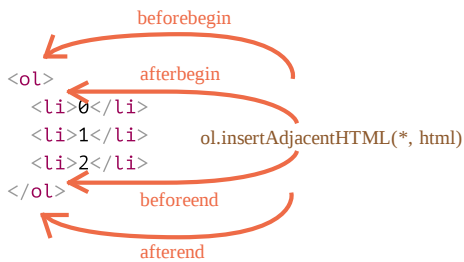
```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Hello</p>');
  div.insertAdjacentHTML('afterend', '<p>Bye</p>');
</script>
```

...Would lead to:

```
<p>Hello</p>
<div id="div"></div>
<p>Bye</p>
```

That's how we can append arbitrary HTML to the page.

Here's the picture of insertion variants:



We can easily notice similarities between this and the previous picture. The insertion points are actually the same, but this method inserts HTML.

The method has two brothers:

- `elem.insertAdjacentText(where, text)` – the same syntax, but a string of `text` is inserted “as text” instead of HTML,
- `elem.insertAdjacentElement(where, elem)` – the same syntax, but inserts an element.

They exist mainly to make the syntax “uniform”. In practice, only `insertAdjacentHTML` is used most of the time. Because for elements and text, we have methods `append/prepend/before/after` – they are shorter to write and can insert nodes/text pieces.

So here's an alternative variant of showing a message:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
    <strong>Hi there!</strong> You've read an important message.
  </div>`);
</script>
```

Node removal

To remove a node, there's a method `node.remove()`.

Let's make our message disappear after a second:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

  document.body.append(div);
  setTimeout(() => div.remove(), 1000);
</script>
```

Please note: if we want to *move* an element to another place – there's no need to remove it from the old one.

All insertion methods automatically remove the node from the old place.

For instance, let's swap elements:

```
<div id="first">First</div>
<div id="second">Second</div>
<script>
  // no need to call remove
  second.after(first); // take #second and after it insert #first
</script>
```

Cloning nodes: cloneNode

How to insert one more similar message?

We could make a function and put the code there. But the alternative way would be to *clone* the existing `div` and modify the text inside it (if needed).

Sometimes when we have a big element, that may be faster and simpler.

- The call `elem.cloneNode(true)` creates a “deep” clone of the element – with all attributes and subelements. If we call `elem.cloneNode(false)`, then the clone is made without child elements.

An example of copying the message:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert" id="div">
  <strong>Hi there!</strong> You've read an important message.
</div>

<script>
  let div2 = div.cloneNode(true); // clone the message
  div2.querySelector('strong').innerHTML = 'Bye there!'; // change the clone

  div.after(div2); // show the clone after the existing div
</script>
```

DocumentFragment

`DocumentFragment` is a special DOM node that serves as a wrapper to pass around lists of nodes.

We can append other nodes to it, but when we insert it somewhere, then its content is inserted instead.

For example, `getListContent` below generates a fragment with `` items, that are later inserted into ``:

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }
}
```

```

    return fragment;
}

ul.append(getListContent()); // (*)
</script>

```

Please note, at the last line `(*)` we append `DocumentFragment`, but it “blends in”, so the resulting structure will be:

```

<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>

```

`DocumentFragment` is rarely used explicitly. Why append to a special kind of node, if we can return an array of nodes instead? Rewritten example:

```

<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append + "..." operator = friends!
</script>

```

We mention `DocumentFragment` mainly because there are some concepts on top of it, like [template](#) element, that we’ll cover much later.

Old-school insert/remove methods

⚠ Vieja escuela

Esta información ayuda a entender los viejos scripts, pero no es necesaria para nuevos desarrollos.

There are also “old school” DOM manipulation methods, existing for historical reasons.

These methods come from really ancient times. Nowadays, there’s no reason to use them, as modern methods, such as `append`, `prepend`, `before`, `after`, `remove`, `replaceWith`, are more flexible.

The only reason we list these methods here is that you can find them in many old scripts:

`parentElem.appendChild(node)`

Appends `node` as the last child of `parentElem`.

The following example adds a new `` to the end of ``:

```

<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

```

```
<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';

  list.appendChild(newLi);
</script>
```

parentElem.insertBefore(node, nextSibling)

Inserts `node` before `nextSibling` into `parentElem`.

The following code inserts a new list item before the second ``:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';

  list.insertBefore(newLi, list.children[1]);
</script>
```

To insert `newLi` as the first element, we can do it like this:

```
list.insertBefore(newLi, list.firstChild);
```

parentElem.replaceChild(node, oldChild)

Replaces `oldChild` with `node` among children of `parentElem`.

parentElem.removeChild(node)

Removes `node` from `parentElem` (assuming `node` is its child).

The following example removes first `` from ``:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  let li = list.firstChild;
  list.removeChild(li);
</script>
```

All these methods return the inserted/removed node. In other words, `parentElem.appendChild(node)` returns `node`. But usually the returned value is not used, we just run the method.

A word about “document.write”

There's one more, very ancient method of adding something to a web-page: `document.write`.

The syntax:

```
<p>Somewhere in the page...</p>
<script>
  document.write('<b>Hello from JS</b>');
</script>
<p>The end</p>
```


The call to `document.write(html)` writes the `html` into page “right here and now”. The `html` string can be dynamically generated, so it's kind of flexible. We can use JavaScript to create a full-fledged webpage and write it.

The method comes from times when there was no DOM, no standards... Really old times. It still lives, because there are scripts using it.

In modern scripts we can rarely see it, because of the following important limitation:

The call to `document.write` only works while the page is loading.

If we call it afterwards, the existing document content is erased.

For instance:

```
<p>After one second the contents of this page will be replaced...</p>
<script>
  // document.write after 1 second
  // that's after the page loaded, so it erases the existing content
  setTimeout(() => document.write('<b>...By this.</b>'), 1000);
</script>
```

So it's kind of unusable at “after loaded” stage, unlike other DOM methods we covered above.

That's the downside.

There's an upside also. Technically, when `document.write` is called while the browser is reading (“parsing”) incoming HTML, and it writes something, the browser consumes it just as if it were initially there, in the HTML text.

So it works blazingly fast, because there's *no DOM modification* involved. It writes directly into the page text, while the DOM is not yet built.

So if we need to add a lot of text into HTML dynamically, and we're at page loading phase, and the speed matters, it may help. But in practice these requirements rarely come together. And usually we can see this method in scripts just because they are old.

Summary

- Methods to create new nodes:
 - `document.createElement(tag)` – creates an element with the given tag,
 - `document.createTextNode(value)` – creates a text node (rarely used),
 - `elem.cloneNode(deep)` – clones the element, if `deep==true` then with all descendants.
- Insertion and removal:
 - `node.append(...nodes or strings)` – insert into `node`, at the end,
 - `node.prepend(...nodes or strings)` – insert into `node`, at the beginning,
 - `node.before(...nodes or strings)` – insert right before `node`,
 - `node.after(...nodes or strings)` – insert right after `node`,
 - `node.replaceWith(...nodes or strings)` – replace `node`.
 - `node.remove()` – remove the `node`.

Text strings are inserted “as text”.

- There are also “old school” methods:
 - `parent.appendChild(node)`
 - `parent.insertBefore(node, nextSibling)`
 - `parent.removeChild(node)`
 - `parent.replaceChild(newElem, node)`

All these methods return `node`.

- Given some HTML in `html`, `elem.insertAdjacentHTML(where, html)` inserts it depending on the value of `where`:

- `"beforebegin"` – insert html right before `elem`,
- `"afterbegin"` – insert html into `elem`, at the beginning,
- `"beforeend"` – insert html into `elem`, at the end,
- `"afterend"` – insert html right after `elem`.

Also there are similar methods, `elem.insertAdjacentText` and `elem.insertAdjacentElement`, that insert text strings and elements, but they are rarely used.

- To append HTML to the page before it has finished loading:
 - `document.write(html)`

After the page is loaded such a call erases the document. Mostly seen in old scripts.

✓ Tareas

createTextNode vs innerHTML vs textContent

importancia: 5

We have an empty DOM element `elem` and a string `text`.

Which of these 3 commands will do exactly the same?

1. `elem.append(document.createTextNode(text))`
2. `elem.innerHTML = text`
3. `elem.textContent = text`

[A solución](#)

Clear the element

importancia: 5

Create a function `clear(elem)` that removes everything from the element.

```
<ol id="elem">
  <li>Hello</li>
  <li>World</li>
</ol>

<script>
  function clear(elem) { /* your code */ }

  clear(elem); // clears the list
</script>
```

[A solución](#)

Why does "aaa" remain?

importancia: 1

In the example below, the call `table.remove()` removes the table from the document.

But if you run it, you can see that the text `"aaa"` is still visible.

Why does that happen?

```
<table id="table">
  aaa
  <tr>
    <td>Test</td>
  </tr>
```

```
</table>

<script>
  alert(table); // the table, as it should be

  table.remove();
  // why there's still aaa in the document?
</script>
```

[A solución](#)

Create a list

importancia: 4

Write an interface to create a list from user input.

For every list item:

1. Ask a user about its content using `prompt` .
2. Create the `` with it and add it to `` .
3. Continue until the user cancels the input (by pressing `Esc` or via an empty entry).

All elements should be created dynamically.

If a user types HTML-tags, they should be treated like a text.

[Demo en nueva ventana](#) ↗

[A solución](#)

Create a tree from the object

importancia: 5

Write a function `createTree` that creates a nested `ul/li` list from the nested object.

For instance:

```
let data = {
  "Fish": {
    "trout": {},
    "salmon": {}
  },

  "Tree": {
    "Huge": {
      "sequoia": {},
      "oak": {}
    },
    "Flowering": {
      "apple tree": {},
      "magnolia": {}
    }
  }
};
```

The syntax:

```
let container = document.getElementById('container');
createTree(container, data); // creates the tree in the container
```

The result (tree) should look like this:

- Fish
 - trout
 - salmon
- Tree
 - Huge
 - sequoia
 - oak
 - Flowering
 - apple tree
 - magnolia

Choose one of two ways of solving this task:

1. Create the HTML for the tree and then assign to `container.innerHTML`.
2. Create tree nodes and append with DOM methods.

Would be great if you could do both.

P.S. The tree should not have “extra” elements like empty `` for the leaves.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Show descendants in a tree

importancia: 5

There's a tree organized as nested `ul/li`.

Write the code that adds to each `` the number of its descendants. Skip leaves (nodes without children).

The result:

- Animals [9]
 - Mammals [4]
 - Cows
 - Donkeys
 - Dogs
 - Tigers
 - Other [3]
 - Snakes
 - Birds
 - Lizards
- Fishes [5]
 - Aquarium [2]
 - Guppy
 - Angelfish
 - Sea [1]
 - Sea trout

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Create a calendar

importancia: 4

Write a function `createCalendar(elem, year, month)`.

The call should create a calendar for the given year/month and put it inside `elem`.

The calendar should be a table, where a week is `<tr>` , and a day is `<td>` . The table top should be `<th>` with weekday names: the first day should be Monday, and so on till Sunday.

For instance, `createCalendar(cal, 2012, 9)` should generate in element `cal` the following calendar:

MO	TU	WE	TH	FR	SA	SU
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

P.S. For this task it's enough to generate the calendar, should not yet be clickable.

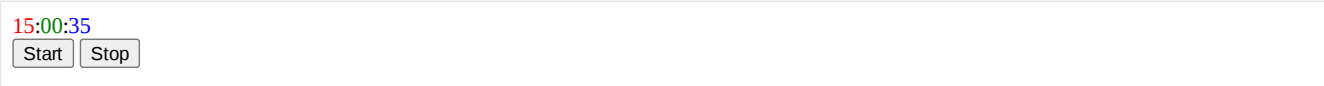
[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Colored clock with setInterval

importancia: 4

Create a colored clock like here:



Use HTML/CSS for the styling, JavaScript only updates time in elements.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Insert the HTML in the list

importancia: 5

Write the code to insert `23` between two `` here:

```
<ul id="ul">
  <li id="one">1</li>
  <li id="two">4</li>
</ul>
```

[A solución](#)

Sort the table

importancia: 5

There's a table:

```
<table>
<thead>
  <tr>
    <th>Name</th><th>Surname</th><th>Age</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>John</td><td>Smith</td><td>10</td>
```

```

</tr>
<tr>
  <td>Pete</td><td>Brown</td><td>15</td>
</tr>
<tr>
  <td>Ann</td><td>Lee</td><td>5</td>
</tr>
<tr>
  <td>...</td><td>...</td><td>...</td>
</tr>
</tbody>
</table>

```

There may be more rows in it.

Write the code to sort it by the "name" column.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Estilos y clases

Antes de profundizar en cómo JavaScript maneja las clases y los estilos, hay una regla importante. Aunque es lo suficientemente obvio, aún tenemos que mencionarlo.

Por lo general, hay dos formas de dar estilo a un elemento:

1. Crear una clase `css` y agregarla: `<div class="...">`
2. Escribir las propiedades directamente en `style`: `<div style="...">`.

JavaScript puede modificar ambos, clases y las propiedades de `style`.

Nosotros deberíamos preferir las clases `css` en lugar de `style`. Este último solo debe usarse si las clases “no pueden manejarlo”.

Por ejemplo, `style` es aceptable si nosotros calculamos las coordenadas de un elemento dinámicamente y queremos establecer estas desde JavaScript, así:

```

let top = /* cálculos complejos */;
let left = /* cálculos complejos */;

elem.style.left = left; // ej. '123px', calculado en tiempo de ejecución
elem.style.top = top; // ej. '456px'

```

Para otros casos como convertir un texto en rojo, agregar un icono de fondo. Escribir eso en CSS y luego agregar la clase (JavaScript puede hacer eso), es más flexible y más fácil de mantener.

className y classList

Cambiar una clase es una de las acciones más utilizadas.

En la antigüedad, había una limitación en JavaScript: una palabra reservada como `"class"` no podía ser una propiedad de un objeto. Esa limitación no existe ahora, pero en ese momento era imposible tener una propiedad `"class"`, como `elem.class`.

Entonces para clases de similares propiedades, `"className"` fue introducido: el `elem.className` corresponde al atributo `"class"`.

Por ejemplo:

```

<body class="main page">
  <script>

```

```
    alert(document.body.className); // página principal
  </script>
</body>
```

Si asignamos algo a `elem.className`, reemplaza toda la cadena de clases. A veces es lo que necesitamos, pero a menudo queremos agregar o eliminar una sola clase.

Hay otra propiedad para eso: `elem.classList`.

El `elem.classList` es un objeto especial con métodos para agregar, eliminar y alternar (`add/remove/toggle`) una sola clase.

Por ejemplo:

```
<body class="main page">
  <script>
    // agregar una clase
    document.body.classList.add('article');

    alert(document.body.className); // clase "article" de la página principal
  </script>
</body>
```

Entonces podemos trabajar con ambos: todas las clases como una cadena usando `className` o con clases individuales usando `classList`. Lo que elijamos depende de nuestras necesidades.

Métodos de `classList`:

- `elem.classList.add/remove("class")` – agrega o remueve la clase.
- `elem.classList.toggle("class")` – agrega la clase si no existe, si no la remueve.
- `elem.classList.contains("class")` – verifica si tiene la clase dada, devuelve `true/false`.

Además, `classList` es iterable, entonces podemos listar todas las clases con `for...of`, así:

```
<body class="main page">
  <script>
    for (let name of document.body.classList) {
      alert(name); // main y luego page
    }
  </script>
</body>
```

style de un elemento

La propiedad `elem.style` es un objeto que corresponde a lo escrito en el atributo `"style"`. Establecer `elem.style.width="100px"` funciona igual que si tuviéramos en el atributo `style` una cadena con `width:100px`.

Para propiedades de varias palabras se usa `camelCase`:

```
background-color => elem.style.backgroundColor
z-index          => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Por ejemplo:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

i Propiedades prefijadas

Propiedades con prefijos del navegador como `-moz-border-radius`, `-webkit-border-radius` también siguen la misma regla: un guion significa mayúscula.

Por ejemplo:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Reseteando la propiedad `style`

A veces queremos asignar una propiedad de estilo y luego removerla.

Por ejemplo, para ocultar un elemento, podemos establecer `elem.style.display = "none"`.

Luego, más tarde, es posible que queramos remover `style.display` como si no estuviera establecido. En lugar de `delete elem.style.display` deberíamos asignarle una cadena vacía: `elem.style.display = ""`.

```
// si ejecutamos este código, el <body> parpadeará
document.body.style.display = "none"; // ocultar

setTimeout(() => document.body.style.display = "", 1000); // volverá a lo normal
```

Si establecemos `style.display` como una cadena vacía, entonces el navegador aplica clases y estilos CSS incorporados normalmente por el navegador, como si no existiera tal `style.display`.

i Reescribir todo usando `style.cssText`

Normalmente, podemos usar `style.*` para asignar propiedades de estilo individuales. No podemos establecer todo el estilo como `div.style="color: red; width: 100px"`, porque `div.style` es un objeto y es solo de lectura.

Para establecer todo el estilo como una cadena, hay una propiedad especial: `style.cssText`:

```
<div id="div">Button</div>

<script>
  // podemos establecer estilos especiales con banderas como "important"
  div.style.cssText=`color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center;
  `;

  alert(div.style.cssText);
</script>
```

Esta propiedad es rara vez usada, porque tal asignación remueve todo los estilos: no agrega estilos sino que los reemplaza en su totalidad. Ocasionalmente podría eliminar algo necesario. Pero podemos usarlo de manera segura para nuevos elementos, cuando sabemos que no vamos a eliminar un estilo existente.

Lo mismo se puede lograr estableciendo un atributo: `div.setAttribute('style', 'color: red...')`.

Cuidado con las unidades CSS

No olvidar agregar las unidades CSS a los valores.

Por ejemplo, nosotros no debemos establecer `elem.style.top` a `10`, sino más bien a `10px`. De lo contrario no funcionaría:


```

<body>
  <script>
    // ¡no funciona!
    document.body.style.margin = 20;
    alert(document.body.style.margin); // '' (cadena vacía, la asignación es ignorada)

    // ahora agregamos la unidad CSS (px) y esta sí funciona
    document.body.style.margin = '20px';
    alert(document.body.style.margin); // 20px

    alert(document.body.style.marginTop); // 20px
    alert(document.body.style.marginLeft); // 20px
  </script>
</body>

```

Tenga en cuenta: el navegador “desempaqueta” la propiedad `style.margin` en las últimas líneas e infiere `style.marginLeft` y `style.marginTop` de eso.

Estilos calculados: `getComputedStyle`

Entonces, modificar un estilo es fácil. ¿Pero cómo *leerlo*?

Por ejemplo, queremos saber el tamaño, los márgenes, el color de un elemento. ¿Cómo hacerlo?

La propiedad `style` solo opera en el valor del atributo `"style"`, sin ninguna cascada de `css`.

Entonces no podemos leer ninguna clase CSS usando `elem.style`.

Por ejemplo, aquí `style` no ve el margen:

```

<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  El texto en rojo
  <script>
    alert(document.body.style.color); // vacío
    alert(document.body.style.marginTop); // vacío
  </script>
</body>

```

Pero si necesitamos incrementar el margen a `20px` ? vamos a querer el valor de la misma.

Hay otro método para eso: `getComputedStyle`.

La sintaxis es:

```
getComputedStyle(element, [pseudo])
```

element

Elemento del cual se va a leer el valor.

pseudo

Un pseudo-elemento es requerido, por ejemplo `::before`. Una cadena vacía o sin argumento significa el elemento mismo.

El resultado es un objeto con estilos, como `elem.style`, pero ahora con respecto a todas las clases CSS.

Por ejemplo:

```

<head>
  <style> body { color: red; margin: 5px } </style>
</head>

```

```

<body>

<script>
  let computedStyle = getComputedStyle(document.body);

  // ahora podemos leer los márgenes y el color de ahí

  alert( computedStyle.marginTop ); // 5px
  alert( computedStyle.color ); // rgb(255, 0, 0)
</script>

</body>

```

i Valores calculado y resueltos

Hay dos conceptos en [CSS](#) ↗ :

1. Un estilo *calculado* es el valor final de aplicar todas las reglas y herencias CSS, como resultado de la cascada CSS. Puede parecer `height:1em` o `font-size:125%`.
2. Un estilo *resuelto* es la que finalmente se aplica al elemento. Valores como `1em` o `125%` son relativos. El navegador toma el valor calculado y hace que todas las unidades sean fijas y absolutas, por ejemplo: `height:20px` o `font-size:16px`. Para las propiedades de geometría los valores resueltos pueden tener un punto flotante, como `width:50.5px`.

Hace mucho tiempo `getComputedStyle` fue creado para obtener los valores calculados, pero los valores resueltos son muchos más convenientes, y el estándar cambió.

Así que hoy en día `getComputedStyle` en realidad devuelve el valor resuelto de la propiedad, usualmente en `px` para geometría.

⚠ El método `getComputedStyle` requiere el nombre completo de la propiedad

Siempre deberíamos preguntar por la propiedad exacta que queremos, como `paddingLeft` o `marginTop` o `borderTopWidth`. De lo contrario, no se garantiza el resultado correcto.

Por ejemplo, si hay propiedades `paddingLeft/paddingTop`, ¿entonces qué deberíamos obtener de `getComputedStyle(elem).padding`? ¿Nada, o tal vez un valor "generado" de los paddings? No hay una regla estándar aquí.

Hay otras inconsistencias. Por ejemplo, algunos navegadores (Chrome) muestran `10px` en el documento a continuación, y alguno de ellos (Firefox) no:

```

<style>
  body {
    margin: 10px;
  }
</style>
<script>
  let style = getComputedStyle(document.body);
  alert(style.margin); // cadena vacía en Firefox
</script>

```

i ¡Los estilos aplicados a los enlaces `:visited` están ocultos!

Los enlaces visitados pueden ser coloreados usando la pseudo-clase `:visited` de CSS.

Pero `getComputedStyle` no da acceso a ese color, porque de lo contrario una página cualquiera podría averiguar si el usuario visitó un enlace creándolo en la página y verificar los estilos.

JavaScript no puede ver los estilos aplicados por `:visited`. También hay una limitación en CSS que prohíbe la aplicación de estilos de cambio de geometría en `:visited`. Eso es para garantizar que no haya forma para que una página maligna pruebe si un enlace fue visitado y vulnere la privacidad.

Resumen

Para manejar clases, hay dos propiedades del DOM:

- `className` – el valor de la cadena, perfecto para manejar todo el conjunto de clases.
- `classList` – el objeto con los métodos: `add/remove/toggle/contains`, perfecto para clases individuales.

Para cambiar los estilos:

- La propiedad `style` es un objeto con los estilos en `camelcase`. Leer y escribir tiene el mismo significado que modificar propiedades individuales en el atributo `"style"`. Para ver cómo aplicar `important` y otras cosas raras, hay una lista de métodos en [MDN](#).
- La propiedad `style.cssText` corresponde a todo el atributo `"style"`, la cadena completa de estilos.

Para leer los estilos resueltos (con respecto a todas las clases, después de que se aplica todo el `css` y se calculan los valores finales):

- El método `getComputedStyle(elem, [pseudo])` retorna el objeto de estilo con ellos (solo lectura).

✓ Tareas

Crear una notificación

importancia: 5

Escribir una función `showNotification(options)` que cree una notificación: `<div class="notification">` con el contenido dado. La notificación debería desaparecer automáticamente después de 1.5 segundos.

Las opciones son:

```
// muestra un elemento con el texto "Hello" cerca de la parte superior de la ventana
showNotification({
  top: 10, // 10px desde la parte superior de la ventana (por defecto es 0px)
  right: 10, // 10px desde el borde derecho de la ventana (por defecto es 0px)
  html: "Hello!", // el HTML de la notificación
  className: "welcome" // una clase adicional para el "div" (opcional)
});
```

[Demo en nueva ventana](#)

Usar posicionamiento CSS para mostrar el elemento en las coordenadas (top/right) dadas. El documento tiene los estilos necesarios.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Elementos de tamaño y desplazamiento

Estas son muchas de las propiedades de JavaScript que nos permiten leer información de los elementos sobre el ancho, alto y otras características geométricas.

Nosotros a menudo necesitamos de ellas cuando movemos o posicionamos un elemento en JavaScript.

Elemento de muestra

Como un elemento de muestra para demostrar las propiedades, usaremos el que se indica a continuación:

```
<div id="example">
  ...Texto...
```

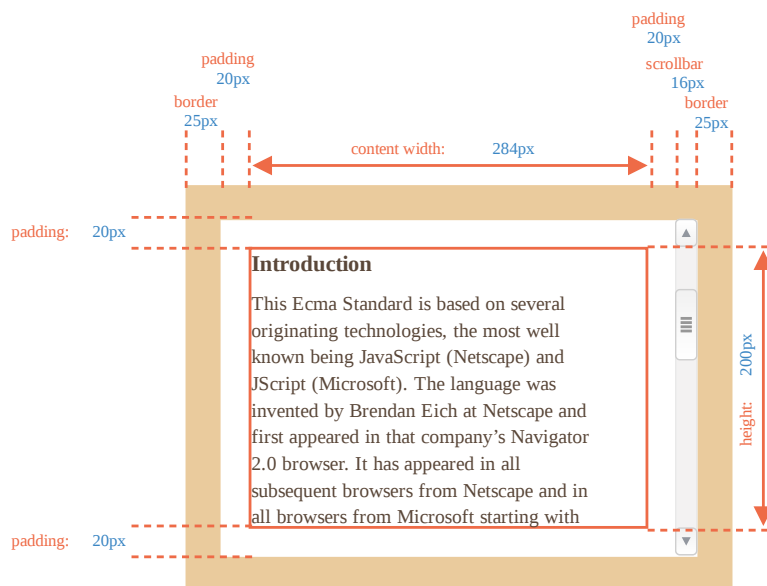
```

</div>
<style>
  #example {
    width: 300px;
    height: 200px;
    border: 25px solid #E8C48F;
    padding: 20px;
    overflow: auto;
  }
</style>

```

Este tiene el borde, relleno y desplazamiento. El conjunto completo de funciones. No hay márgenes, ya que no son parte del elemento en sí, y estos no tiene propiedades especiales.

El elemento tiene este aspecto:



Tu puedes [abrir el documento en la zona de pruebas](#) .

i Mind the scrollbar

La imagen de arriba muestra el caso más complejo cuando el elemento tiene una barra de desplazamiento. Algunos navegadores (no todos) reservan espacio para tomarlo del contenido (etiquetado como "content width" arriba).

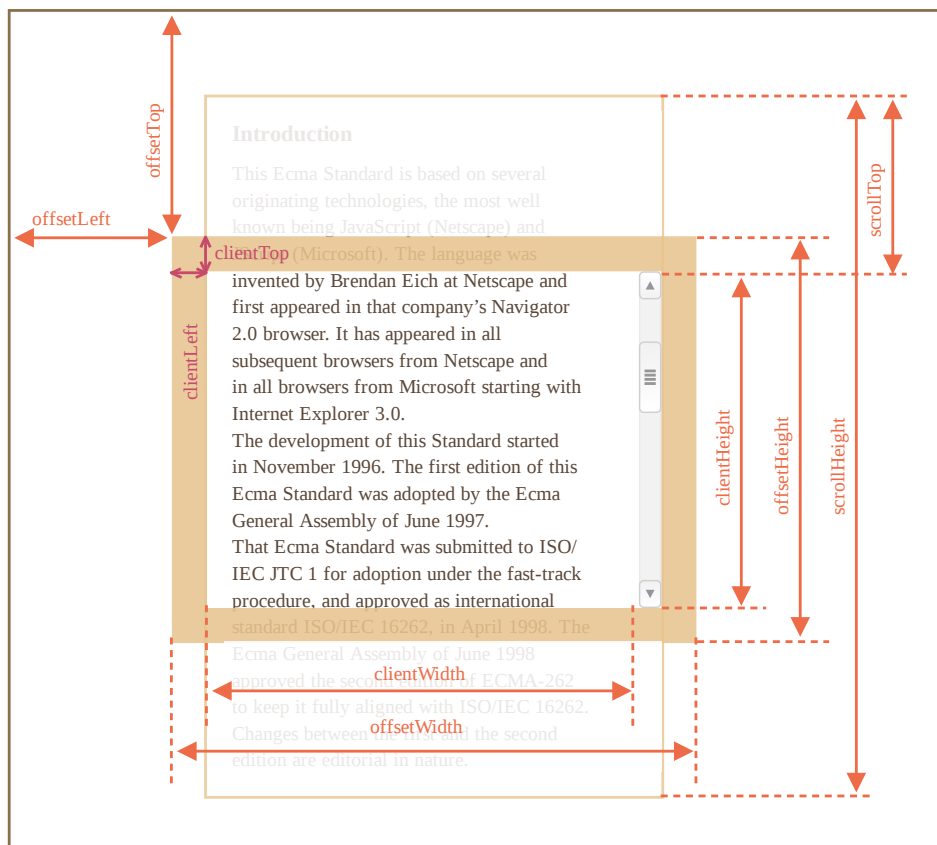
Entonces, sin la barra de desplazamiento el ancho del contenido sería 300px , pero si la barra de desplazamiento tiene un ancho de 16px (el ancho puede variar entre dispositivos y navegadores) entonces solo queda $300 - 16 = 284\text{px}$, y deberíamos tenerlo en cuenta. Es por eso que los ejemplos de este capítulo asumen que hay una barra de desplazamiento. Sin él, algunos cálculos son más sencillos.

i El área padding-bottom puede estar lleno de texto

Por lo general, los rellenos se muestran vacíos en nuestras ilustraciones, pero si hay mucho texto en el elemento y se desborda, los navegadores muestran el texto "desbordado" en padding-bottom , eso es normal.

Geometría

Aquí está la imagen general con propiedades geométricas:



Los valores de estas propiedades son técnicamente números, pero estos números son “de píxeles”, así que estas son medidas de píxeles.

Comencemos a explotar las propiedades, iniciando desde el interior del elemento.

offsetParent, offsetLeft/Top

Estas propiedades son raramente necesarias, pero aún son las propiedades de geometría “más externas”, así que comenzaremos con ellas.

El `offsetParent` es el antepasado más cercano que usa el navegador para calcular las coordenadas durante el renderizado.

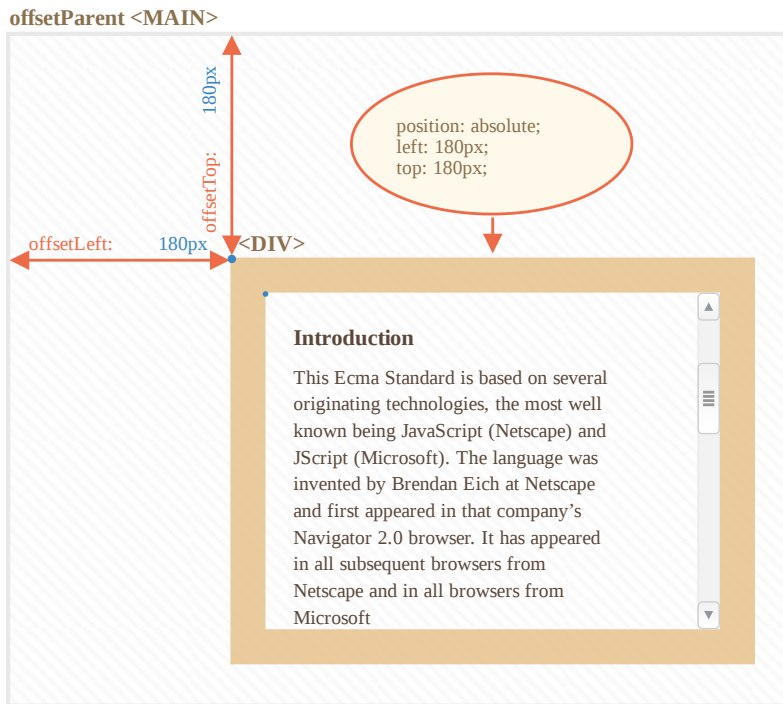
Ese es el antepasado más cercano que es uno de los siguientes:

1. CSS-positioned (`position` is `absolute`, `relative`, `fixed` or `sticky`), or
2. `<td>`, `<th>`, or `<table>`, or
3. `<body>`.

Las propiedades `offsetLeft/offsetTop` proporcionan coordenadas x/y relativas a la esquina superior izquierda de `offsetParent`.

En el siguiente ejemplo el `<div>` interno tiene `<main>` como `offsetParent` y `offsetLeft/offsetTop` se desplaza desde su esquina superior izquierda (180):

```
<main style="position: relative" id="main">
  <article>
    <div id="example" style="position: absolute; left: 180px; top: 180px">...</div>
  </article>
</main>
<script>
  alert(example.offsetParent.id); // main
  alert(example.offsetLeft); // 180 (note: a number, not a string "180px")
  alert(example.offsetTop); // 180
</script>
```



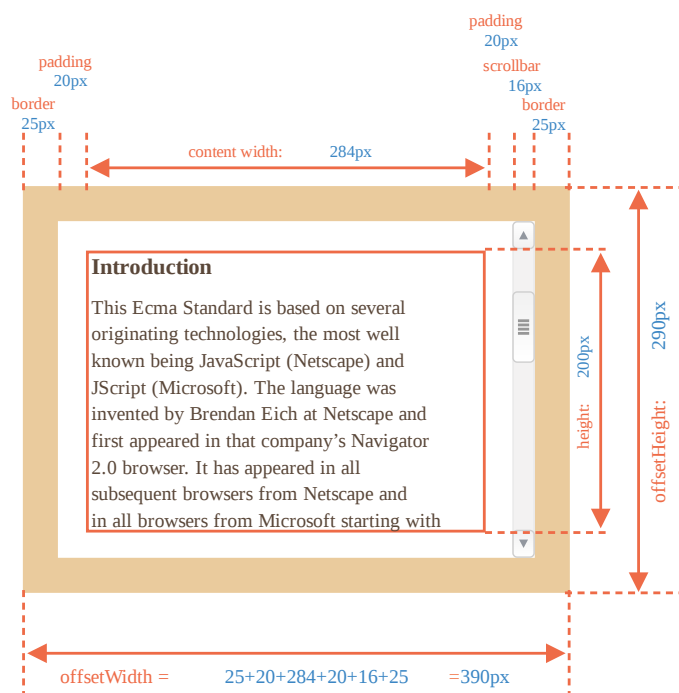
Hay varias ocasiones en la que `offsetParent` es `null`:

1. Para elementos no mostrados (`display:none` o no en el documento).
2. Para `<body>` y `<html>`.
3. Para elementos con `position:fixed`.

offsetWidth/Height

Ahora pasemos al elemento en sí.

Estas dos propiedades son las más simples. Proporcionan el ancho y alto "exterior" del elemento. O, en otras palabras, su tamaño completo, incluidos los bordes.



Para nuestro elemento de muestra:

- `offsetWidth = 390` – el ancho exterior, puede ser calculado como CSS-width interno (300px) más acolchonados (2 * 20px) y bordes (2 * 25px).

- `offsetHeight = 290` – el alto exterior.

i Las propiedades geométricas para elementos no mostrados son cero o null

Las propiedades geométricas son calculadas solo para elementos mostrados.

En un elemento (o cualquiera de sus antepasados) tiene `display:none` o no está en el documento, entonces las propiedades geométricas son cero (o `null` para `offsetParent`).

Por ejemplo, `offsetParent` es `null`, y `offsetWidth`, `offsetHeight` son `0` cuando creamos un elemento, pero aún no lo han insertado en el documento, o eso (o su antecesor) tiene `display:none`.

Nosotros podemos usar esto para verificar si un elemento está oculto, así:

```
function isHidden(elem) {
  return !elem.offsetWidth && !elem.offsetHeight;
}
```

Por favor ten en cuenta que `isHidden` retorna `true` para elementos que están en pantalla, pero tienen tamaño cero (como un `<div>` vacío).

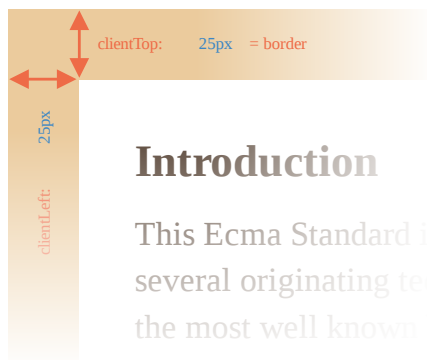
clientTop/Left

Dentro del elemento, nosotros tenemos los bordes.

Para medirlos, están las propiedades `clientTop` y `clientLeft`.

En nuestro ejemplo:

- `clientLeft = 25` – ancho del borde izquierdo
- `clientTop = 25` – ancho del borde superior



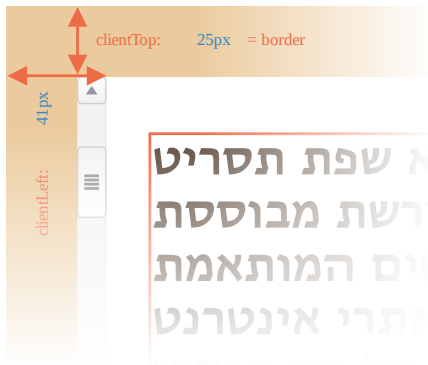
...Para ser precisos – Estas propiedades no son el ancho/alto del borde, sino las coordenadas relativas del lado interior del lado exterior.

Cuál es la diferencia?

Se vuelve visible cuando el documento está de derecha a izquierda (el sistema operativo está en idioma Árabe o Hebreo). La barra de desplazamiento no está a la derecha, sino a la izquierda, y luego `clientLeft` también incluye el ancho de la barra de desplazamiento.

En este caso, `clientLeft` no sería `25`, pero con ancho de la barra de desplazamiento $25 + 16 = 41$.

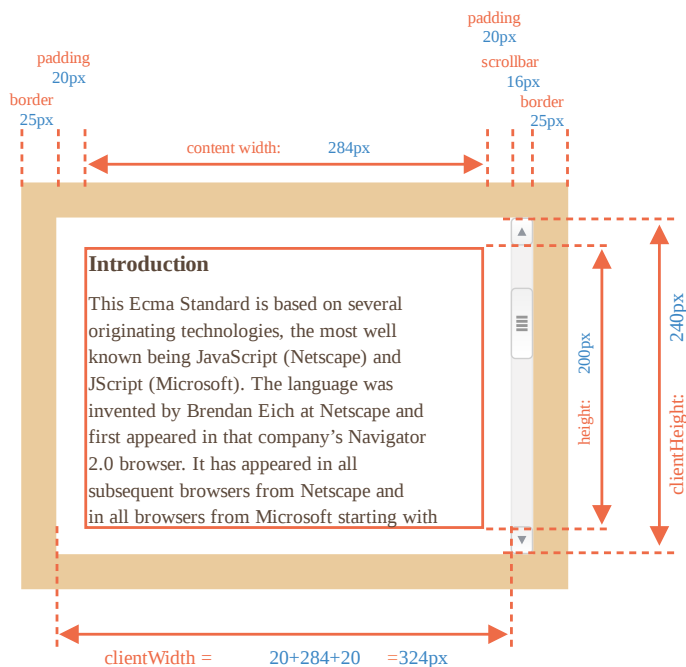
Aquí está el ejemplo en hebreo:



clientWidth/Height

Esta propiedad proporciona el tamaño del área dentro de los bordes del elemento.

Incluyen el ancho del contenido junto con los rellenos, pero sin la barra de desplazamiento:

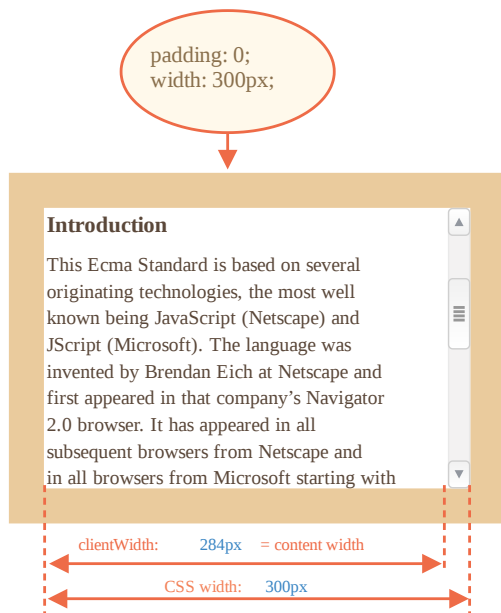


En la imagen de arriba, consideramos primero `clientHeight`.

No hay barra de desplazamiento horizontal, por lo que es exactamente la suma de lo que está dentro de los bordes: CSS-height 200px más relleno superior e inferior ($2 * 20\text{px}$) total 240px.

Ahora `clientWidth` – aquí el ancho del contenido no es 300px, sino 284px, porque los 16px son ocupados por la barra de desplazamiento. Entonces la suma es 284px más los rellenos de izquierda y derecha, total 324px.

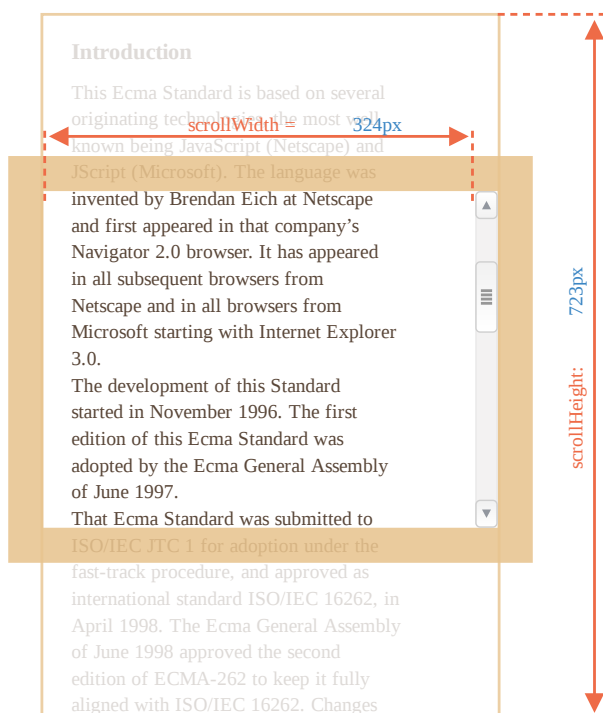
Si no hay rellenos, entonces `clientWidth/Height` es exactamente el área de contenido, dentro de los bordes y la barra de desplazamiento (si la hay).



Entonces, cuando no hay relleno, podremos usar `clientWidth/clientHeight` para obtener el tamaño del área de contenido.

scrollWidth/Height

Estas propiedades son como `clientWidth/clientHeight`, pero también incluyen las partes desplazadas (ocultas):

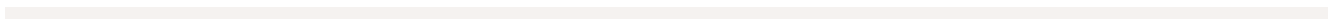


En la imagen de arriba:

- `scrollHeight = 723` – es la altura interior completa del área de contenido, incluyendo las partes desplazadas.
- `scrollWidth = 324` – es el ancho interior completo, aquí no tenemos desplazamiento horizontal, por lo que es igual a `clientWidth`.

Podemos usar estas propiedades para expandir el elemento a su ancho/alto completo.

Como esto:

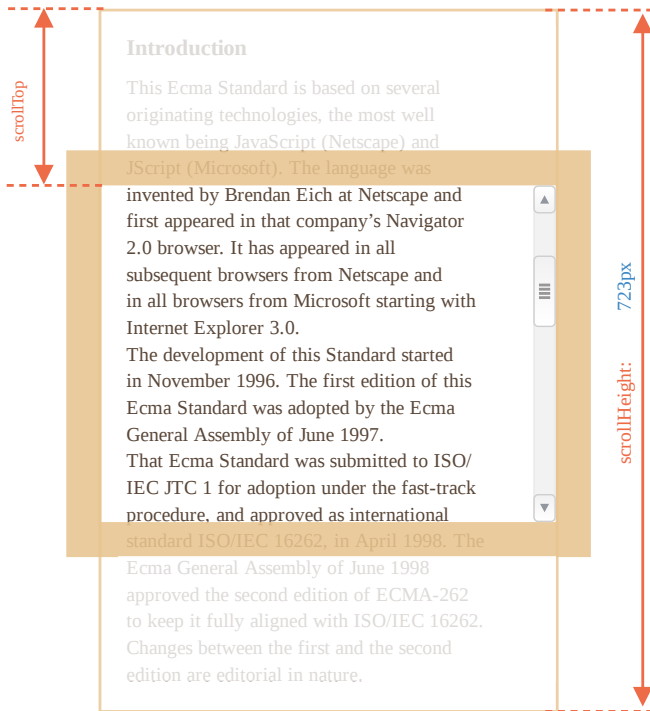


```
// expanda el elemento a la altura completa del contenido
element.style.height = `${element.scrollHeight}px`;
```

scrollTop/scrollLeft

Las propiedades `scrollTop/scrollLeft` son el ancho/alto de la parte oculta y desplazada del elemento.

En la imagen abajo podemos ver `scrollHeight` y `scrollTop` para un bloque con un desplazamiento vertical.



En otras palabras, `scrollTop` es “cuánto se desplaza hacia arriba”.

i `scrollTop/scrollLeft` puede ser modificado

La mayoría de las propiedades aquí son solo lectura, pero `scrollTop/scrollLeft` se puede cambiar, y el navegador desplazará el elemento.

Establecer `scrollTop` en `0` o un valor grande, como `1e9` hará que el elemento se desplace hacia arriba/abajo respectivamente.

No uses width/height de CSS

Acabamos de cubrir las propiedades geométricas de los elementos DOM, que se pueden usar para obtener anchos, alturas y calcular distancias.

Pero como sabemos por el capítulo [Estilos y clases](#), podemos leer CSS-height y width usando `getComputedStyle`.

Entonces, ¿Por qué no leer el ancho de un elemento con `getComputedStyle`, como este?

```
let elem = document.body;

alert( getComputedStyle(elem).width ); // muestra CSS width por elemento
```

¿Por qué deberíamos usar propiedades geométricas en su lugar? Hay dos razones:

1. Primero, CSS `width/height` dependen de otra propiedad: `box-sizing` que define “qué es” CSS width y height. Un cambio en `box-sizing` para propósitos de CSS puede romper dicho JavaScript.
2. Segundo, CSS `width/height` puede ser `auto`, por ejemplo para un elemento en línea:

```
<span id="elem">Hola!</span>

<script>
  alert( getComputedStyle(elem).width ); // auto
</script>
```

Desde el punto de vista de CSS, `width:auto` es perfectamente normal, pero en JavaScript necesitamos un tamaño exacto en `px` que podríamos usar en los cálculos. Entonces, aquí el ancho de CSS `width` es inútil.

Y hay una razón más: una barra de desplazamiento. A veces, el código que funcionaba bien sin una barra de desplazamiento tiene errores, porque una barra de desplazamiento toma el espacio del contenido en algunos navegadores. Entonces, el ancho real disponible para el contenido es *menor* que el ancho de CSS. Y `clientWidth/clientHeight` tiene eso en cuenta.

...Pero con `getComputedStyle(elem).width` la situación es diferente. Algunos navegadores (p.e. Chrome) devuelven el ancho interno real, menos la barra de desplazamiento, y algunos de ellos (p.e. Firefox) – CSS `width` (ignora la barra de desplazamiento). Estas diferencias entre los navegadores son la razón para no usar `getComputedStyle`, sino confiar en las propiedades geométricas.

Tenga en cuenta que la diferencia descrita es solo de leer `getComputedStyle(...).width` de JavaScript, visualmente todo es correcto.

Resumen

Los elementos tienen las siguientes propiedades geométricas:

- `offsetParent` – es el ancestro posicionado más cercano o `td`, `th`, `table`, `body`.
- `offsetLeft/offsetTop` – coordenadas relativas al borde superior izquierdo de `offsetParent`.
- `offsetWidth/offsetHeight` – ancho/alto “exterior” de un elemento, incluidos los bordes.
- `clientLeft/clientTop` – las distancias desde la esquina exterior superior izquierda a la esquina interior superior izquierda (contenido + relleno). Para los Sistemas Operativos de izquierda a derecha, siempre son los anchos de los bordes izquierdo/superior. Para los Sistemas Operativos de derecha a izquierda, la barra de desplazamiento está a la izquierda, por lo que `clientLeft` también incluye su ancho.
- `clientWidth/clientHeight` – el ancho/alto del contenido incluyendo rellenos, pero sin la barra de desplazamiento.
- `scrollWidth/scrollHeight` – el ancho/alto del contenido, al igual que `clientWidth/clientHeight`, pero también incluye la parte invisible desplazada del elemento.
- `scrollLeft/scrollTop` – ancho/alto de la parte superior desplazada del elemento, comenzando desde la esquina superior izquierda.

Todas las propiedades son solo lectura excepto `scrollLeft/scrollTop` que hacen que el navegador desplace el elemento si se cambia.

✓ Tareas

¿Qué es el desplazamiento desde la parte inferior?

importancia: 5

La propiedad `elem.scrollTop` es el tamaño desplazado desde la parte superior. ¿Cómo obtener el tamaño de la parte inferior desplazada (vamos a llamarlo `scrollBottom`)?

Escribe el código que funcione para un `elem` arbitrario.

P.S. Por favor revisa tu código: si no hay desplazamiento o el elemento está completamente desplazado, debería retornar `0`.

A solución

¿Qué es el ancho de la barra de desplazamiento?

importancia: 3

Escribe el código que retorna el tamaño de una barra de desplazamiento estándar.

Para Windows esto usualmente varía entre `12px` y `20px` . Si el navegador no reserva algún espacio para esto (la barra de desplazamiento es medio traslúcida sobre el texto, también pasa), entonces puede ser `0px` .

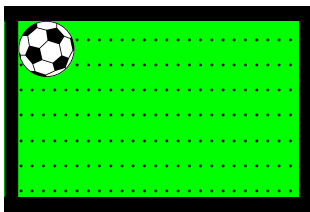
P.S. El código debería funcionar con cualquier documento HTML, no depende de su contenido.

[A solución](#)

Coloca la pelota en el centro del campo.

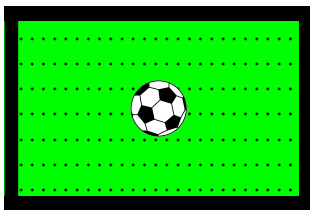
importancia: 5

Así es como se ve el documento de origen:



¿Cuáles son las coordenadas del centro de campo?

Calcúlalos y úsalos para colocar la pelota en el centro del campo verde:



- El elemento debe ser movido por JavaScript, no por CSS.
- El código debería funcionar con cualquier una pelota de cualquier tamaño (`10` , `20` , `30` pixels) y cualquier tamaño de campo, no debe estar vinculado a los valores dados.

P.S. Claro, el centrado se podría hacer con CSS, pero aquí lo queremos específicamente con JavaScript. Además, conoceremos otros temas y situaciones más complejas en las que se debe utilizar JavaScript. Aquí hacemos un “calentamiento”.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

La diferencia: CSS width versus clientWidth

importancia: 5

¿Cuál es la diferencia entre `getComputedStyle(elem).width` y `elem.clientWidth` ?

Dar al menos 3 diferencias. Mientras más, mejor.

[A solución](#)

Tamaño de ventana y desplazamiento

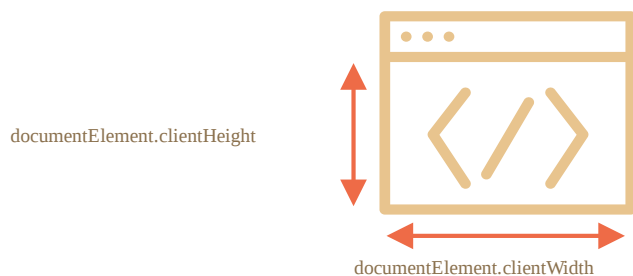
¿Cómo encontramos el ancho y el alto de la ventana del navegador? ¿Cómo obtenemos todo el ancho y la altura del documento, incluida la parte desplazada? ¿Cómo desplazamos la página usando JavaScript?

Para la mayoría de estas cuestiones, podemos usar el elemento de documento raíz

`document.documentElement`, que corresponde a la etiqueta `<html>`. Pero hay métodos y peculiaridades adicionales lo suficientemente importantes para considerar.

Ancho/alto de la ventana

Para obtener el ancho y alto de la ventana, podemos usar `clientWidth` / `clientHeight` de `document.documentElement`:



⚠ No `window.innerWidth/Height`

Los navegadores también admiten propiedades `window.innerWidth` / `innerHeight`. Se parecen a lo que queremos. Entonces, ¿por qué no usarlos?

Si existe una barra de desplazamiento, y ocupa algo de espacio, `clientWidth` / `clientHeight` proporciona el ancho / alto sin ella (resta el espacio desplazado). En otras palabras, devuelven ancho / alto de la parte visible del documento, disponible para el contenido.

... Y `window.innerWidth` / `innerHeight` incluye la barra de desplazamiento.

Si hay una barra de desplazamiento y ocupa algo de espacio, estas dos líneas muestran valores diferentes:

```
alert( window.innerWidth ); // ancho de la ventana completa
alert( document.documentElement.clientWidth ); // ancho de ventana menos el desplazamiento.
```

En la mayoría de los casos, necesitamos el ancho de ventana *disponible*, para dibujar o colocar algo. Es decir: el espacio del desplazamiento si hay alguno. Entonces deberíamos usar `documentElement.clientHeight/Width`.

⚠ DOCTYPE es importante

Tenga en cuenta que las propiedades de geometría de nivel superior pueden funcionar de manera un poco diferente cuando no hay `<!DOCTYPE HTML>` en HTML. Pueden suceder cosas extrañas.

En HTML moderno siempre debemos escribir `DOCTYPE`.

Ancho/Alto del documento

Teóricamente, como el elemento del documento raíz es `document.documentElement`, e incluye todo el contenido, podríamos medir el tamaño completo del documento con `document.documentElement.scrollHeight` / `scrollWidth`.

Pero en ese elemento, para toda la página, estas propiedades no funcionan según lo previsto. ¡En Chrome / Safari / Opera si no hay desplazamiento, entonces `documentElement.scrollHeight` puede ser incluso menor que `documentElement.clientHeight`! Suena como una tontería, raro, ¿verdad?

Para obtener de manera confiable la altura completa del documento, debemos tomar el máximo de estas propiedades:

```
let scrollHeight = Math.max(
```

```
document.body.scrollHeight, document.documentElement.scrollHeight,  
document.body.offsetHeight, document.documentElement.offsetHeight,  
document.body.clientHeight, document.documentElement.clientHeight  
);  
  
alert('Altura completa del documento, con parte desplazada: ' + scrollTop);
```

¿Por qué? Mejor no preguntes. Estas inconsistencias provienen de tiempos antiguos, no una lógica “inteligente”.

Obtener el desplazamiento actual

Los elementos DOM tienen su estado de desplazamiento actual en sus propiedades `elem.scrollLeft/scrollTop`.

El desplazamiento de documentos, `document.documentElement.scrollLeft / Top` funciona en la mayoría de los navegadores, excepto los más antiguos basados en WebKit, como Safari (bug [5991](#)), donde deberíamos usar `document.body` en lugar de `document.documentElement`.

Afortunadamente, no tenemos que recordar estas peculiaridades en absoluto, porque el desplazamiento está disponible en las propiedades especiales `window.pageXOffset/pageYOffset`:

```
alert('Desplazamiento actual desde la parte superior: ' + window.pageYOffset);  
alert('Desplazamiento actual desde la parte izquierda: ' + window.pageXOffset);
```

Estas propiedades son de solo lectura.

También disponible como propiedades window: `scrollX` y `scrollY`

Por razones históricas existen ambas propiedades pero ambas son lo mismo:

- `window.pageXOffset` es un alias de `window.scrollX`.
- `window.pageYOffset` es un alias de `window.scrollY`.

Desplazamiento: `scrollTo`, `scrollBy`, `scrollIntoView`

Importante:

para desplazar la página desde JavaScript, su DOM debe estar completamente construido.

Por ejemplo, si intentamos desplazar la página desde el script en `<head>`, no funcionará.

Los elementos regulares se pueden desplazar cambiando `scrollTop/scrollLeft`.

Nosotros podemos hacer lo mismo para la página usando `document.documentElement.scrollTop/Left` (excepto Safari, donde `document.body.scrollTop/Left` debería usarse en su lugar).

Alternativamente, hay una solución más simple y universal: métodos especiales [window.scrollBy\(x,y\)](#) y [window.scrollTo\(pageX,pageY\)](#).

- El método `scrollBy(x, y)` desplaza la página *en relación con su posición actual*. Por ejemplo, `scrollBy(0, 10)` desplaza la página 10px hacia abajo.
- El método `scrollTo(pageX, pageY)` desplaza la página *a coordenadas absolutas*, de modo que la esquina superior izquierda de la parte visible tiene coordenadas `(pageX, pageY)` en relación con la esquina superior izquierda del documento. Es como configurar `scrollLeft / scrollTop`.

Para desplazarnos hasta el principio, podemos usar `scrollTo(0, 0)`.

Estos métodos funcionan para todos los navegadores de la misma manera.

`scrollIntoView`

Para completar, cubramos un método más: [elem.scrollToView\(top\)](#) ↗.

La llamada a `elem.scrollToView(top)` desplaza la página para hacer visible `elem`. Tiene un argumento:

- si `top=true` (ese es el valor predeterminado), la página se desplazará para que aparezca `element` en la parte superior de la ventana. El borde superior del elemento está alineado con la parte superior de la ventana.
- si `top=false`, la página se desplaza para hacer que `element` aparezca en la parte inferior. El borde inferior del elemento está alineado con la parte inferior de la ventana.

Prohibir el desplazamiento

A veces necesitamos hacer que el documento sea “inescrutable”. Por ejemplo, cuando necesitamos cubrirlo con un mensaje grande que requiere atención inmediata, y queremos que el visitante interactúe con ese mensaje, no con el documento.

Para hacer que el documento sea inescrutable, es suficiente establecer `document.body.style.overflow="hidden"`. La página se congelará en su desplazamiento actual.

Podemos usar la misma técnica para “congelar” el desplazamiento para otros elementos, no solo para `document.body`.

El inconveniente del método es que la barra de desplazamiento desaparece. Si ocupaba algo de espacio, entonces ese espacio ahora es libre y el contenido “salta” para llenarlo.

Eso parece un poco extraño, pero puede solucionarse si comparamos `clientWidth` antes y después del congelamiento, y si aumentó (la barra de desplazamiento desapareció) luego agregue `padding` a `document.body` en lugar de la barra de desplazamiento, para que mantenga el ancho del contenido igual.

Resumen

Geometría:

- Ancho/alto de la parte visible del documento (área de contenido ancho / alto):
`document.documentElement.clientWidth/Height`
- Ancho/alto de todo el documento, con la parte desplazada:

```
let scrollHeight = Math.max(
  document.body.scrollHeight, document.documentElement.scrollHeight,
  document.body.offsetHeight, document.documentElement.offsetHeight,
  document.body.clientHeight, document.documentElement.clientHeight
);
```

Desplazamiento:

- Lee el desplazamiento actual: `window.pageYOffset/pageXOffset`.
- Cambia el desplazamiento actual:
 - `window.scrollTo(pageX, pageY)` – coordenadas absolutas
 - `window.scrollBy(x, y)` – desplazamiento relativo al lugar actual,
 - `elem.scrollToView(top)` – desplácese para hacer visible el `elem` (alineación con la parte superior / inferior de la ventana).

Coordenadas

Para mover elementos debemos estar familiarizados con las coordenadas.

La mayoría de los métodos de JavaScript tratan con uno de dos sistemas de coordenadas:

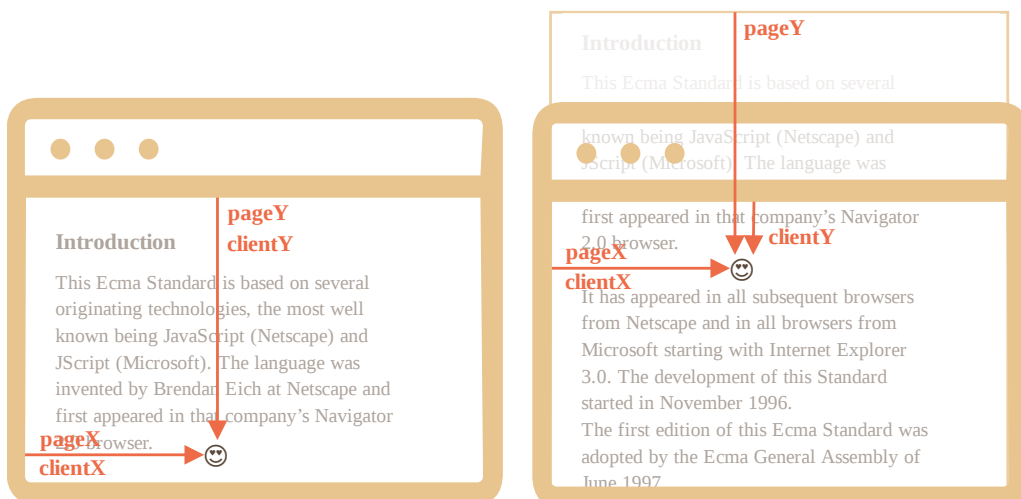
1. **Relativo a la ventana:** similar a `position: fixed`, calculado desde el borde superior / izquierdo de la ventana.
 - Designaremos estas coordenadas como `clientX/clientY`, el razonamiento para tal nombre se aclarará más adelante, cuando estudiemos las propiedades de los eventos.

2. **Relative al documento** – similar a `position: absolute` en la raíz del documento, calculado a partir del borde superior / izquierdo del documento.

- Las designaremos como `pageX/pageY`.

Cuando la página se desplaza hasta el comienzo, de modo que la esquina superior / izquierda de la ventana es exactamente la esquina superior / izquierda del documento, estas coordenadas son iguales entre sí. Pero después de que el documento cambia, las coordenadas relativas a la ventana de los elementos cambian, a medida que los elementos se mueven a través de la ventana, mientras que las coordenadas relativas al documento permanecen iguales.

En esta imagen tomamos un punto en el documento y demostramos sus coordenadas antes del desplazamiento (primera imagen) y después (segunda imagen):



Cuando el documento se desplazó:

- La coordenada `pageY` relativa al documento se mantuvo igual, se cuenta desde la parte superior del documento (ahora desplazada).
- La coordenada `clientY` relativa a la ventana cambió (la flecha se acortó), ya que el mismo punto se acercó a la parte superior de la ventana.

Coordenadas de elemento: `getBoundingClientRect`

El método `elem.getBoundingClientRect()` devuelve las coordenadas de la ventana para un rectángulo mínimo que encasilla a `elem` como un objeto de la clase interna [DOMRect](#).

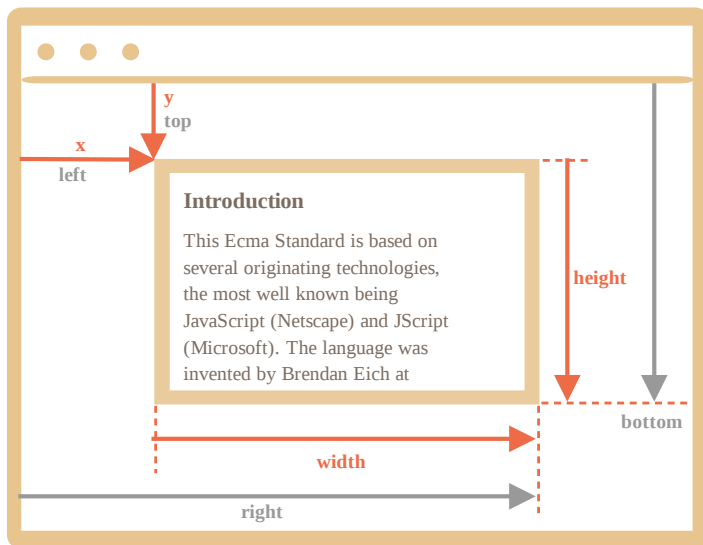
Propiedades principales de `DOMRect`:

- `x/y`: coordenadas X/Y del origen del rectángulo con relación a la ventana.
- `width/height`: ancho/alto del rectángulo (pueden ser negativos).

Adicionalmente existen estas propiedades derivadas:

- `top/bottom`: coordenada Y para el borde superior/inferior del rectángulo.
- `left/right`: coordenada X para el borde izquierdo/derecho del rectángulo.

Aquí hay la imagen con el output de `elem.getBoundingClientRect()`:



Como puedes ver `x/y` y `width/height` describen completamente el rectángulo. Las propiedades derivadas pueden ser calculadas a partir de ellas:

- `left = x`
- `top = y`
- `right = x + width`
- `bottom = y + height`

Toma en cuenta:

- Las coordenadas pueden ser fracciones decimales, tales como `10.5`. Esto es normal ya que internamente el navegador usa fracciones en los cálculos. No tenemos que redondearlos para poder asignarlos a `style.left/top`.
- Las coordenadas pueden ser negativas. Por ejemplo, si la página se desplaza hasta que `elem` rebase el borde superior de la ventana, entonces `elem.getBoundingClientRect().top` será negativo.

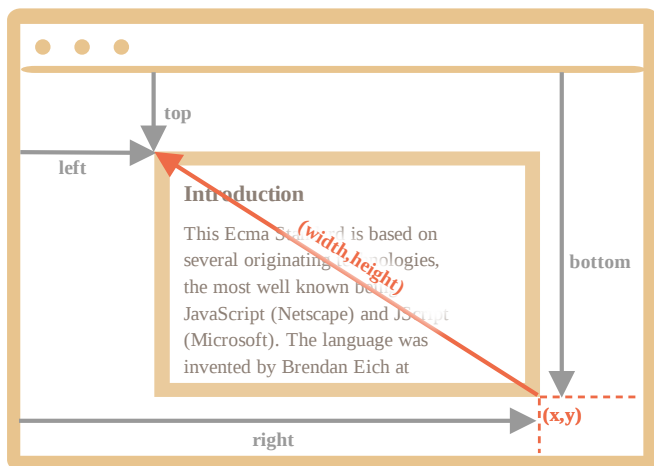
i ¿Por qué se necesitan propiedades derivadas? ¿Por qué `top/left` si existe `x/y`?

Matemáticamente un rectángulo se define de únicamente con su punto de partida (x, y) y el vector de dirección $(width, height)$. Por lo tanto, las propiedades derivadas adicionales son por conveniencia.

Técnicamente es posible que `width/height` sean negativos, lo que permite un rectángulo “dirigido”. Por ejemplo, para representar la selección del mouse con su inicio y final debidamente marcados.

Los valores negativos para `width/height` indican que el rectángulo comienza en su esquina inferior derecha y luego se extiende hacia la izquierda y arriba.

Aquí hay un rectángulo con valores `width` y `height` negativos (ejemplo: `width=-200`, `height=-100`):



Como puedes ver: `left/top` no es igual a `x/y` en tal caso.

Pero en la práctica `elem.getBoundingClientRect()` siempre devuelve el ancho y alto positivos. Aquí hemos mencionado los valores negativos para `width/height` solo para que comprendas por qué estas propiedades aparentemente duplicadas en realidad no lo son.

⚠ En Internet Explorer no hay soporte para `x/y`

Internet Explorer no tiene soporte para las propiedades `x/y` por razones históricas.

De manera que podemos crear un polyfill y (obtenerlo con `DomRect.prototype`) o solo usar `top/left`, ya que son siempre las mismas que `x/y` para `width/height` positivos, en particular en el resultado de `elem.getBoundingClientRect()`.

⚠ Las coordenadas `right/bottom` son diferentes a las propiedades de posición en CSS

Existen muchas similitudes obvias entre las coordenadas relativas a la ventana y `position:fixed` en CSS.

Pero en el posicionamiento con CSS, la propiedad `right` define la distancia entre el borde derecho y el elemento y la propiedad `bottom` supone la distancia entre el borde inferior y el elemento.

Si echamos un vistazo a la imagen anterior veremos que en JavaScript esto no es así. Todas las coordenadas de la ventana se cuentan a partir de la esquina superior izquierda, incluyendo estas.

elementFromPoint(x, y)

La llamada a `document.elementFromPoint(x, y)` devuelve el elemento más anidado dentro de las coordenadas de la ventana (x, y) .

La sintaxis es:

```
let elem = document.elementFromPoint(x, y);
```

Por ejemplo, el siguiente código resalta y muestra la etiqueta del elemento que ahora se encuentra en medio de la ventana:

```
let centerX = document.documentElement.clientWidth / 2;
let centerY = document.documentElement.clientHeight / 2;

let elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";
alert(elem.tagName);
```

Debido a que utiliza las coordenadas de la ventana, el elemento puede ser diferente dependiendo de la posición actual del scroll.

⚠ Para coordenadas fuera de la ventana, el `elementFromPoint` devuelve `null`

El método `document.elementFromPoint(x,y)` solo funciona si `(x,y)` se encuentra dentro del área visible.

Si alguna de las coordenadas es negativa o excede el ancho o alto de la ventana entonces devolverá `null`.

Aquí hay un error típico que podría ocurrir si no nos aseguramos de ello:

```
let elem = document.elementFromPoint(x, y);
// si las coordenadas sobrepasan la ventana entonces elem = null
elem.style.background = 'red'; // ¡Error!
```

Usándolas para posicionamiento “fijo”

La mayoría del tiempo necesitamos coordenadas para posicionar algo.

Para mostrar algo cercano a un elemento podemos usar `getBoundingClientRect` para obtener sus coordenadas y entonces CSS `position` junto con `left/top` (o `right/bottom`).

Por ejemplo, la función `createMessageUnder(elem, html)` a continuación nos muestra un mensaje debajo de `elem`:

```
let elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, html) {
  // Crea un elemento de mensaje
  let message = document.createElement('div');
  // Lo mejor es usar una clase css para el estilo aquí
  message.style.cssText = "position:fixed; color: red";

  // Asignando las coordenadas, no olvides "px"!
  let coords = elem.getBoundingClientRect();

  message.style.left = coords.left + "px";
  message.style.top = coords.bottom + "px";

  message.innerHTML = html;

  return message;
}

// Uso:
// agregarlo por 5 segundos en el documento
let message = createMessageUnder(elem, '¡Hola, mundo!');
document.body.append(message);
setTimeout(() => message.remove(), 5000);
```

El código puede ser modificado para mostrar el mensaje a la izquierda, derecha, abajo, aplicando animaciones con CSS para “desvanecerlo” y así. Es fácil una vez que tenemos todas las coordenadas y medidas del elemento.

Pero nota un detalle importante: cuando la página se desplaza, el mensaje se aleja del botón.

La razón es obvia: el elemento del mensaje se basa en `position:fixed`, esto lo reubica al mismo lugar en la ventana mientras se desplaza.

Para cambiar esto necesitamos usar las coordenadas basadas en el documento y `position:absolute`.

Coordenadas del documento

Las coordenadas relativas al documento comienzan en la esquina superior izquierda del documento, no de la ventana.

En CSS las coordenadas de la ventana corresponden a `position:fixed` mientras que las del documento son similares a `position:absolute` en la parte superior.

Podemos usar `position:absolute` y `top/left` para colocar algo en un lugar determinado del documento, esto lo reubicará ahí mismo durante un desplazamiento de página. Pero primero necesitamos las coordenadas correctas.

No existe un estándar para obtener las coordenadas de un elemento en un documento. Pero es fácil de codificarlo.

Los dos sistemas de coordenadas están relacionados mediante la siguiente fórmula:

- `pageY = clientY` + el alto de la parte vertical desplazada del documento.
- `pageX = clientX` + el ancho de la parte horizontal desplazada del documento.

La función `getCoords(elem)` toma las coordenadas de la ventana de `elem.getBoundingClientRect()` y agrega el desplazamiento actual a ellas:

```
// obteniendo las coordenadas en el documento del elemento
function getCoords(elem) {
  let box = elem.getBoundingClientRect();

  return {
    top: box.top + window.pageYOffset,
    right: box.right + window.pageXOffset,
    bottom: box.bottom + window.pageYOffset,
    left: box.left + window.pageXOffset
  };
}
```

Si el ejemplo anterior se usara con `position:absolute` entonces el mensaje podría permanecer cerca del elemento durante el desplazamiento.

La función modificada `createMessageUnder`:

```
function createMessageUnder(elem, html) {
  let message = document.createElement('div');
  message.style.cssText = "position:absolute; color: red";

  let coords = getCoords(elem);

  message.style.left = coords.left + "px";
  message.style.top = coords.bottom + "px";

  message.innerHTML = html;

  return message;
}
```

Resumen

Cualquier punto en la página tiene coordenadas:

1. Relativas a la ventana: `elem.getBoundingClientRect()`.
2. Relativas al documento: `elem.getBoundingClientRect()` mas el desplazamiento actual de la página.

Las coordenadas de la ventana son ideales para usarse con `position:fixed`, y las coordenadas del documento funcionan bien con `position:absolute`.

Ambos sistemas de coordenadas tienen pros y contras; habrá ocasiones en que ocuparemos una u otra, justamente como con los valores `absolute` y `fixed` para `position` en CSS.

✓ Tareas

Encuentra las coordenadas del campo en la ventana

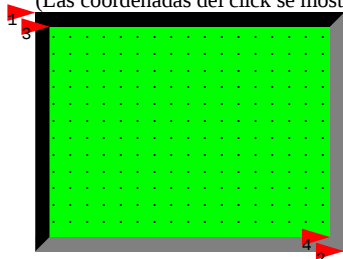
importancia: 5

En el siguiente iframe puedes ver un documento con el “campo” verde.

Usa JavaScript para encontrar las coordenadas de las esquinas de la ventana señaladas con las flechas.

Hay una pequeña característica implementada en el documento para conveniencia. Un click en cualquier lugar mostrará las coordenadas ahí.

Haz click en cualquier lugar para obtener las coordenadas de la ventana.
Esto es útil para testear y confirmar el resultado que obtuviste con JavaScript.
(Las coordenadas del click se mostrarán aquí)



Tu código debe usar el DOM para obtener las coordenadas en la ventana de:

1. La esquina superior izquierda externa (eso es simple).
2. La esquina inferior derecha externa (simple también).
3. La esquina superior izquierda interna (un poco más difícil).
4. La esquina inferior derecha interna (existen muchas maneras, elige una).

Las coordenadas que tú calcules deben ser iguales a las devueltas por el click del mouse.

P.D. El código también debe funcionar si el elemento tiene otro tamaño o borde, no está ligado a ningún valor fijo.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Muestra una nota cercana al elemento

importancia: 5

Crea una función `positionAt(anchor, position, elem)` que posicione `elem`, dependiendo de la proximidad de `position` al elemento `anchor`.

`position` debe ser un string con alguno de estos 3 valores:

- `"top"` – posiciona `elem` encima de `anchor`
- `"right"` – posiciona `elem` inmediatamente a la derecha de `anchor`
- `"bottom"` – posiciona `elem` debajo de `anchor`

Esto será usado dentro de la función `showNote(anchor, position, html)`, proveída en el código fuente de la tarea, que crea un elemento “note” con el `html` y lo muestra en el lugar proporcionado por `position` cercano a `anchor`.

Aquí está el demo de las notas:

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

nota encima

Maestra: Por qué llegas tarde?
Alumno: Alguien perdió un billete de cien dólares.
Maestra: Que bueno. Lo estabas ayudando a buscarlo?
Alumno: No. Estaba parado encima del billete.

nota a la derecha

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

nota debajo

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Muestra una nota cercana al elemento (absolute)

importancia: 5

Modifica la solución de la [tarea previa](#) de manera que la nota use `position: absolute` en lugar de `position: fixed`.

Esto evitará que se “aleje” del elemento cuando se desplace la página.

Toma la solución de la tarea anterior como punto de partida. Para testear el scroll, agrega el estilo `<body style="height: 2000px">`.

[A solución](#)

Posiciona la nota adentro (absolute)

importancia: 5

Ampliando a la tarea anterior [Muestra una nota cercana al elemento \(absolute\)](#): enséñale a la función `positionAt(anchor, position, elem)` a insertar `elem` dentro de `anchor`.

Los nuevos valores para posición son `position`:

- `top-out`, `right-out`, `bottom-out` – funciona igual que antes, inserta el `elem` encima, a la derecha o debajo de `anchor`.
- `top-in`, `right-in`, `bottom-in` – inserta el `elem` dentro del `anchor`: lo fija en la parte superior, derecha o inferior del borde.

Por ejemplo:

```
// Muestra la nota encima de la cita textual
positionAt(blockquote, "top-out", note);

// Muestra la nota dentro de la cita textual en la parte superior
positionAt(blockquote, "top-in", note);
```

El resultado:

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

nota superior externa

nota superior interna

Maestra: Por qué llegas tarde?

Alumno: Alguien perdió un billete de cien dólares.

Maestra: Que bueno. Lo estabas ayudando a buscarlo?

Alumno: No. Estaba parado encima del billete.

nota a la derecha externa

nota inferior interna

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

Para el código fuente toma la solución de la tarea [Muestra una nota cercana al elemento \(absolute\)](#).

[A solución](#)

Introducción a los eventos

Una introducción a los eventos del navegador, las propiedades de los eventos y los patrones de manejo.

Introducción a los eventos en el navegador

Un evento es una señal de que algo ocurrió. Todos los nodos del DOM generan dichas señales (pero los eventos no están limitados sólo al DOM).

Aquí hay una lista con los eventos del DOM más utilizados, solo para echar un vistazo:

Eventos del mouse:

- `click` – cuando el mouse hace click sobre un elemento (los dispositivos touch lo generan con un toque).
- `contextmenu` – cuando el mouse hace click derecho sobre un elemento.
- `mouseover` / `mouseout` – cuando el cursor del mouse ingresa / abandona un elemento.
- `mousedown` / `mouseup` – cuando el botón del mouse es presionado / soltado sobre un elemento.
- `mousemove` – cuando el mouse se mueve.

Eventos del teclado:

- `keydown` / `keyup` – cuando se presiona / suelta una tecla.

Eventos del elemento form:

- `submit` – cuando el visitante envía un `<form>`.
- `focus` – cuando el visitante se centra sobre un elemento, por ejemplo un `<input>`.

Eventos del documento:

- `DOMContentLoaded` --cuando el HTML es cargado y procesado, el DOM está completamente construido

Eventos del CSS:

- `transitionend` – cuando una animación CSS concluye.

Hay muchos más eventos. Entraremos en detalle con eventos en particular en los siguientes capítulos.

Controladores de eventos

Para reaccionar con eventos podemos asignar un *handler* (*controlador*) el cual es una función que se ejecuta en caso de un evento.

Los handlers son una forma de ejecutar código JavaScript en caso de acciones por parte del usuario.

Hay muchas maneras de asignar un handler. Vamos a verlas empezando por las más simples.

Atributo HTML

Un handler puede ser establecido en el HTML con un atributo llamado `on<event>`.

Por ejemplo, para asignar un handler `click` para un `input` podemos usar `onclick`, como aquí:

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

Al hacer click, el código dentro de `onclick` se ejecuta.

Toma en cuenta que dentro de `onclick` usamos comillas simples, porque el atributo en sí va entre comillas dobles.

Si olvidamos que el código está dentro del atributo y usamos comillas dobles dentro, así:

`onclick="alert("Click!")"`, no funcionará correctamente.

Un atributo HTML no es un lugar conveniente para escribir un motón de código, así que mejor creamos una función JavaScript y la llamamos allí.

Aquí un click ejecuta la función `countRabbits()`:

```
<script>
function countRabbits() {
  for(let i=1; i<=3; i++) {
    alert("Conejo número " + i);
  }
}
</script>

<input type="button" onclick="countRabbits()" value="¡Cuenta los conejos!">
```

¡Cuenta los conejos!

Como ya sabemos, los nombres de los atributos HTML no distinguen entre mayúsculas y minúsculas, entonces `ONCLICK` funciona bien al igual que `onClick` y `onCLICK` ... Pero usualmente los atributos van con minúsculas: `onclick`.

Propiedad del DOM

Podemos asignar un handler usando una propiedad del DOM `on<event>`.

Por ejemplo, `elem.onclick`:

```
<input id="elem" type="button" value="Haz click en mí">
<script>
  elem.onclick = function() {
    alert('¡Gracias!');
  };
</script>
```

Haz click en mí

Si el handler es asignado usando un atributo HTML entonces el navegador lo lee, crea una nueva función desde el contenido del atributo y lo escribe en la propiedad del DOM.

Esta forma en realidad es la misma que ya habíamos visto antes.

Estás dos piezas de código funcionan igual:

1. Solo HTML:

```
<input type="button" onclick="alert('¡Click!')" value="Botón">
```

Botón

2. HTML + JS:

```
<input type="button" id="button" value="Botón">
<script>
  button.onclick = function() {
    alert('¡Click!');
  };
</script>
```

Botón

En el primer ejemplo el atributo HTML es usado para inicializar el `button.onclick`. Mientras que en el segundo ejemplo se usa el script, esa es toda la diferencia.

Como solo hay una propiedad `onclick`, no podemos asignar más de un handler.

En el siguiente ejemplo se agrega un handler con JavaScript que sobrescribe el handler existente:

```
<input type="button" id="elem" onclick="alert('Antes')" value="¡Haz click en mí!">
<script>
  elem.onclick = function() { // sobrescribe el handler existente
    alert('Después'); // solo se mostrará este
  };
</script>
```

¡Haz click en mí!

Para eliminar un handler, asigna `elem.onclick = null`.

Accediendo al elemento: `this`

El valor de `this` dentro de un handler es el elemento. El cuál tiene el handler dentro.

En el siguiente código el `button` muestra su contenido usando `this.innerHTML`:

```
<button onclick="alert(this.innerHTML)">Haz click en mí</button>
```

Haz click en mí

Posibles errores

Si estás empezando a trabajar con eventos, por favor, nota algunas sutilezas.

Nosotros podemos establecer una función existente como un handler:

```
function sayThanks() {
  alert('¡Gracias!');
}

elem.onclick = sayThanks;
```

Pero ten cuidado: la función debe ser asignada como `sayThanks`, no `sayThanks()`.

```
// correcto
button.onclick = sayThanks;

// incorrecto
button.onclick = sayThanks();
```

Si agregamos paréntesis, entonces `sayThanks()` se convierte en una llamada de función. Por lo que en realidad la última línea toma el *resultado* de la ejecución de la función, que es `undefined` (ya que la función no devuelve nada), y lo asigna a `onclick`. Esto no funciona.

...Por otro lado, en el markup necesitamos los paréntesis:

```
<input type="button" id="button" onclick="sayThanks()">
```

La diferencia es fácil de explicar. Cuando el navegador lee el atributo crea una función handler con cuerpo a partir del contenido del atributo.

Por lo que el markup genera esta propiedad:

```
button.onclick = function() {  
  sayThanks(); // <-- el contenido del atributo va aquí  
};
```

No uses `setAttribute` para handlers.

Tal llamada no funcionará:

```
// un click sobre <body> generará errores,  
// debido a que los atributos siempre son strings, la función se convierte en un string  
document.body.setAttribute('onclick', function() { alert(1) });
```

Las mayúsculas en las propiedades DOM importan.

Asignar un handler a `elem.onclick`, en lugar de `elem.ONCLICK`, ya que las propiedades DOM son sensibles a mayúsculas.

addEventListener

El problema fundamental de las formas ya mencionadas para asignar handlers es que no podemos asignar múltiples handlers a un solo evento.

Digamos que una parte de nuestro código quiere resaltar un botón al hacer click, y otra quiere mostrar un mensaje en el mismo click.

Nos gustaría asignar dos handlers de eventos para eso. Pero una nueva propiedad DOM sobrescribirá la que ya existe:

```
input.onclick = function() { alert(1); }  
// ...  
input.onclick = function() { alert(2); } // reemplaza el handler anterior handler
```

Los desarrolladores de los estándares de la web entendieron eso hace mucho tiempo y sugirieron una forma alternativa de administrar los handlers utilizando los métodos especiales `addEventListener` and `removeEventListener`. Estos están libres de ese problema.

La sintaxis para agregar un handler:

```
element.addEventListener(event, handler, [options]);
```

event

Nombre del evento, por ejemplo: `"click"`.

handler

La función handler.

options

Un objeto adicional, opcional, con las propiedades:

- `once`: si es `true` entonces el listener se remueve automáticamente después de activarlo.
- `capture`: la fase en la que se controla el evento, que será cubierta en el capítulo [Propagación y captura](#). Por razones históricas, `options` también puede ser `false/true`, lo que es igual a `{capture: false/true}`.
- `passive`: si es `true` entonces el handler no llamará a `preventDefault()`, esto lo explicaremos más adelante en [Acciones predeterminadas del navegador](#).

Para remover el handler, usa `removeEventListener`:

```
element.removeEventListener(event, handler, [options]);
```

⚠ Remove requiere la misma función

Para remover un handler deberemos pasar exactamente la misma función que asignamos.

Esto no funciona:

```
elem.addEventListener( "click" , () => alert('¡Gracias!'));  
// ....  
elem.removeEventListener( "click", () => alert('¡Gracias!'));
```

El handler no será removido porque `removeEventListener` obtiene otra función, con el mismo código, pero eso no importa, ya que es un objeto de función diferente.

Aquí está la manera correcta:

```
function handler() {  
  alert( '¡Gracias!' );  
}  
  
input.addEventListener("click", handler);  
// ....  
input.removeEventListener("click", handler);
```

Por favor nota que si no almacenamos la función en una variable entonces no podremos removerla. No hay forma de “volver a leer” los handlers asignados por `addEventListener`.

Múltiples llamadas a `addEventListener` permiten agregar múltiples handlers, como este:

```
<input id="elem" type="button" value="Haz click en mí"/>  
  
<script>  
  function handler1() {  
    alert('¡Gracias!');  
  };  
  
  function handler2() {  
    alert('¡Gracias de nuevo!');  
  }  
  
  elem.onclick = () => alert("Hola");  
  elem.addEventListener("click", handler1); // Gracias!  
  elem.addEventListener("click", handler2); // Gracias de nuevo!  
</script>
```

Como podemos ver en el ejemplo anterior, podemos establecer handlers *tanto* usando un propiedad DOM como `addEventListener` juntos. Pero por lo general solo usamos una de esas maneras.

⚠ Para algunos eventos, los handlers solo funcionan con `addEventListener`

Hay eventos que no pueden ser asignados por medio de una propiedad del DOM. Solamente con `addEventListener`.

Por ejemplo, el evento `DOMContentLoaded`, que se activa cuando el documento está cargado y el DOM está construido.

```
// nunca se ejecutará
document.onDOMContentLoaded = function() {
  alert("DOM construido");
};
```

```
// Así sí funciona
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM construido");
});
```

Por lo que `addEventListener` es más universal. Aún así, tales eventos son una excepción más que la regla.

Objeto del evento

Pero para manejar correctamente un evento nos gustaría saber todavía más acerca de lo que está pasando. No solo si fue un "click" o un "teclazo", sino ¿cuáles eran coordenadas del cursor o qué tecla fue oprimida? Y así.

Cuando un evento ocurre, el navegador crea un *objeto del evento* que coloca los detalles dentro y los pasa como un argumento al handler.

Aquí hay un ejemplo para obtener las coordenadas del cursor a partir del objeto del evento:

```
<input type="button" value="Haz click en mí!" id="elem">

<script>
  elem.onclick = function(event) {
    // muestra el tipo de evento, el elemento y las coordenadas del click
    alert(event.type + " en el " + event.currentTarget);
    alert("Coordenadas: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Algunas propiedades del objeto `event`:

`event.type`

Tipo de evento, en este caso fue `"click"`.

`event.currentTarget`

Elemento que maneja el evento. Lo que exactamente igual a `this`, a menos que el handler sea una función de flecha o su `this` esté vinculado a otra cosa, entonces podemos obtener el elemento desde `event.currentTarget`.

`event.clientX` / `event.clientY`

Coordenadas del cursor relativas a la ventana, para eventos de cursor.

Hay más propiedades. La mayoría dependen del tipo de evento: los eventos del teclado tienen algunas propiedades establecidas, las de cursor otras, las estudiaremos después cuando lleguemos a los detalles de diferentes eventos.

El objeto del evento también está disponible para handlers HTML

Si asignamos un handler en HTML también podemos usar el objeto `event` , así:

```
<input type="button" onclick="alert(event.type)" value="Event type">
```

Event type

Esto es posible porque cuando el navegador lee el atributo, crea un handler como este: `function(event) { alert(event.type) }` . Lo que significa que el primer argumento es llamado `"event"` y el cuerpo es tomado del atributo.

Objetos handlers: `handleEvent`

Podemos asignar no solo una función, sino un objeto como handler del evento usando `addEventListener` . Cuando el evento ocurre, el método `handleEvent` es llamado.

Por ejemplo:

```
<button id="elem">Haz click en mí</button>

<script>
  let obj = {
    handleEvent(event) {
      alert(event.type + " en " + event.currentTarget);
    }
  };

  elem.addEventListener('click', obj);
</script>
```

Como podemos ver, cuando `addEventListener` recibe como handler a un objeto, llama a `obj.handleEvent(event)` en caso de un evento.

También podríamos usar una clase para ello:

```
<button id="elem">Haz click en mí</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "Botón del mouse presionado";
          break;
        case 'mouseup':
          elem.innerHTML += "...y soltado.";
          break;
      }
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Aquí el mismo objeto maneja ambos eventos. Nota que necesitamos configurar explícitamente los eventos a escuchar usando `addEventListener` . El objeto `menu` solo obtiene `mousedown` y `mouseup` aquí, no hay ningún otro tipo de eventos.

El método `handleEvent` no tiene que hacer todo el trabajo por sí solo. En su lugar puede llamar a otros métodos específicos de eventos, como este:

```
<button id="elem">Haz click en mí</button>

<script>
class Menu {
  handleEvent(event) {
    // mousedown -> onMousedown
    let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
    this[method](event);
  }

  onMousedown() {
    elem.innerHTML = "Botón del mouse presionado";
  }

  onMouseup() {
    elem.innerHTML += "...y soltado.";
  }
}

let menu = new Menu();
elem.addEventListener('mousedown', menu);
elem.addEventListener('mouseup', menu);
</script>
```

Ahora los handlers del evento están claramente separados, lo que puede ser más fácil de mantener.

Resumen

Hay tres formas de asignar handlers:

1. Atributos HTML: `onclick="..."`.
2. Propiedades del DOM: `elem.onclick = function`.
3. Métodos: `elem.addEventListener(event, handler[, phase])` para agregar ó `removeEventListener` para remover.

Los atributos HTML se usan con moderación, porque JavaScript en medio de una etiqueta HTML luce un poco extraño y ajeno. Además no podemos escribir montones de código ahí.

Las propiedades del DOM son buenas para usar, pero no podemos asignar más de un handler a un evento en particular. En la mayoría de casos esta limitación no es apremiante.

La última forma es la más flexible, pero también es la más larga para escribir. Hay pocos eventos que solo funcionan con ésta, por ejemplo `transitionend` y `DOMContentLoaded` (que veremos después). Además `addEventListener` soporta objetos como handlers de eventos. En este caso `handleEvent` es llamado en caso del evento.

No importa como asignes el handler, este obtiene un objeto como primer argumento. Este objeto contiene los detalles sobre lo que pasó.

Vamos a aprender más sobre eventos en general y sobre diferentes tipos de eventos en los siguientes capítulos.

✓ Tareas

Ocultar con un click

importancia: 5

Agrega JavaScript al `button` para hacer que `<div id="text">` desaparezca al clickearlo.

El demo:

Haz click para desaparecer el texto

Texto

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Ocultarse

importancia: 5

Crea un botón que se oculte a sí mismo al darle un click.

[A solución](#)

¿Qué handlers se ejecutan?

importancia: 5

Hay un botón en la variable. No hay handlers en él.

¿Qué handlers se ejecutan con el click después del siguiente código? ¿Qué alertas se muestran?

```
button.addEventListener("click", () => alert("1"));  
  
button.removeEventListener("click", () => alert("1"));  
  
button.onclick = () => alert(2);
```

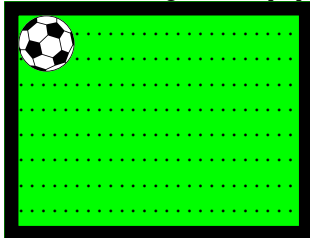
[A solución](#)

Mueve el balón por el campo

importancia: 5

Mueve el balón por el campo con un click. Así:

Haz click en un lugar del campo para mover el balón allí.



Requerimientos:

- El centro del balón debe quedar exactamente bajo el cursor al hacer click (sin atravesar el borde del campo si es posible).
- Las animaciones CSS son bienvenidas.
- El balón no debe cruzar los límites del campo.
- Cuando la página se desplace nada se debe romper.

Notas:

- El código también debe funcionar con medidas diferentes de campo y balón, no debe estar asociado a ningún valor fijo.

- Usa las propiedades `event.clientX/event.clientY` para las coordenadas del click.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Crear un menú deslizante

importancia: 5

Crea un menú que se abra/colapse al hacer click:

► Sweeties (click me)!

P.D. El HTML/CSS del documento fuente se debe modificar.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Agregar un botón de cierre

importancia: 5

Hay una lista de mensajes.

Usa JavaScript para agregar un botón de cierre en la esquina superior derecha de cada mensaje.

El resultado debería verse algo así:

Horse

[X]

The horse is one of two extant subspecies of *Equus ferus*. It is an odd-toed ungulate mammal belonging to the taxonomic family Equidae. The horse has evolved over the past 45 to 55 million years from a small multi-toed creature, *Eohippus*, into the large, single-toed animal of today.

Donkey

[X]

The donkey or ass (*Equus africanus asinus*) is a domesticated member of the horse family, Equidae. The wild ancestor of the donkey is the African wild ass, *E. africanus*. The donkey has been used as a working animal for at least 5000 years.

Cat

[X]

The domestic cat (Latin: *Felis catus*) is a small, typically furry, carnivorous mammal. They are often called house cats when kept as indoor pets or simply cats when there is no need to distinguish them from other felids and felines. Cats are often valued by humans for companionship and for their ability to hunt vermin.

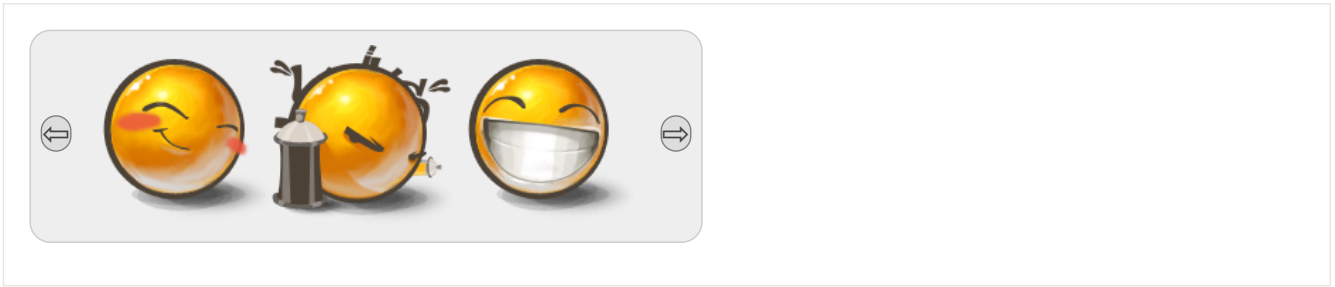
[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Carrusel

importancia: 4

Crea un “carrusel”: una cinta de imágenes que se puede desplazar haciendo clic en las flechas.



Más adelante podemos agregarle más funciones: desplazamiento infinito, carga dinámica, etc.

P.D. Para esta tarea, la estructura HTML / CSS es en realidad el 90% de la solución.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Propagación y captura

Vamos a empezar con un ejemplo.

Este manejador está asignado a `<div>`, pero también se ejecuta si haces clic a cualquier elemento anidado como `` ó `<code>`:

```
<div onclick="alert('¡El manejador!')">
  <em>Si haces clic en<code>EM</code>, el manejador en <code>DIV</code> es ejecutado.</em>
</div>
```

Si haces clic enEM, el manejador en DIV es ejecutado.

¿No es un poco extraño? ¿Por qué el manejador en `<div>` es ejecutado, si el clic fue hecho en ``?

Propagación

El principio de propagación es simple.

Cuando un evento ocurre en un elemento, este primero ejecuta los manejadores que tiene asignados, luego los manejadores de su padre, y así hasta otros ancestros.

Díganos que tenemos 3 elementos anidados FORM > DIV > P con un manejador en cada uno de ellos:

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

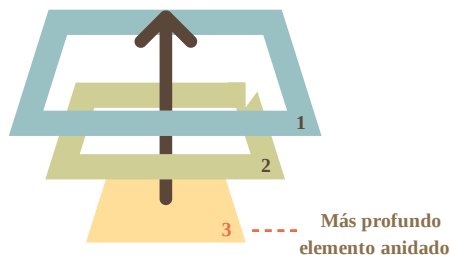
FORM

DIV

P

Un clic en el elemento del interior `<p>` primero ejecuta `onclick`:

1. En ese `<p>`.
2. Luego en el `<div>` de arriba.
3. Luego en el `<form>` de más arriba.
4. Y así sucesivamente hasta el objeto `document`.



Así si hacemos clic en `<p>`, entonces veremos 3 alertas: `p` → `div` → `form`.

Este proceso se conoce como “propagación” porque los eventos “se propagan” desde el elemento más al interior, a través de los padres, como una burbuja en el agua.

⚠ Casi todos los elementos se propagan.

La palabra clave en esta frase es “casi”.

Por ejemplo, un evento `focus` no se propaga. Hay otros ejemplos también, los veremos. Pero aún así, esta es la excepción a la regla, la mayoría de eventos sí se propagan.

event.target

Un manejador en un elemento padre siempre puede obtener los detalles sobre dónde realmente ocurrió el evento.

El elemento anidado más profundo que causó el evento es llamado elemento *objetivo*, accesible como `event.target`

Nota la diferencia de `this` (`= event.currentTarget`):

- `event.target` – es el elemento “objetivo” que inició el evento, no cambia a través de todo el proceso de propagación.
- `this` – es el elemento “actual”, el que tiene un manejador ejecutándose en el momento.

Por ejemplo, si tenemos un solo manejador `form.onclick`, este puede `atrapar` todos los clics dentro del formulario. No importa dónde el clic se hizo, se propaga hasta el `<form>` y ejecuta el manejador.

En el manejador `form.onclick`:

- `this` (`= event.currentTarget`) es el elemento `<form>`, porque el manejador se ejecutó en él.
- `event.target` es el elemento actual dentro de el formulario al que se le hizo clic.

Mira esto:

<https://plnkr.co/edit/Ffx43GvPqLxJxTfH?p=preview>

Es posible que `event.target` sea igual a `this`: ocurre cuando el clic se hace directamente en el elemento `<form>`.

Detener la propagación

Una propagación de evento empieza desde el elemento objetivo hacia arriba. Normalmente este continúa hasta `<html>` y luego hacia el objeto `document`, algunos eventos incluso alcanzan `window`, llamando a todos los manejadores en el camino.

Pero cualquier manejador podría decidir que el evento se ha procesado por completo y detener su propagación.

El método para esto es `event.stopPropagation()`.

Por ejemplo, aquí `body.onclick` no funciona si haces clic en `<button>`:

```
<body onclick="alert(`No se propagó hasta aquí`)">
  <button onclick="event.stopPropagation()">Haz clic</button>
</body>
```

Haz clic

`event.stopImmediatePropagation()`

Si un elemento tiene múltiples manejadores para un solo evento, aunque uno de ellos detenga la propagación, los demás aún se ejecutarán.

En otras palabras, `event.stopPropagation()` detiene la propagación hacia arriba, pero todos los manejadores en el elemento actual se ejecutarán.

Para detener la propagación y prevenir que los manejadores del elemento actual se ejecuten, hay un método `event.stopImmediatePropagation()`. Después de él, ningún otro manejador será ejecutado.

¡No detengas la propagación si no es necesario!

La propagación es conveniente. No la detengas sin una necesidad real, obvia y arquitectónicamente bien pensada.

A veces `event.stopPropagation()` crea trampas ocultas que luego se convierten en problemas.

Por ejemplo:

1. Creamos un menú anidado. Cada submenú maneja los clics en sus elementos y ejecuta `stopPropagation` para que el menu de arriba no se desencadene.
2. Luego decidimos atrapar los clic en toda la ventana, para seguir el rastro del comportamiento del usuario (donde hacen clic). Algunos sistemas de análisis hacen eso. Usualmente el código usa `document.addEventListener('click',...)` para atrapar todos los clics.
3. Nuestro análisis no funcionará sobre el área dónde los clics son detenidos por `stopPropagation`. Tristemente, tenemos una “zona muerta”.

Usualmente no hay una necesidad real para prevenir la propagación, pero una tarea que aparentemente lo requiera puede ser resuelta por otros medios. Uno de ellos es usar eventos personalizados, cubriremos eso más tarde. También podemos escribir nuestros datos en el objeto `event` en un manejador y leerlo en otro, para así poder pasar información sobre el proceso de abajo a los manejadores en los padres.

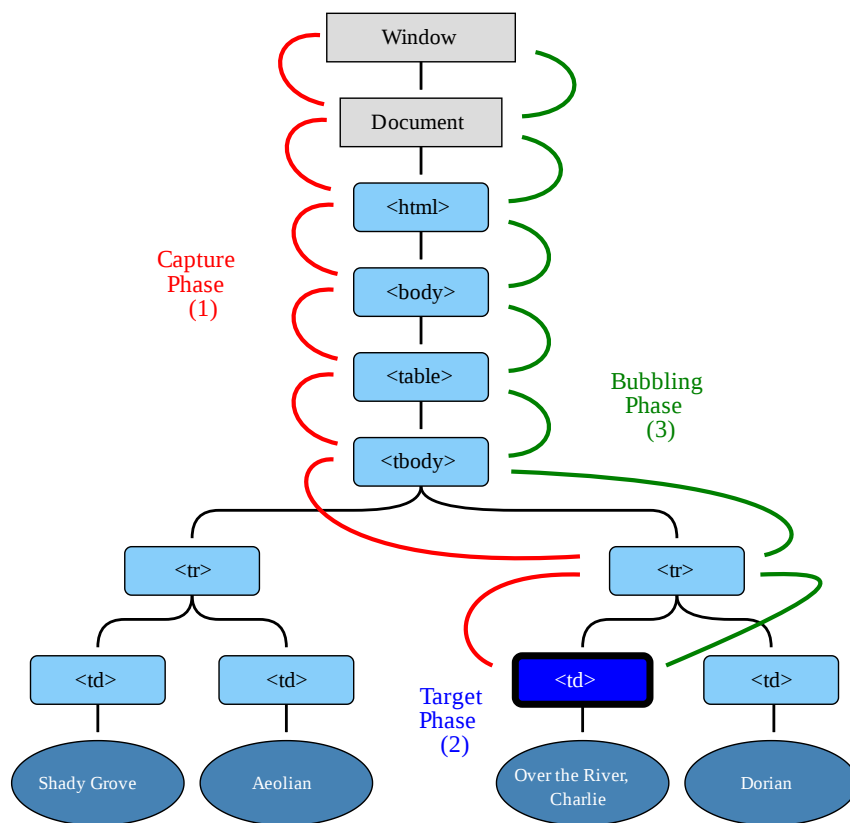
Captura

Hay otra fase en el procesamiento de eventos llamada “captura”. Es raro usarla en código real, pero a veces puede ser útil.

El estándar de [eventos del DOM](#) describe 3 fases de la propagación de eventos:

1. Fase de captura – el evento desciende al elemento.
2. Fase de objetivo – el evento alcanza al elemento.
3. Fase de propagación – el evento se propaga hacia arriba del elemento.

Aquí está la imagen de un clic en `<td>` dentro de una tabla, tomada desde la especificación:



Se explica así: por un clic en `<td>` el evento va primero a través de la cadena de ancestros hacia el elemento (fase de captura), luego alcanza el objetivo y se desencadena ahí (fase de objetivo), y por último va hacia arriba (fase de propagación), ejecutando los manejadores en su camino.

Antes solo hablamos de la propagación porque la fase de captura es raramente usada. Normalmente es invisible a nosotros.

Los manejadores agregados usando la propiedad `on<event>` ó usando atributos HTML ó `addEventListener(event, handler)` con dos argumentos no ejecutarán la fase de captura, únicamente ejecutarán la 2da y 3ra fase.

Para atrapar un evento en la fase de captura, necesitamos preparar la opción `capture` como `true` en el manejador:

```
elem.addEventListener(..., {capture: true})
// o, solo "true" es una forma más corta de {capture: true}
elem.addEventListener(..., true)
```

Hay dos posibles valores para la opción `capture` :

- Si es `false` (por defecto), entonces el manejador es preparado para la fase de propagación.
- Si es `true` , entonces el manejador es preparado para la fase de captura.

Es de notar que mientras formalmente hay 3 fases, la 2da fase ("la fase de objetivo": el evento alcanzó el elemento) no es manejada de forma separada; los manejadores en ambas fases, la de captura y propagación, se disparan en esa fase.

Veamos ambas fases, captura y propagación, en acción:

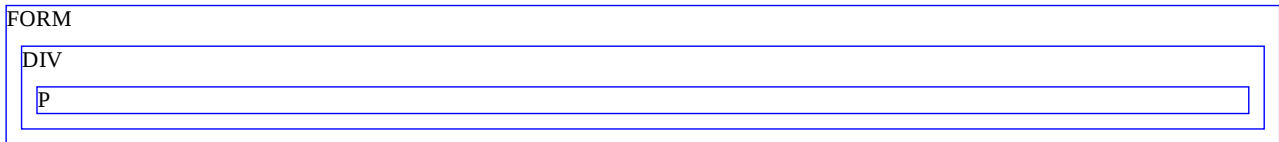
```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>
```

```

<form>FORM
  <div>DIV
    <p>P</p>
  </div>
</form>

<script>
  for(let elem of document.querySelectorAll('*')) {
    elem.addEventListener("click", e => alert(`Captura: ${elem.tagName}`), true);
    elem.addEventListener("click", e => alert(`Propagación: ${elem.tagName}`));
  }
</script>

```



El código prepara manejadores de clic en *cada* elemento en el documento para ver cuáles están funcionando.

Si haces clic en `<p>`, verás que la secuencia es:

1. HTML → BODY → FORM → DIV (fase de captura, el primer detector):
2. P (fase de objetivo, se dispara dos veces, tan pronto como preparemos los dos detectores: de captura y propagación)
3. DIV → FORM → BODY → HTML (fase de propagación, el segundo detector).

Hay una propiedad `event.eventPhase` que nos dice el número de fase en la que el evento fue capturado. Pero es raramente usada, ya que usualmente lo sabemos en el manejador.

i Para quitar el manejador, `removeEventListener` necesita la misma fase

Si nosotros `addEventListener(..., true)`, entonces deberíamos mencionar la misma fase en `removeEventListener(..., true)` para remover el manejador correctamente.

i Detectores de eventos en el mismo elemento y en la misma fase se ejecutan en el orden de asignación

Si tenemos múltiples manejadores de eventos en la misma fase, asignados al mismo elemento con `addEventListener`, se ejecutarán en el orden que fueron creados:

```

elem.addEventListener("click", e => alert(1)); // garantizado que se ejecutará primero
elem.addEventListener("click", e => alert(2));

```

Resumen

Cuando ocurre un evento, el elemento más anidado dónde ocurrió se reconoce como el “elemento objetivo” (`event.target`).

- Luego el evento se mueve hacia abajo desde el documento raíz hacia `event.target`, llamando a los manejadores en el camino asignados con `addEventListener(..., true)` (`true` es una abreviación para `{capture: true}`).
- Luego los manejadores son llamados en el elemento objetivo mismo.
- Luego el evento se propaga desde `event.target` hacia la raíz, llamando a los manejadores que se asignaron usando `on<event>`, atributos HTML y `addEventListener` sin el 3er argumento o con el 3er argumento `false`/`{capture:false}`.

Cada manejador puede acceder a las propiedades del objeto `event`:

- `event.target` – el elemento más profundo que originó el evento.

- `event.currentTarget` (= `this`) – el elemento actual que maneja el evento (el que tiene al manejador en él)
- `event.eventPhase` – la fase actual (captura=1, objetivo=2, propagación=3).

Cualquier manejador de evento puede detener el evento al llamar `event.stopPropagation()`, pero no es recomendado porque no podemos realmente asegurar que no lo necesitaremos más adelante, quizá para completar diferentes cosas.

La fase de captura raramente es usada, usualmente manejamos los evento en propagación. Y hay una lógica atrás de eso.

En el mundo real, cuando un accidente ocurre, las autoridades locales reaccionan primero. Ellos conocen mejor el área dónde ocurrió. Luego, si es necesario, autoridades de alto nivel.

Lo mismo para los manejadores de eventos. El código que se prepara en el manejador de un elemento en particular conoce el máximo de detalles sobre el elemento y qué hace. Un manejador en un `<td>` particular puede ser adecuado para ese exacto `<td>`, conocer todo sobre él, entonces debe tener su oportunidad primero. Luego su padre inmediato también conoce sobre el contexto, pero un poco menos, y así sucesivamente hasta el elemento de arriba que maneja conceptos generales y se ejecuta al final.

La propagación y captura ponen los cimientos para “delegación de eventos”: un extremadamente poderoso patrón de manejo de eventos que se estudia en el siguiente capítulo.

Delegación de eventos

La captura y el propagación nos permiten implementar uno de los más poderosos patrones de manejo de eventos llamado *delegación de eventos*.

La idea es que si tenemos muchos elementos manejados de manera similar podemos, en lugar de asignar un manejador a cada uno de ellos, poner un único manejador a su ancestro común.

En el manejador obtenemos `event.target` para ver dónde ocurrió realmente el evento y manejarlo.

Veamos un ejemplo: El [diagrama Pa kua](#) que refleja la antigua filosofía china.

Aquí está:

Bagua Chart: Direction, Element, Color, Meaning		
Northwest Metal Silver Elders	North Water Blue Change	Northeast Earth Yellow Direction
West Metal Gold Youth	Center All Purple Harmony	East Wood Blue Future
Southwest Earth Brown Tranquility	South Fire Orange Fame	Southeast Wood Green Romance

El HTML es este:

```
<table>
<tr>
  <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning</th>
</tr>
<tr>
  <td class="nw"><strong>Northwest</strong><br>Metal<br>Silver<br>Elders</td>
  <td class="n">...</td>
  <td class="ne">...</td>
</tr>
<tr>...2 more lines of this kind...</tr>
<tr>...2 more lines of this kind...</tr>
</table>
```

La tabla tiene 9 celdas, pero puede haber 99 o 999, eso no importa.

Nuestra tarea es destacar una celda `<td>` al hacer clic en ella.

En lugar de asignar un manejador `onclick` a cada `<td>` (puede haber muchos), configuramos un manejador “atrapa-todo” en el elemento `<table>`.

Este usará `event.target` para obtener el elemento del clic y destacarlo.

El código:

```
let selectedTd;

table.onclick = function(event) {
  let target = event.target; // ¿dónde fue el clic?

  if (target.tagName !== 'TD') return; // ¿no es un TD? No nos interesa

  highlight(target); // destacarlo
};

function highlight(td) {
  if (selectedTd) { // quitar cualquier celda destacada que hubiera antes
    selectedTd.classList.remove('highlight');
  }
  selectedTd = td;
  selectedTd.classList.add('highlight'); // y destacar el nuevo td
}
```

A tal código no le interesa cuántas celdas hay en la tabla. Podemos agregar y quitar `<td>` dinámicamente en cualquier momento y el realzado aún funcionará.

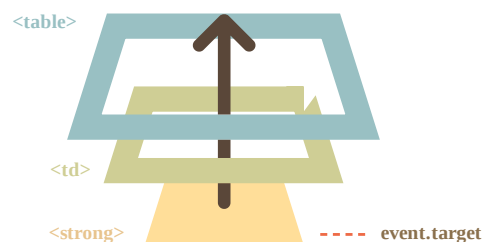
Pero hay un inconveniente.

El clic puede ocurrir no sobre `<td>`, sino dentro de él.

En nuestro caso, si miramos dentro del HTML, podemos ver tags anidados dentro de `<td>`, como ``:

```
<td>
  <strong>Northwest</strong>
  ...
</td>
```

Naturalmente, si el clic ocurre en ``, este se vuelve el valor de `event.target`.



En el manejador `table.onclick` debemos tomar tal `event.target` e indagar si el clic fue dentro de `<td>` o no.

Aquí el código mejorado:

```
table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)
```

```
highlight(td); // (4)
};
```

Explicación:

1. El método `elem.closest(selector)` devuelve el ancestro más cercano que coincide con el selector. En nuestro caso buscamos `<td>` hacia arriba desde el elemento de origen.
2. Si `event.target` no ocurrió dentro de algún `<td>`, el llamado retorna inmediatamente pues no hay nada que hacer.
3. En caso de tablas anidadas, `event.target` podría ser un `<td>` pero fuera de la tabla actual. Entonces verificamos que sea realmente un `<td>` de *nuestra tabla*.
4. Y, si es así, destacarla.

Como resultado, tenemos un código de realzado rápido y eficiente al que no le afecta la cantidad total de `<td>` en la tabla.

Ejemplo de delegación: acciones en markup

Hay otros usos para la delegación de eventos.

Digamos que queremos hacer un menú con los botones “Save”, “Load”, “Search” y así. Y hay objetos con los métodos `save`, `load`, `search` ... ¿Cómo asociarlos?

La primera idea podría ser asignar un controlador separado para cada botón. Pero hay una solución más elegante. Podemos agregar un controlador para el menú completo y un atributo `data-action` a los botones con el método a llamar:

```
<button data-action="save">Click to Save</button>
```

El manejador lee el atributo y ejecuta el método. Puedes ver el siguiente ejemplo en funcionamiento:

```
<div id="menu">
  <button data-action="save">Save</button>
  <button data-action="load">Load</button>
  <button data-action="search">Search</button>
</div>

<script>
  class Menu {
    constructor(elem) {
      this._elem = elem;
      elem.onclick = this.onClick.bind(this); // (*)
    }

    save() {
      alert('saving');
    }

    load() {
      alert('loading');
    }

    search() {
      alert('searching');
    }

    onClick(event) {
      let action = event.target.dataset.action;
      if (action) {
        this[action]();
      }
    }
  };
}
```



```
new Menu(menu);
</script>
```

Save Load Search

Ten en cuenta que `this.onClick` está ligado a `this` en `(*)`. Esto es importante, porque de otra manera el `this` que está dentro haría referencia al elemento DOM (`elem`), no al objeto `Menu`, y `this[action]` no sería lo que necesitamos.

Entonces, ¿qué ventajas nos ofrece la delegación aquí?

- No necesitamos escribir el código para asignar el manejador a cada botón. Simplemente hacer un método y ponerlo en el markup.
- La estructura HTML es flexible, podemos agregar y quitar botones en cualquier momento.

Podríamos usar clases `.action-save`, `.action-load`, pero un atributo `data-action` es mejor semánticamente. Y podemos usarlo con reglas CSS también.

El patrón “comportamiento”

También podemos usar delegación de eventos para agregar “comportamiento” a los elementos de forma *declarativa*, con atributos y clases especiales.

El patrón tiene dos partes:

1. Agregamos un atributo personalizado al elemento que describe su comportamiento.
2. Un manejador rastrea eventos del documento completo, y si un evento ocurre en un elemento con el atributo ejecuta la acción.

Comportamiento: Contador

Por ejemplo, aquí el atributo `data-counter` agrega un comportamiento: “incrementar el valor con un clic” a los botones:

```
Counter: <input type="button" value="1" data-counter>
One more counter: <input type="button" value="2" data-counter>

<script>
  document.addEventListener('click', function(event) {

    if (event.target.dataset.counter !== undefined) { // si el atributo existe...
      event.target.value++;
    }

  });
</script>
```

Counter: 1 One more counter: 2

Si hacemos clic en un botón, su valor se incrementa. Lo importante aquí no son los botones sino el enfoque general.

Puede haber tantos atributos `data-counter` como queramos. Podemos agregar nuevos al HTML en cualquier momento. Usando delegación de eventos “extendimos” el HTML, agregando un atributo que describe un nuevo comportamiento.

⚠ Para manejadores de nivel de documento: siempre `addEventListener`

Cuando asignamos un manejador de evento al objeto `document`, debemos usar siempre `addEventListener`, no `document.on<event>`, porque este último causa conflictos: los manejadores nuevos sobrescribirán los viejos.

En proyectos reales es normal que haya muchos manejadores en `document`, asignados en diferentes partes del código.

Comportamiento: Conmutador (toggle)

Un ejemplo más de comportamiento. Un clic en un elemento con el atributo `data-toggle-id` mostrará/ocultará el elemento con el `id` recibido:

```
<button data-toggle-id="subscribe-mail">
  Show the subscription form
</button>

<form id="subscribe-mail" hidden>
  Your mail: <input type="email">
</form>

<script>
  document.addEventListener('click', function(event) {
    let id = event.target.dataset.toggleId;
    if (!id) return;

    let elem = document.getElementById(id);

    elem.hidden = !elem.hidden;
  });
</script>
```

Show the subscription form

Veamos una vez más lo que hicimos aquí: ahora, para agregar la funcionalidad de conmutación a un elemento, no hay necesidad de conocer JavaScript, simplemente usamos el atributo `data-toggle-id`.

Esto puede ser muy conveniente: no hay necesidad de escribir JavaScript para cada elemento. Simplemente usamos el comportamiento. El manejador a nivel de documento hace el trabajo para cualquier elemento de la página.

Podemos combinar múltiples comportamientos en un único elemento también.

El patrón “comportamiento” puede ser una alternativa a los mini-fragmentos de JavaScript.

Resumen

¡La delegación de eventos es verdaderamente fantástica! Es uno de los patrones más útiles entre los eventos DOM.

A menudo es usado para manejar elementos similares, pero no solamente para eso.

El algoritmo:

1. Pone un único manejador en el contenedor.
2. Dentro del manejador revisa el elemento de origen `event.target`.
3. Si el evento ocurrió dentro de un elemento que nos interesa, maneja el evento.

Beneficios:

- Simplifica la inicialización y ahorra memoria: no hay necesidad de agregar muchos controladores.
- Menos código: cuando agregamos o quitamos elementos, no hay necesidad de agregar y quitar controladores.
- Modificaciones del DOM: podemos agregar y quitar elementos en masa con `innerHTML` y similares.

La delegación tiene sus limitaciones por supuesto:

- Primero, el evento debe “propagarse”. Algunos eventos no lo hacen. Además manejadores de bajo nivel no deben usar `event.stopPropagation()`.
- Segundo, la delegación puede agregar carga a la CPU, porque el controlador a nivel de contenedor reacciona a eventos en cualquier lugar del mismo, no importa si nos interesan o no. Pero usualmente la carga es imperceptible y no la tomamos en cuenta.

✓ Tareas

Ocultar mensajes con delegación

importancia: 5

Hay una lista de mensajes con botones para borrarlos `[x]`. Haz que funcionen.

Como esto:

Horse	[x]
The horse is one of two extant subspecies of <i>Equus ferus</i> . It is an odd-toed ungulate mammal belonging to the taxonomic family Equidae. The horse has evolved over the past 45 to 55 million years from a small multi-toed creature, <i>Eohippus</i> , into the large, single-toed animal of today.	
Donkey	[x]
The donkey or ass (<i>Equus africanus asinus</i>) is a domesticated member of the horse family, Equidae. The wild ancestor of the donkey is the African wild ass, <i>E. africanus</i> . The donkey has been used as a working animal for at least 5000 years.	
Cat	[x]
The domestic cat (Latin: <i>Felis catus</i>) is a small, typically furry, carnivorous mammal. They are often called house cats when kept as indoor pets or simply cats when there is no need to distinguish them from other felids and felines. Cats are often valued by humans for companionship and for their ability to hunt vermin.	

P.D. Debe haber solamente un *event listener* en el contenedor, usa delegación de eventos.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Menú de árbol

importancia: 5

Crea un árbol que muestre y oculte nodos hijos con clics:

- Animals
 - Mammals
 - Cows
 - Donkeys
 - Dogs
 - Tigers
 - Other
 - Snakes
 - Birds
 - Lizards
- Fishes
 - Aquarium
 - Guppy
 - Angelfish
 - Sea
 - Sea trout

Requerimientos:

- Solamente un manejador de eventos (usa delegación)
- Un clic fuera de los nodos de títulos (en un espacio vacío) no debe hacer nada.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Tabla ordenable

importancia: 4

Haz que la tabla se pueda ordenar: los clics en elementos `<th>` deberían ordenarla por la columna correspondiente.

Cada `<th>` tiene su tipo de datos en el atributo, como esto:

```
<table id="grid">
  <thead>
    <tr>
      <th data-type="number">Age</th>
      <th data-type="string">Name</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>5</td>
      <td>John</td>
    </tr>
    <tr>
      <td>10</td>
      <td>Ann</td>
    </tr>
    ...
  </tbody>
</table>
```

En el ejemplo anterior la primera columna tiene números y la segunda cadenas. La función de ordenamiento debe manejar el orden de acuerdo al tipo de dato.

Solamente los tipos `"string"` y `"number"` deben ser soportados.

Ejemplo en funcionamiento:

Age	Name
5	John
2	Pete
12	Ann
9	Eugene
1	Ilya

P.D. La tabla puede ser grande, con cualquier cantidad de filas y columnas.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Comportamiento: Tooltip

importancia: 5

Crea código JS para el comportamiento “tooltip”.

Cuando un mouse pasa sobre un elemento con `data-tooltip`, el tooltip debe aparecer sobre él, y ocultarse cuando se va.

Un ejemplo en HTML comentado:

```
<button data-tooltip="the tooltip is longer than the element">Short button</button>
<button data-tooltip="HTML<br>tooltip">One more button</button>
```

Debe funcionar así:

LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa
LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa

Short button One more button

Scroll the page to make buttons appear on the top, check if the tooltips show up correctly.

En esta tarea suponemos que todos los elementos con `data-tooltip` solo tienen texto dentro. Sin tags anidados (todavía).

Detalles:

- La distancia entre el elemento y el tooltip debe ser `5px`.
- El tooltip debe ser centrado relativo al elemento si es posible.
- El tooltip no debe cruzar los bordes de la ventana. Normalmente debería estar sobre el elemento, pero si el elemento está en la parte superior de la página y no hay espacio para el tooltip, entonces debajo de él.
- El contenido del tooltip está dado en el atributo `data-tooltip`. Este puede ser HTML arbitrario.

Necesitarás dos eventos aquí:

- `mouseover` se dispara cuando el puntero pasa sobre el elemento.
- `mouseout` se dispara cuando el puntero deja el elemento.

Usa delegación de eventos: prepare dos manejadores en el `document` para rastrear todos los “overs” y “outs” de los elementos con `data-tooltip` y administra los tooltips desde allí.

Después de implementar el comportamiento, incluso gente no familiarizada con JavaScript puede agregar elementos anotados.

P.D. Solamente un tooltip puede mostrarse a la vez.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Acciones predeterminadas del navegador

Muchos eventos conducen automáticamente a determinadas acciones realizadas por el navegador.

Por ejemplo:

- Un clic en un enlace: inicia la navegación a su URL.
- Un clic en el botón de envío de un formulario inicia su envío al servidor.
- Al presionar un botón del ratón sobre un texto y moverlo, se selecciona el texto.

Si manejamos un evento en JavaScript, es posible que no queramos que suceda la acción correspondiente del navegador e implementar en cambio otro comportamiento.

Evitar las acciones del navegador

Hay dos formas de decirle al navegador que no queremos que actúe:

- La forma principal es utilizar el objeto `event`. Hay un método `event.preventDefault()`.
- Si el controlador se asigna usando `on<event>` (no por `addEventListener`), entonces devolver `false` también funciona igual.

En este HTML, un clic en un enlace no conduce a la navegación, el navegador no hace nada:

```
<a href="/" onclick="return false">Haz clic aquí</a>
o
<a href="/" onclick="event.preventDefault()">aquí</a>
```

[Haz clic aquí](#) o [aquí](#)

En el siguiente ejemplo usaremos esta técnica para crear un menú basado en JavaScript.

⚠ Regresar `false` desde un controlador es una excepción

El valor devuelto por un controlador de eventos generalmente se ignora.

La única excepción es `return false` de un controlador asignado usando `on<event>`.

En todos los demás casos, se ignora el valor `return`. En particular, no tiene sentido devolver `true`.

Ejemplo: el menú

Considere un menú de sitio, como este:

```
<ul id="menu" class="menu">
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/css">CSS</a></li>
</ul>
```

Así es como se ve con algo de CSS:

HTML

JavaScript

CSS

Los elementos del menú se implementan como enlaces HTML `<a>`, no como botones `<botón>`. Hay varias razones para hacerlo, por ejemplo:

- A muchas personas les gusta usar “clic derecho” – “abrir en una nueva ventana”. Si usamos `<button>` o ``, eso no funciona.
- Los motores de búsqueda siguen los enlaces `` durante la indexación.

Entonces usamos `<a>` en el markup. Pero normalmente pretendemos manejar clics en JavaScript. Por tanto, deberíamos evitar la acción predeterminada del navegador.

Como aquí:

```
menu.onclick = function(event) {
  if (event.target.nodeName !== 'A') return;

  let href = event.target.getAttribute('href');
  alert( href ); // ...se puede cargar desde el servidor, generación de interfaz de usuario, etc.

  return false; // evitar la acción del navegador (no vaya a la URL)
};
```

Si omitimos `return false`, luego de ejecutar nuestro código el navegador realizará su “acción predeterminada”: navegar a la URL en `href`. Y no lo necesitamos aquí, ya que estamos manejando el clic nosotros mismos.

Por cierto, usar la delegación de eventos aquí hace que nuestro menú sea muy flexible. Podemos agregar listas anidadas y diseñarlas usando CSS para “deslizarlas hacia abajo”.

i Eventos de seguimiento

Ciertos eventos fluyen unos a otros. Si evitamos el primer evento, no habrá segundo.

Por ejemplo, `mousedown` en un campo `<input>` conduce a enfocarse en él, y al evento `focus`. Si evitamos el evento `mousedown`, no hay enfoque.

Intenta hacer clic en el primer `<input>` a continuación: se produce el evento `focus`. Pero si haces clic en el segundo, no hay enfoque.

```
<input value="Enfoque funciona" onfocus="this.value=' '">
<input onmousedown="return false" onfocus="this.value=' '" value="Haz clic en mí">
```

Enfoque funciona Haz clic en mí

Eso es porque la acción del navegador se cancela en `mousedown`. El enfoque aún es posible si usamos otra forma de ingresar la entrada. Por ejemplo, la tecla `Tab` para cambiar de la primera entrada a la segunda. Pero ya no con el clic del ratón.

La opción de controlador “pasivo”

La opción opcional `passive:true` de `addEventListener` indica al navegador que el controlador no llamará a `preventDefault()`.

¿Por qué puede ser necesario?

Hay algunos eventos como `touchmove` en dispositivos móviles (cuando el usuario mueve el dedo por la pantalla), que provocan el desplazamiento por defecto, pero ese desplazamiento se puede evitar usando `preventDefault()` en el controlador.

Entonces, cuando el navegador detecta tal evento, primero tiene que procesar todos los controladores, y luego, si no se llama a `preventDefault` en ninguna parte, puede continuar con el desplazamiento. Eso puede causar retrasos innecesarios y “movimientos de salto repentinos” en la interfaz de usuario.

Las opciones `passive: true` le dicen al navegador que el controlador no va a cancelar el desplazamiento. Entonces el navegador se desliza de inmediato para brindar una experiencia con la máxima fluidez, y el evento se maneja de inmediato.

Para algunos navegadores (Firefox, Chrome), `passive` es `true` por defecto para los eventos `touchstart` y `touchmove`.

event.defaultPrevented

La propiedad `event.defaultPrevented` es `true` si se impidió la acción predeterminada y `false` en caso contrario.

Hay un caso de uso interesante para ello.

¿Recuerdas que en el capítulo [Propagación y captura](#) hablamos sobre `event.stopPropagation()` y por qué detener propagación es malo?

A veces podemos usar `event.defaultPrevented` en su lugar, para señalar a otros controladores de eventos que el evento fue manejado.

Veamos un ejemplo práctico.

Por defecto, el navegador en el evento `contextmenu` (clic derecho del ratón) muestra un menú contextual con opciones estándar. Podemos prevenirlo y mostrar el nuestro, así:

```
<button>El clic derecho muestra el menú contextual del navegador</button>

<button oncontextmenu="alert('Dibuja nuestro menú'); return false">
  El clic derecho muestra nuestro menú contextual
</button>
```

El clic derecho muestra el menú contextual del navegador El clic derecho muestra nuestro menú contextual

Ahora, además de ese menú contextual, nos gustaría implementar un menú contextual para todo el documento.

Al hacer clic derecho, debería aparecer el menú contextual más cercano.

```
<p>Haz clic derecho aquí para el menú contextual del documento</p>
<button id="elem">Haz clic derecho aquí para el menú contextual del botón</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Menú contextual del botón");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Menú contextual del documento");
  };
</script>
```

Haz clic derecho aquí para el menú contextual del documento

Haz clic derecho aquí para el menú contextual del botón

El problema es que cuando hacemos clic en `elem`, obtenemos dos menús: el de nivel de botón y (el evento emerge) el menú de nivel de documento.

¿Como arreglarlo? Una de las soluciones es pensar así: “Cuando hagamos clic con el botón derecho en el controlador de botones, detengamos su propagación” y usemos `event.stopPropagation()`:

```
<p>Haz clic derecho para el menú del documento</p>
<button id="elem">Haz clic derecho para el menú del botón (arreglado con event.stopPropagation)</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    event.stopPropagation();
  };
</script>
```



```

    alert("Menú contextual del botón");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Menú contextual del documento");
  };
</script>

```

Haz clic derecho para el menú del documento

Haz clic derecho para el menú del botón (arreglado con event.stopPropagation)

Ahora, el menú de nivel de botón funciona según lo previsto. Pero el precio es alto. Siempre negamos el acceso a la información sobre los clics con el botón derecho del ratón para cualquier código externo, incluidos los contadores que recopilan estadísticas, etc. Eso es bastante imprudente.

¿Sería una solución alternativa verificar en el controlador `document` si se evitó la acción predeterminada? Si es así, entonces se manejó el evento y no necesitamos reaccionar ante él.

```

<p>Haz clic con el botón derecho en el menú del documento (se agregó una marca de verificación para event.defaultPrevented)</p>
<button id="elem">Haz clic derecho para el menú de botones</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Menú contextual del botón");
  };

  document.oncontextmenu = function(event) {
    if (event.defaultPrevented) return;

    event.preventDefault();
    alert("Menú contextual del documento");
  };
</script>

```

Haz clic con el botón derecho en el menú del documento (se agregó una marca de verificación para event.defaultPrevented)

Haz clic derecho para el menú de botones

Ahora todo también funciona correctamente. Si tenemos elementos anidados, y cada uno de ellos tiene un menú contextual propio, eso también funcionaría. Solo asegúrate de buscar `event.defaultPrevented` en cada controlador de `contextmenu`.

i event.stopPropagation() y event.preventDefault()

Como podemos ver claramente, `event.stopPropagation()` y `event.preventDefault()` (también conocido como `return false`) son dos cosas diferentes. No están relacionados entre sí.

i Arquitectura de menús contextuales anidados

También hay formas alternativas de implementar menús contextuales anidados. Uno de ellos es tener un único objeto global con un manejador para `document.oncontextmenu`, y también métodos que nos permitan almacenar otros manejadores en él.

El objeto detectará cualquier clic derecho, examinará los controladores almacenados y ejecutará el apropiado.

Pero entonces cada fragmento de código que quiera un menú contextual debe conocer ese objeto y usar su ayuda en lugar del propio controlador `contextmenu`.

Resumen

Hay muchas acciones predeterminadas del navegador:

- `mousedown` – inicia la selección (mueva el ratón para seleccionar).
- `click` en `<input type="checkbox">` – marca/desmarca el `input`.
- `submit` – dar clic en `<input type="submit">` o presionar `Enter` dentro de un campo de formulario hace que suceda este evento y el navegador envía el formulario a continuación.
- `keydown` – presionar una tecla puede llevar a agregar un carácter a un campo u otras acciones.
- `contextmenu` – el evento ocurre con un clic derecho, la acción es mostrar el menú contextual del navegador.
- ...hay mas...

Todas las acciones predeterminadas se pueden evitar si queremos manejar el evento exclusivamente mediante JavaScript.

Para evitar una acción predeterminada, utiliza `event.preventDefault()` o `return false`. El segundo método funciona solo para los controladores asignados con `on<event>`.

La opción `passive: true` de `addEventListener` le dice al navegador que la acción no se evitará. Eso es útil para algunos eventos móviles, como `touchstart` y `touchmove`, para decirle al navegador que no debe esperar a que todos los controladores terminen antes de desplazarse.

Si se evitó la acción predeterminada, el valor de `event.defaultPrevented` se convierte en `true`, de lo contrario, es `false`.

⚠ Mantente semántico, no abuses

Técnicamente, al evitar acciones predeterminadas y agregar JavaScript, podemos personalizar el comportamiento de cualquier elemento. Por ejemplo, podemos hacer que un enlace `<a>` funcione como un botón, y un botón `<button>` se comporte como un enlace (redirigir a otra URL o algo así).

Pero en general deberíamos mantener el significado semántico de los elementos HTML. Por ejemplo, la navegación debe realizarla `<a>`, no un botón.

Además de ser “algo bueno”, hace que su HTML sea mejor en términos de accesibilidad.

Además, si consideramos el ejemplo con `<a>`, ten en cuenta: un navegador nos permite abrir dichos enlaces en una nueva ventana (usando el botón derecho u otros medios). Y a la gente le gusta. Pero si hacemos que un botón se comporte como un enlace usando JavaScript e incluso parezca un enlace usando CSS, las características específicas de `<a>` no funcionarán en él.

✅ Tareas

¿Por qué "return false" no funciona?

importancia: 3

¿Por qué en el código de abajo `return false` no funciona en absoluto?

```
<script>
function handler() {
  alert( "... " );
  return false;
}
</script>

<a href="https://w3.org" onclick="handler()">el navegador irá a w3.org</a>
```

[el navegador irá a w3.org](https://w3.org)

El navegador sigue la URL al hacer clic, pero no la queremos.

¿Como se arregla?

[A solución](#)

Captura enlaces en el elemento

importancia: 5

Haz que todos los enlaces dentro del elemento con `id="contents"` pregunten al usuario si realmente quiere irse. Y si no quiere, no sigas.

Así:

#contents

¿Que tal si leemos [Wikipedia](#) o visitamos [W3.org](#) y aprendemos sobre los estándares modernos?

Detalles:

- El HTML dentro del elemento puede cargarse o regenerarse dinámicamente en cualquier momento, por lo que no podemos encontrar todos los enlaces y ponerles controladores. Utilice la delegación de eventos.
- El contenido puede tener etiquetas anidadas. Dentro de los enlaces también, como `<i>...</i>`.

[Abrir un entorno controlado para la tarea.](#)

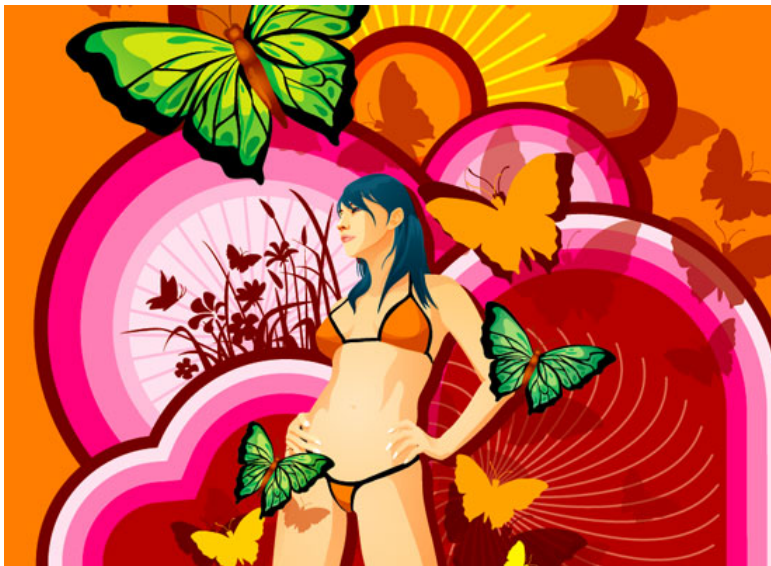
[A solución](#)

Galería de imágenes

importancia: 5

Crea una galería de imágenes donde la imagen principal cambia al hacer clic en una miniatura.

Así:



P.D. Utiliza la delegación de eventos.

[Abrir un entorno controlado para la tarea.](#)

Envío de eventos personalizados

No solo podemos asignar controladores, sino también generar eventos desde JavaScript.

Los eventos personalizados se pueden utilizar para crear “componentes gráficos”. Por ejemplo, un elemento raíz de nuestro propio menú basado en JS puede desencadenar eventos que indiquen lo que sucede con el menú: `abrir` (menú abierto), `seleccionar` (se selecciona un elemento) y así sucesivamente. Otro código puede escuchar los eventos y observar lo que sucede con el menú.

No solo podemos generar eventos completamente nuevos, que inventamos para nuestros propios fines, sino también eventos integrados, como `click`, `mousedown`, etc. Eso puede ser útil para las pruebas automatizadas.

Constructor de eventos

Las clases de eventos integradas forman una jerarquía, similar a las clases de elementos DOM. La raíz es la clase incorporada `Event` [↗](#).

Podemos crear objetos `Event` así:

```
let event = new Event(type[, options]);
```

Argumentos:

- *type* – tipo de event, un string como `"click"` o nuestro propio evento como `"mi-evento"`.
- *options* – el objeto con 2 propiedades opcionales:
 - `bubbles: true/false` – si es `true`, entonces el evento se propaga.
 - `cancelable: true/false` – si es `true`, entonces la “acción predeterminada” puede ser prevenida. Más adelante veremos qué significa para los eventos personalizados.

Por defecto, los dos son false: `{bubbles: false, cancelable: false}`.

dispatchEvent

Después de que se crea un objeto de evento, debemos “ejecutarlo” en un elemento usando la llamada `elem.dispatchEvent(event)`.

Luego, los controladores reaccionan como si fuera un evento normal del navegador. Si el evento fue creado con la bandera `bubbles`, entonces se propaga.

En el siguiente ejemplo, el evento `click` se inicia en JavaScript. El controlador funciona de la misma manera que si se hiciera clic en el botón:

```
<button id="elem" onclick="alert('Clic!');">Click automático</button>

<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

event.isTrusted

Hay una forma de diferenciar un evento de usuario “real” de uno generado por script.

La propiedad `event.isTrusted` es `true` para eventos que provienen de acciones de usuarios reales y `false` para eventos generados por script.

Ejemplo de Bubbling

Podemos crear un evento bubbling con el nombre "hello" y capturarlo en document .

Todo lo que necesitamos es establecer bubbles en true :

```
<h1 id="elem">Hola desde el script!</h1>

<script>
  // Captura en document...
  document.addEventListener("hello", function(event) { // (1)
    alert("Hola desde " + event.target.tagName); // Hola desde H1
  });

  // ...Envío en elem!
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);

  // el controlador del documento se activará y mostrará el mensaje.

</script>
```

Notas:

1. Debemos usar addEventListener para nuestros eventos personalizados, porque on<event> solo existe para eventos incorporados, document.onhello no funciona.
2. Debes poner bubbles: true , de otra manera el evento no se propagará.

La mecánica de bubbling es la misma para los eventos integrados (click) y personalizados (hello). También hay etapas de captura y propagación.

MouseEvent, KeyboardEvent y otros

Aquí hay una breve lista de clases para eventos UI (interfaz de usuario) de la [especificación de eventos UI](#) ↗ :

- UIEvent
- FocusEvent
- MouseEvent
- WheelEvent
- KeyboardEvent
- ...

Deberíamos usarlos en lugar de new Event si queremos crear tales eventos. Por ejemplo, new MouseEvent("click").

El constructor correcto permite especificar propiedades estándar para ese tipo de evento.

Como clientX/clientY para un evento de mouse:

```
let event = new MouseEvent("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // 100
```

Tenga en cuenta: el constructor genérico Event no lo permite.

Intentemos:

```
let event = new Event("click", {
  bubbles: true, // solo bubbles y cancelable
  cancelable: true, // funcionan en el constructor de Event
```

```

    clientX: 100,
    clientY: 100
  });

  alert(event.clientX); // undefined, se ignora la propiedad desconocida!

```

Técnicamente, podemos solucionarlo asignando directamente `event.clientX=100` después de la creación. Entonces eso es una cuestión de conveniencia y de seguir las reglas. Los eventos generados por el navegador siempre tienen el tipo correcto.

La lista completa de propiedades para diferentes eventos de UI se encuentra en la especificación, por ejemplo, [MouseEvent](#) .

Eventos personalizados

Para nuestros tipos de eventos completamente nuevos, como "hello", deberíamos usar `new CustomEvent` . Técnicamente, [CustomEvent](#) es lo mismo que `Event` , con una excepción.

En el segundo argumento (objeto) podemos agregar una propiedad adicional `detail` para cualquier información personalizada que queramos pasar con el evento.

Por ejemplo:

```

<h1 id="elem">Hola para John!</h1>

<script>
  // detalles adicionales que vienen con el evento para el controlador.
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  });

  elem.dispatchEvent(new CustomEvent("hello", {
    detail: { name: "John" }
  }));
</script>

```

La propiedad `detail` puede tener cualquier dato. Técnicamente podríamos vivir sin él, porque podemos asignar cualquier propiedad a un objeto `new Event` regular después de su creación. Pero `CustomEvent` proporciona el campo especial `detail` para evitar conflictos con otras propiedades del evento.

Además, la clase de evento describe "qué tipo de evento" es, y si el evento es personalizado, entonces deberíamos usar `CustomEvent` solo para tener claro qué es.

event.preventDefault()

Muchos eventos del navegador tienen una "acción predeterminada", como ir a un enlace, iniciar una selección, etc.

Para eventos nuevos y personalizados, definitivamente no hay acciones predeterminadas del navegador, pero un código que distribuye dicho evento puede tener sus propios planes de qué hacer después de activar el evento.

Al llamar a `event.preventDefault()` , un controlador de eventos puede enviar una señal de que esas acciones deben cancelarse.

En ese caso, la llamada a `elem.dispatchEvent(event)` devuelve `false` . Y el código que lo envió sabe que no debería continuar.

Veamos un ejemplo práctico: un conejo escondido (podría ser un menú de cierre u otra cosa).

A continuación puede ver una función `#rabbit` y `hide()` que distribuye el evento "hide" en él, para que todas las partes interesadas sepan que el conejo se va a esconder.

Cualquier controlador puede escuchar ese evento con `rabbit.addEventListener('hide', ...)` y, si es necesario, cancelar la acción usando `event.preventDefault()` . Entonces el conejo no desaparecerá:

```

<pre id="rabbit">

```

```

| \  _/ |
 \ | _ | /
  / . . \
= \ _ Y _ / =
 {>0<}

</pre>
<button onclick="hide()">Esconder()</button>

<script>
// hide() será llamado automáticamente en 2 segundos.
function hide() {
  let event = new CustomEvent("hide", {
    cancelable: true // sin esa bandera preventDefault no funciona
  });
  if (!rabbit.dispatchEvent(event)) {
    alert('La acción fue impedida por un controlador');
  } else {
    rabbit.hidden = true;
  }
}

rabbit.addEventListener('hide', function(event) {
  if (confirm("¿Llamar a preventDefault?")) {
    event.preventDefault();
  }
});
</script>

```

```

| \  _/ |
 \ | _ | /
  / . . \
= \ _ Y _ / =
 {>0<}

```

Esconder()

Tenga en cuenta: el evento debe tener la bandera `cancelable: true`, de lo contrario, la llamada `event.preventDefault()` se ignora.

Los eventos dentro de eventos son sincrónicos

Por lo general, los eventos se procesan en una cola. Es decir: si el navegador está procesando `onclick` y ocurre un nuevo evento, por ejemplo, con el mouse movido, luego su despacho se pone en cola, los controladores correspondientes `mousemove` serán llamados después de que el procesamiento de `onclick` haya terminado.

La excepción notable es cuando un evento se inicia desde dentro de otro, por ejemplo, usando `dispatchEvent`. Dichos eventos se procesan inmediatamente: se llaman los nuevos controladores de eventos y luego se reanuda el manejo de eventos actual.

Por ejemplo, en el código siguiente, el evento `menu-open` se activa durante el `onclick`.

Se procesa inmediatamente, sin esperar a que termine el controlador `onclick`:

```

<button id="menu">Menu (dame clic)</button>

<script>
menu.onclick = function() {
  alert(1);

  menu.dispatchEvent(new CustomEvent("menu-open", {
    bubbles: true
  }));

  alert(2);
};

// se dispara entre 1 y 2
document.addEventListener('menu-open', () => alert('anidado'));
</script>

```

Menu (dame clic)

El orden de salida es: 1 → anidado → 2.

Tenga en cuenta que el evento anidado `menu-open` se captura en `document`. La propagación y el manejo del evento anidado finaliza antes de que el procesamiento vuelva al código externo (`onclick`).

No se trata solo de `dispatchEvent`, hay otros casos. Si un controlador de eventos llama a métodos que desencadenan otros eventos, también se procesan sincrónicamente, de forma anidada.

Supongamos que no nos gusta. Querríamos que `onclick` se procesara por completo primero, independientemente de `menu-open` o cualquier otro evento anidado.

Entonces podemos poner el `dispatchEvent` (u otra llamada de activación de eventos) al final de `onclick` o, mejor aún, envolverlo en el `setTimeout` de retardo cero:

```
<button id="menu">Menu (dame clic)</button>

<script>
  menu.onclick = function() {
    alert(1);

    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('anidado'));
</script>
```

Ahora `dispatchEvent` se ejecuta asincrónicamente después de que la ejecución del código actual finaliza, incluyendo `menu.onclick`. Los controladores de eventos están totalmente separados.

El orden de salida se convierte en: 1 → 2 → anidado.

Resumen

Para generar un evento a partir del código, primero necesitamos crear un objeto de evento.

El constructor genérico `Event(name, options)` acepta un nombre de evento arbitrario y el objeto `options` con dos propiedades:

- `bubbles: true` si el evento debe propagarse.
- `cancelable: true` si `event.preventDefault()` debe funcionar.

Otros constructores de eventos nativos como `MouseEvent`, `KeyboardEvent`, y similares, aceptan propiedades específicas para ese tipo de evento. Por ejemplo, `clientX` para eventos de mouse.

Para eventos personalizados deberíamos usar el constructor `CustomEvent`. Este tiene una opción adicional llamada `detail` a la que podemos asignarle los datos específicos del evento. De esta forma, todos los controladores pueden accederlos como `event.detail`.

A pesar de la posibilidad técnica de generar eventos del navegador como `click` o `keydown`, debemos usarlo con mucho cuidado.

No deberíamos generar eventos de navegador, ya que es una forma trillada de ejecutar controladores. Esa es una mala arquitectura la mayor parte del tiempo.

Se pueden generar eventos nativos:

- Como un truco sucio para hacer que las bibliotecas de terceros funcionen de la manera necesaria, si es que ellas no proporcionan otros medios de interacción.
- Para pruebas automatizadas, que el script “haga clic en el botón” y vea si la interfaz reacciona correctamente.

Los eventos personalizados con nuestros propios nombres a menudo se generan con fines arquitectónicos, para señalar lo que sucede dentro de nuestros menús, controles deslizantes, carruseles, etc.

Eventos en la UI

Aquí cubriremos los eventos más importantes de la interfaz de usuario y cómo podemos trabajar con ellos.

Eventos del Mouse

En este capítulo vamos a entrar en más detalles sobre los eventos del mouse y sus propiedades.

Ten en cuenta que tales eventos pueden provenir no sólo del “dispositivo mouse”, sino también de otros dispositivos, como teléfonos y tabletas, donde se emulan por compatibilidad.

Tipos de eventos del mouse

Ya hemos visto algunos de estos eventos:

`mousedown/mouseup`

Se oprime/suelta el botón del ratón sobre un elemento.

`mouseover/mouseout`

El puntero del mouse se mueve sobre/sale de un elemento.

`mousemove`

Cualquier movimiento del mouse sobre un elemento activa el evento.

`click`

Se activa después de `mousedown` y un `mouseup` enseguida sobre el mismo elemento si se usó el botón.

`dblclick`

Se activa después de dos clicks seguidos sobre el mismo elemento. Hoy en día se usa raramente.

`contextmenu`

Se activa al pulsar el botón derecho del ratón. Existen otras formas de abrir el menú contextual, por ejemplo: usando un comando especial de teclado también puede activarse, de manera que no es exactamente un evento exclusivo del mouse.

...Existen otros eventos más que cubriremos más tarde.

El orden de los eventos

Como pudiste ver en la lista anterior, una acción del usuario puede desencadenar varios eventos.

Por ejemplo, un click izquierdo primero activa `mousedown` cuando se presiona el botón, enseguida `mouseup` y `click` cuando se suelta.

En casos así, el orden es fijo. Es decir, los controladores son llamados en el siguiente orden `mousedown` → `mouseup` → `click`.

El botón del mouse

Los eventos relacionados con clics siempre tienen la propiedad `button`, esta nos permite conocer el botón exacto del mouse.

Normalmente no la usamos para eventos `click` y `contextmenu` events, porque sabemos que ocurren solo con click izquierdo y derecho respectivamente.

Por otro lado, los controladores `mousedown` y `mouseup` pueden necesitar `event.button` ya que estos eventos se activan con cualquier botón y `button` nos permitirá distinguir entre “mousedown derecho” y “mousedown izquierdo”.

Los valores posibles para `event.button` son:

Estado del botón	<code>event.button</code>
Botón izquierdo (primario)	0
Botón central (auxiliar)	1
Botón derecho (secundario)	2
Botón X1 (atrás)	3
Botón X2 (adelante)	4

La mayoría de los dispositivos de ratón sólo tienen los botones izquierdo y derecho, por lo que los valores posibles son `0` o `2`. Los dispositivos táctiles también generan eventos similares cuando se toca sobre ellos.

También hay una propiedad `event.buttons` que guarda todos los botones presionados actuales en un solo entero, un bit por botón. En la práctica, esta propiedad es raramente utilizada. Puedes encontrar más detalles en [MDN](#) si alguna vez lo necesitas.

⚠ El obsoleto `event.which`

El código puede utilizar la propiedad `event.which` que es una forma antigua no estándar de obtener un botón con los valores posibles:

- `event.which == 1` – botón izquierdo,
- `event.which == 2` – botón central,
- `event.which == 3` – botón derecho.

Ahora `event.which` está en desuso, no deberíamos usarlo.

Modificadores: shift, alt, ctrl y meta

Todos los eventos del mouse incluyen la información sobre las teclas modificadoras presionadas.

Propiedades del evento:

- `shiftKey`: `Shift`
- `altKey`: `Alt` (p `Opt` para Mac)
- `ctrlKey`: `Ctrl`
- `metaKey`: `Cmd` para Mac

Su valor es `true` si la tecla fue presionada durante el evento.

Por ejemplo, el botón abajo solo funciona con `Alt+Shift+click`:

```
<button id="button">Alt+Shift+¡Click sobre mí!</button>
```

```
<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('¡Genial!');
    }
  };
</script>
```

Alt+Shift+¡Click sobre mí!

⚠ Atención: en Mac suele ser `Cmd` en lugar de `Ctrl`

En Windows y Linux existen las teclas modificadoras `Alt`, `Shift` y `Ctrl`. En Mac hay una más: `Cmd`, correspondiente a la propiedad `metaKey`.

En la mayoría de las aplicaciones, cuando Windows/Linux usan `Ctrl`, en Mac se usa `Cmd`.

Es decir: cuando un usuario de Windows usa `Ctrl+Enter` o `Ctrl+A`, un usuario Mac presionaría `Cmd+Enter` o `Cmd+A`, y así sucesivamente.

Entonces si queremos darle soporte a combinaciones como `Ctrl+click`, entonces para Mac tendría más sentido usar `Cmd+click`. Esto es más cómodo para los usuarios de Mac.

Incluso si quisiéramos obligar a los usuarios de Mac a hacer `Ctrl+click` – esto supone algo de dificultad. El problema es que: un click izquierdo con `Ctrl` es interpretado como *click derecho* en MacOS, y esto genera un evento `contextmenu`, no un `click` como en Windows/Linux.

Así que si queremos que los usuarios de todos los sistemas operativos se sientan cómodos, entonces junto con `ctrlKey` debemos verificar `metaKey`.

Para código JS significa que debemos hacer la comprobación `if (event.ctrlKey || event.metaKey)`.

⚠ También hay dispositivos móviles

Las combinaciones de teclado son buenas como una adición al flujo de trabajo. De modo que si el visitante usa un teclado – funcionan.

Pero si su dispositivo no lo tiene – entonces debería haber una manera de vivir sin teclas modificadoras.

Coordenadas: `clientX/Y`, `pageX/Y`

Todos los eventos del ratón proporcionan coordenadas en dos sabores:

1. Relativas a la ventana: `clientX` y `clientY`.
2. Relativos al documento: `pageX` y `pageY`.

Ya cubrimos la diferencia entre ellos en el capítulo [Coordenadas](#).

En resumen, las coordenadas relativas al documento `pageX/Y` se cuentan desde la esquina superior izquierda del documento y no cambian cuando se desplaza la página, mientras que `clientX/Y` se cuentan desde la esquina superior actual. Cambian cuando se desplaza la página.

Por ejemplo, si tenemos una ventana del tamaño 500x500, y el mouse está en la esquina superior izquierda, entonces `clientX` y `clientY` son `0`, sin importar cómo se desplace la página.

Y si el mouse está en el centro, entonces `clientX` y `clientY` son `250`, No importa en qué parte del documento se encuentren. Esto es similar a `position:fixed` en ese aspecto.

Previniendo la selección en `mousedown`

El doble clic del mouse tiene un efecto secundario que puede ser molesto en algunas interfaces: selecciona texto.

Por ejemplo, un doble clic en el texto de abajo lo selecciona además de activar nuestro controlador:

```
<span onclick="alert('dblclick')">Haz doble click en mi</span>
```

Haz doble click en mi

Si se pulsa el botón izquierdo del ratón y, sin soltarlo, mueve el ratón, también hace la selección, a menudo no deseado.

Hay varias maneras de evitar la selección, que se pueden leer en el capítulo [Selection y Range](#).

En este caso particular, la forma más razonable es evitar la acción del navegador `mousedown` . Esto evita ambas selecciones:

Antes...

```
<b onclick="alert('Click!')" onmousedown="return false">  
  Haz doble click en mí  
</b>  
...Después
```

Antes... **Haz doble click en mí** ...Después

Ahora el elemento en negrita no se selecciona con doble clic, y al mantener presionado el botón izquierdo y arrastrar no se iniciará la selección.

Tenga en cuenta: el texto dentro de él todavía es seleccionable. Sin embargo, la selección no debe comenzar en el texto en sí, sino antes o después. Por lo general, eso está bien para los usuarios.

Previendo copias

Si queremos inhabilitar la selección para proteger nuestro contenido de la página del copy-paste, entonces podemos utilizar otro evento: `oncopy` .

```
<div oncopy="alert(';Copiado prohibido!');return false">  
  Querido usuario,  
  El copiado está prohibida para ti.  
  Si sabes JS o HTML entonces puedes obtener todo de la fuente de la página.  
</div>
```

Querido usuario, El copiado está prohibida para ti. Si sabes JS o HTML entonces puedes obtener todo de la fuente de la página.

Si intenta copiar un fragmento de texto en el `<div>` no va a funcionar porque la acción default de `oncopy` fue evitada.

Seguramente el usuario tiene acceso a la fuente HTML de la página, y puede tomar el contenido desde allí, pero no todos saben cómo hacerlo.

Resumen

Los eventos del mouse tienen las siguientes propiedades:

- Botón: `button` .
- Teclas modificadoras (`true` si fueron presionadas): `altKey` , `ctrlKey` , `shiftKey` y `metaKey` (Mac).
 - Si quieres controlar las acciones de la tecla `Ctrl` no te olvides de los usuarios de Mac que generalmente usan `Cmd` , de manera que es mejor verificar con la condicional: `if (e.metaKey || e.ctrlKey)` .
- Coordenadas relativas a la ventana: `clientX/clientY` .
- Coordenadas relativas al documento: `pageX/pageY` .

La acción predeterminada del navegador `mousedown` es la selección del texto, si no es bueno para la interfaz, entonces debe evitarse.

En el próximo capítulo veremos más detalles sobre los eventos que siguen al movimiento del puntero y cómo rastrear los cambios de elementos debajo de él.

Tareas

Lista seleccionable

importancia: 5

Cree una lista donde los elementos son seleccionables, como en los administradores de archivos.

- Un clic en un elemento de la lista selecciona solo ese elemento (agrega la clase `.selected`), deselecta todos los demás.
- Si se hace un clic con `Ctrl` (`Cmd` para Mac), el estado seleccionado/deseleccionado cambia para ese solo elemento, los otros elementos no se modifican.

Demo:

Haz click en un elemento de la lista para seleccionarlo

- Christopher Robin
- Winnie-the-Pooh
- Tigger
- Kanga
- Rabbit. Just rabbit.

PD: Para esta tarea, podemos suponer que los elementos de la lista son solo de texto. No hay etiquetas anidadas.

PPD: Evita la selección nativa del navegador del texto en los clics.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Moviendo el mouse: `mouseover/out`, `mouseenter/leave`

Entremos en detalle sobre los eventos que suceden cuando el mouse se mueve entre elementos.

Eventos `mouseover/mouseout`, `relatedTarget`

El evento `mouseover` se produce cuando el cursor del mouse aparece sobre un elemento y `mouseout` cuando se va.



Estos eventos son especiales porque tienen la propiedad `relatedTarget`. Esta propiedad complementa a `target`. Cuando el puntero del mouse deja un elemento por otro, uno de ellos se convierte en `target` y el otro en `relatedTarget`.

Para `mouseover`:

- `event.target` – Es el elemento al que se acerca el mouse.
- `event.relatedTarget` – Es el elemento de donde proviene el mouse (`relatedTarget` → `target`).

Para `mouseout` sucede al contrario:

- `event.target` – Es el elemento que el mouse dejó.
- `event.relatedTarget` – es el nuevo elemento bajo el cursor por cuál el cursor dejó al anterior (`target` → `relatedTarget`).

⚠️ `relatedTarget` puede ser `null`

La propiedad `relatedTarget` puede tener un valor `null`.

Eso es normal y solo significa que el mouse no vino de otro elemento, sino de la ventana o que salió de la ventana.

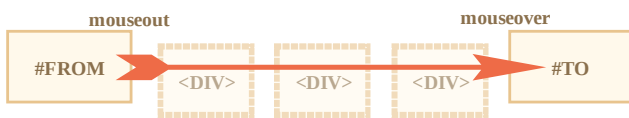
Debemos tener en cuenta esa posibilidad cuando usemos `event.relatedTarget` en nuestro código. Si accedemos a `event.relatedTarget.tagName` entonces habrá un error.

Saltando elementos

El evento `mousemove` se activa cuando el mouse se mueve pero eso no significa que cada píxel nos lleve a un evento.

El navegador verifica la posición del mouse de vez en cuando y si nota cambios entonces activan los eventos.

Eso significa que si el visitante mueve el mouse muy rápido, entonces algunos elementos del DOM podrían estar siendo ignorados:

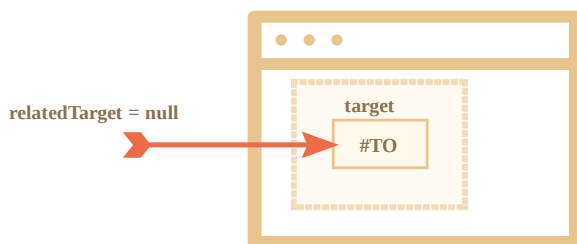


Si el mouse se mueve muy rápido de los elementos `#FROM` a `#TO`, como se muestra arriba, entonces los elementos intermedios `<div>` (o algunos de ellos) podrían ser ignorados. El evento `mouseout` se podría activar en `#FROM` e inmediatamente `mouseover` en `#TO`.

Eso es bueno para el rendimiento porque puede haber muchos elementos intermedios. Realmente no queremos procesar todo lo que sucede dentro y fuera de cada uno.

Por otro lado, debemos tener en cuenta que el puntero del mouse no “visita” todos los elementos en el camino. Los puede “saltar”.

En particular, es posible que el puntero salte dentro de la mitad de la página desde la ventana. En ese caso `relatedTarget` es `null`, porque vino de “la nada”:



i Si `mouseover` se activa, deberá haber `mouseout`

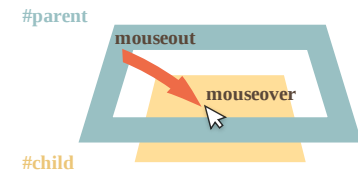
En caso de movimientos rápidos, los elementos intermedios podrían ser ignorados, pero una cosa segura sabemos: si el cursor ingresa “oficialmente” dentro de un elemento (evento `mouseover` generado), una vez que lo deje obtendremos `mouseout`.

Mouseout, cuando se deja un elemento por uno anidado.

Una característica importante de `mouseout` – se activa cuando el cursor se mueve de un elemento hacia su descendiente (elemento anidado o interno). Por ejemplo de `#parent` a `#child` en este HTML:

```
<div id="parent">
  <div id="child">...</div>
</div>
```

Si estamos sobre `#parent` y luego movemos el cursor hacia dentro de `#child`, ¡vamos a obtener `mouseout` en `#parent`!



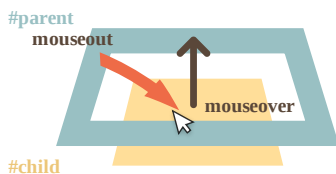
Eso puede parecer extraño, pero puede explicarse fácilmente.

De acuerdo con la lógica del navegador, el cursor podría estar sobre un elemento *individual* en cualquier momento – el anidado y el más alto según el z-index.

Entonces si se dirige hacia otro elemento (incluso uno anidado), está dejando al anterior.

Por favor, note otro importante detalle sobre el procesamiento de eventos.

El evento `mouseover` se aparece en un un elemento anidado (brota o nace, por decirlo así). Entonces si `#parent` tiene el controlador `mouseover`, se activa:



Como se muestra, cuando el cursor se mueve del elemento `#parent` a `#child`, los dos controladores se activan en el elemento padre: `mouseout` y `mouseover`:

```
parent.onmouseout = function(event) {  
  /* event.target: elemento padre */  
};  
parent.onmouseover = function(event) {  
  /* event.target: elemento hijo (brota) */  
};
```

Si no examinamos `event.target` dentro de los controladores podría parecer que el cursor dejó el elemento `#parent` y volvió a él inmediatamente.

Pero ese no es el caso. El cursor aún está sobre el elemento padre, simplemente se adentró más en el elemento hijo.

Si hay algunas acciones al abandonar el elemento padre, por ejemplo: una animación se ejecuta con `parent.onmouseout`, usualmente no la queremos cuando el cursor se adentre más sobre `#parent`.

Para evitar esto lo que podemos hacer es checar `relatedTarget` en el controlador y si el mouse aún permanece dentro del elemento entonces ignorar dicho evento.

Alternativamente podemos usar otros eventos: `mouseenter` y `mouseleave`, los cuales cubriremos a continuación, ya que con ellos no hay tales problemas.

Eventos `mouseenter` y `mouseleave`

Los eventos `mouseenter/mouseleave` son como `mouseover/mouseout`. Se activan cuando el cursor del mouse entra/sale del elemento.

Pero hay dos diferencias importantes:

1. Las transiciones hacia/desde los descendientes no se cuentan.
2. Los eventos `mouseenter/mouseleave` no brotan.

Son eventos extremadamente simples.

Cuando el cursor entra en un elemento `mouseenter` se activa. La ubicación exacta del cursor dentro del elemento o sus descendientes no importa.

Cuando el cursor deja el elemento `mouseleave` se activa.

Delegación de eventos

Los eventos `mouseenter/leave` son muy simples de usar. Pero no brotan por sí solos. Por lo tanto no podemos usar la delegación de eventos con ellos.

Imagina que queremos manejar entrada/salida para celdas de tabla y hay cientos de celdas.

La solución natural sería: ajustar el controlador en `<table>` y manejar los eventos desde ahí. Pero `mouseenter/leave` no aparece. Entonces si cada evento sucede en `<td>`, solamente un controlador `<td>` es capaz de detectarlo.

Los controladores `mouseenter/leave` en `<table>` solamente se activan cuando el cursor entra/deja la tabla completa. Es imposible obtener alguna información sobre las transiciones dentro de ella.

Pues usemos `mouseover/mouseout`.

Comencemos con controladores simples que resaltan el elemento debajo del mouse:

```
// Resaltemos un elemento debajo del cursor
table.onmouseover = function(event) {
  let target = event.target;
  target.style.background = 'pink';
};

table.onmouseout = function(event) {
  let target = event.target;
  target.style.background = '';
};
```

En nuestro caso nos gustaría manejar las transiciones entre las celdas de la tabla `<td>`: entradas y salidas de una celda a otra. Otras transiciones, como dentro de una celda o fuera de cualquiera de ellas no nos interesan. Vamos a filtrarlas.

Esto es lo que podemos hacer:

- Recordar el elemento `<td>` resaltado actualmente en una variable, llamémosla `currentElem`.
- En `mouseover` ignoraremos el evento si permanecemos dentro del `<td>` actual.
- En `mouseout` ignoraremos el evento si no hemos dejado el `<td>` actual.

Aquí hay un ejemplo de código que explica todas las situaciones posibles:

```
// Los elementos <td> bajo el mouse justo ahora(si es que hay)
let currentElem = null;

table.onmouseover = function(event) {
  // antes de ingresar un nuevo elemento, el mouse siempre abandonará al anterior
  // si currentElem está establecido, no abandonamos el <td> anterior,
  // hay un mouseover dentro de él, ignoramos el evento
  if (currentElem) return;

  let target = event.target.closest('td');

  // si no hay movimientos dentro de un <td> - lo ignoramos
  if (!target) return;

  //si hay movimientos dentro de un <td>, pero afuera de una tabla(posiblemente en caso de tablas anidadas)
  // lo ignoramos
  if (!table.contains(target)) return;

  // ¡Genial! ingresamos a un nuevo <td>
  currentElem = target;
  onEnter(currentElem);
```



```

};

table.onmouseout = function(event) {
  // si estamos afuera de algún <td> ahora, entonces ignoramos el evento
  // pueden haber movimientos dentro de una tabla pero fuera de <td>,
  // por ejemplo: de un <tr> a otro <tr>
  if (!currentElem) return;

  // abandonamos el elemento - ¿pero hacia dónde? ¿podría ser hacia un descendiente?
  let relatedTarget = event.relatedTarget;

  while (relatedTarget) {
    // vamos a la cadena de padres y verificamos - si aún estamos dentro de currentElem
    // entonces hay una transición interna - la ignoramos
    if (relatedTarget == currentElem) return;

    relatedTarget = relatedTarget.parentNode;
  }

  // abandonamos el <td>.
  onLeave(currentElem);
  currentElem = null;
};

// algunas funciones para manejar entradas/salidas de un elemento
function onEnter(elem) {
  elem.style.background = 'pink';

  // lo mostramos en el área de texto
  text.value += `over -> ${currentElem.tagName}.${currentElem.className}\n`;
  text.scrollTop = 1e6;
}

function onLeave(elem) {
  elem.style.background = '';

  // lo mostramos en el área de texto
  text.value += `out <- ${elem.tagName}.${elem.className}\n`;
  text.scrollTop = 1e6;
}

```

Una vez más, las características importantes son:

1. Utilizar la delegación de eventos para manejar la entrada/salida de cualquier `<td>` dentro de la tabla. Pues depende de `mouseover/out` en lugar de `mouseenter/leave` que no broten y por lo tanto no permita ninguna delegación.
2. Los eventos adicionales, como moverse entre descendientes de `<td>` son filtrados, así que `onEnter/Leave` solamente se ejecuta si el cursor ingresa a `<td>` o lo deja absolutamente.

Resumen

Hemos cubierto `mouseover`, `mouseout`, `mousemove`, `mouseenter` y `mouseleave`.

Estas cosas son buenas de destacar:

- Un movimiento rápido del mouse puede omitir elementos intermedios.
- Los eventos `mouseover/out` y `mouseenter/leave` tienen una propiedad adicional: `relatedTarget`. Es el elemento de donde venimos o hacia donde vamos, complementario con `target`.

Los eventos `mouseover/out` se activan incluso cuando vamos de un elemento padre a su descendiente. El navegador asume que de el mouse solo puede estar sobre un elemento a la vez – el más interno.

Los eventos `mouseenter/leave` son diferentes en ese aspecto: solo se activan cuando el mouse viene hacia el elemento o lo deja como un todo. Así que no se aparecen de repente.

✓ Tareas

Comportamiento mejorado de un tooltip

importancia: 5

Escribe JavaScript que muestre un tooltip sobre un elemento con el atributo `data-tooltip`. El valor de este atributo debe convertirse en el texto del tooltip.

Es como la tarea [Comportamiento: Tooltip](#), pero aquí los elementos anotados se pueden anidar. Los tooltips más internos se muestran.

Solamente un tooltip puede aparecer a la vez.

Por ejemplo:

```
<div data-tooltip="Aquí - está el interior de la casa" id="house">
  <div data-tooltip="Aquí - está el techo" id="roof"></div>
  ...
  <a href="https://en.wikipedia.org/wiki/The_Three_Little_Pigs" data-tooltip="Continúa leyendo...">Colócate sobre mi</a>
</div>
```

El resultado en el iframe:

Hover over me'." data-bbox="83 389 444 512"/>

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Tooltip "inteligente"

Escribe una función que muestre un tooltip sobre un elemento solamente si el visitante mueve el mouse *hacia él*, pero no *a través de él*.

En otras palabras, si el visitante mueve el mouse hacia el elemento y para ahí, muestra el tooltip. Y si solamente mueve el mouse a través, entonces no lo necesitamos. ¿Quién quiere parpadeos extra?

Técnicamente, podemos medir la velocidad del mouse sobre el elemento, y si es lenta podemos asumir que el mouse viene “sobre el elemento” y mostramos el tooltip, si es rápida – entonces lo ignoramos.

Hay que crear un objeto universal `new HoverIntent(options)` para ello.

Sus `options`:

- `elem` – elemento a seguir.
- `over` – una función a llamar si el mouse viene hacia el elemento: o sea, si viene lentamente o para sobre él.
- `out` – una función a llamar cuando el mouse abandona el elemento (si `over` fue llamado).

Un ejemplo de dicho objeto siendo usado para el tooltip:

```
// Un tooltip de muestra
let tooltip = document.createElement('div');
```

```

tooltip.className = "tooltip";
tooltip.innerHTML = "Tooltip";

// El objeto va a rastrear al mouse y llamar a over/out
new HoverIntent({
  elem,
  over() {
    tooltip.style.left = elem.getBoundingClientRect().left + 'px';
    tooltip.style.top = elem.getBoundingClientRect().bottom + 5 + 'px';
    document.body.append(tooltip);
  },
  out() {
    tooltip.remove();
  }
});

```

El demo:

12 : 30 : 00

passes: 5 failures: 0 duration: 0.02s 100%

Tooltip

verIntent

- ✓ mouseover -> when the pointer just arrived, no tooltip
- ✓ mouseover -> after a delay, the tooltip shows up
- ✓ mouseover -> followed by fast mouseout leads doesn't show tooltip

Si mueves el mouse sobre el “reloj” rápido no pasará nada, y si lo haces lento o paras sobre él entonces habrá un tooltip.

Toma en cuenta que el tooltip no “parpadea” cuando el cursor se mueve entre subelementos del reloj.

[Abrir en entorno controlado con pruebas.](#)

[A solución](#)

Arrastrar y Soltar con eventos del ratón

Arrastrar y Soltar es una excelente solución de interfaz. Tomar algo, arrastrar y soltarlo es una forma clara y simple de hacer muchas cosas, desde copiar y mover documentos (como en los manejadores de archivos) hasta ordenar (arrastrando ítems al carrito).

En el estándar moderno de HTML hay una [sección sobre Arrastrar y Soltar](#) con eventos especiales tales como `dragstart`, `dragend`, y así por el estilo.

Estos eventos nos permiten soportar tipos especiales de Arrastrar y Soltar, como manejar el arrastrado de archivos desde el manejador de archivos del Sistema Operativo y soltarlo en la ventana del navegador. Así JavaScript puede acceder al contenido de dichos archivos.

Pero los eventos nativos de arrastrar tienen limitaciones. Por ejemplo, no nos deja evitar el arrastre desde cierta área. Tampoco podemos hacer que el arrastre sea solamente “horizontal” o “vertical”. Y hay muchas otras tareas de “Arrastrar y Soltar” que no pueden hacerse utilizándolos. Además, el soporte para dichos eventos es muy pobre en dispositivos móviles.

Así que aquí veremos cómo implementar “Arrastrar y Soltar” usando eventos del ratón.

Algoritmo de “Arrastrar y Soltar”

El algoritmo básico de Arrastrar y Soltar se ve así:

- En `mousedown` – preparar el elemento para moverlo, si es necesario (quizá creando un clon de este, añadiéndole una clase, o lo que sea).
- En `mousemove` – moverlo cambiando `left/top` con `position:absolute`.
- En `mouseup` – realizar todas las acciones relacionadas con finalizar el Arrastrar y Soltar.

Esto es lo básico. Luego veremos como añadir características, como resaltar los elementos subyacentes mientras arrastramos sobre ellos.

Aquí esta la implementación de arrastrar una pelota:

```
ball.onmousedown = function(event) {
  // (1) preparar para mover: hacerlo absoluto y ponerlo sobre todo con el z-index
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;

  // quitar cualquier padre actual y moverlo directamente a body
  // para posicionarlo relativo al body
  document.body.append(ball);

  // centrar la pelota en las coordenadas (pageX, pageY)
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
  }

  // mover nuestra pelota posicionada absolutamente bajo el puntero
  moveAt(event.pageX, event.pageY);

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // (2) mover la pelota con mousemove
  document.addEventListener('mousemove', onMouseMove);

  // (3) soltar la pelota, quitar cualquier manejador de eventos innecesario
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };
};
```

Si ejecutamos el código, nos enteramos de algo extraño. Al inicio de arrastrar y soltar, la pelota se duplica: empezamos a arrastrar su “clon”.

Esto es porque el navegador tiene su propio soporte para arrastrar y soltar para imágenes y otros elementos. Se ejecuta automáticamente y entra en conflicto con el nuestro.

Para deshabilitarlo:

```
ball.ondragstart = function() {
  return false;
};
```

Ahora todo estará bien.

Otro aspecto importante: seguimos `mousemove` en `document`, no en `ball`. Desde el primer momento debe verse que el ratón está siempre sobre la pelota, y podemos poner `mousemove` en ella.

Pero como recordamos, `mousemove` se dispara a menudo, pero no por cada pixel. Así que después de un movimiento rápido el puntero puede saltar de la pelota a algún lugar en el medio del documento (o incluso fuera de la ventana).

Así que tenemos que escuchar en `document` para captarlo.

Posicionamiento correcto

En los ejemplos de arriba la pelota siempre se mueve, de manera que su centro queda debajo del puntero:

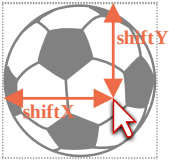
```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
```

```
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Nada mal, pero hay un efecto secundario. Para iniciar el arrastrar y soltar, podemos hacer `mousedown` en cualquier lugar de la pelota. Pero si la “tomamos” por el borde, entonces la pelota “salta” de repente para centrarse bajo el puntero del ratón.

Sería mejor si mantenemos la posición inicial del elemento, relativo al puntero.

Por ejemplo, si empezamos a arrastrar por el borde de la pelota, entonces el puntero debería quedarse sobre el borde mientras se arrastra.



Vamos a actualizar nuestro algoritmo:

1. Cuando un visitante presiona el botón (`mousedown`) – recordar la distancia del puntero a la esquina superior izquierda de la pelota en variables `shiftX/shiftY`. Mantendremos esa distancia mientras arrastramos.

Para obtener esas posiciones podemos restar las coordenadas:

```
// onmousedown
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

2. Entonces mientras arrastra posicionamos la pelota en la misma posición relativa al puntero, de esta forma:

```
// onmousemove
// la pelota tiene position:absolute
ball.style.left = event.pageX - shiftX + 'px';
ball.style.top = event.pageY - shiftY + 'px';
```

El código final con mejor posicionamiento:

```
ball.onmousedown = function(event) {

  let shiftX = event.clientX - ball.getBoundingClientRect().left;
  let shiftY = event.clientY - ball.getBoundingClientRect().top;

  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  document.body.append(ball);

  moveAt(event.pageX, event.pageY);

  // mueve la pelota a las coordenadas (pageX, pageY)
  // tomando la posición inicial en cuenta
  function moveAt(pageX, pageY) {
    ball.style.left = event.pageX - shiftX + 'px';
    ball.style.top = event.pageY - shiftY + 'px';
  }

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // mueve la pelota con mousemove
  document.addEventListener('mousemove', onMouseMove);

  // suelta la pelota, elimina el manejador innecesario
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
  }
}
```

```

    ball.onmouseup = null;
  };

};

ball.ondragstart = function() {
  return false;
};

```

La diferencia es notable especialmente si arrastramos la pelota por su esquina inferior derecha. En el ejemplo anterior la pelota “salta” bajo el puntero. Ahora sigue el puntero fluidamente desde su posición actual.

Objetivos receptores potenciales (droppables)

En los ejemplos anteriores la pelota debe ser soltada simplemente “en cualquier lugar” para quedarse. En la vida real normalmente tomamos un elemento para soltarlo en otro. Por ejemplo, un “archivo” en una “carpeta” o algo más.

Hablando abstracto, tomamos un elemento “arrastrable” y lo soltamos sobre un elemento “receptor”.

Necesitamos saber:

- dónde el elemento fue soltado al final del Arrastrar y Soltar – para hacer la acción correspondiente,
- y, preferiblemente, saber el receptor sobre el que estamos arrastrando, para resaltarlo.

La solución es algo interesante y un poco complicado, así que vamos a cubrirlo aquí.

¿Cuál puede ser la primera idea? ¿Probablemente configurar `mouseover/mouseup` en receptores potenciales?

Pero eso no funciona.

El problema es que, mientras estamos arrastrando, el elemento arrastrable siempre está encima de los demás elementos. Y los eventos del ratón solo suceden en el elemento superior, no en los que están debajo.

Por ejemplo, debajo hay dos elementos `<div>`, el rojo sobre el azul (totalmente cubierto). No hay forma de captar un evento en el azul, porque el rojo está encima:

```

<style>
  div {
    width: 50px;
    height: 50px;
    position: absolute;
    top: 0;
  }
</style>
<div style="background:blue" onmouseover="alert('nunca funciona')"></div>
<div style="background:red" onmouseover="alert('sobre el rojo!')"></div>

```



Lo mismo con un elemento arrastrable. La pelota está siempre sobre los demás elementos, así que los eventos pasan en él. Cualquier manejador que pongamos en los elementos de abajo, no funcionará.

Por eso la idea inicial de poner manejadores en receptores potenciales no funciona en práctica. No se activarán.

Entonces, ¿Qué hacer?

Hay un método llamado `document.elementFromPoint(clientX, clientY)`. Este retorna el elemento más anidado en las coordenadas relativas a la ventana proporcionadas (o `null` si las coordenadas están fuera de la ventana).

Podemos utilizarlo en cualquiera de nuestros manejadores para detectar los receptores potenciales bajo el puntero, de esta forma:

```

// en un manejador de evento del ratón
ball.hidden = true; // (*) ocultar el elemento que arrastramos

```

```
let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow es el elemento debajo de la pelota, puede ser receptor

ball.hidden = false;
```

Favor notar: necesitamos ocultar la pelota antes de llamar `(*)`. De otra forma usualmente tendremos una pelota con esas coordenadas, ya que es el elemento superior bajo el puntero: `elemBelow=ball`. Así que lo ocultamos e inmediatamente lo mostramos de nuevo.

Podemos usar este código para verificar el elemento sobre el que estamos “flotando” en todo momento. Y manejar la caída cuando sucede.

Un código extendido de `onMouseMove` para hallar elementos “receptores”:

```
// elemento potencialmente arrastrable sobre el que flotamos ahora mismo
let currentDraggable = null;

function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);

  ball.hidden = true;
  let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
  ball.hidden = false;

  // los eventos mousemove se pueden activar fuera de la ventana (cuando la pelota se arrastra fuera de la ventana)
  // si clientX/clientY están fuera de la ventana, entonces elementFromPoint devuelve null
  if (!elemBelow) return;

  // receptores potenciales se etiquetan con la clase "draggable" (puede tener otra lógica)
  let draggableBelow = elemBelow.closest('.draggable');

  if (currentDraggable !== draggableBelow) {
    // estamos flotando dentro o afuera
    // nota: ambos valores pueden ser null
    // currentDraggable=null si no estábamos sobre un receptor antes de este evento (ej. sobre un espacio en blanco)
    // draggableBelow=null si no estamos sobre un receptor ahora, durante este evento

    if (currentDraggable) {
      // la lógica para procesar "flying out" del receptor (elimina el resaltado)
      leaveDraggable(currentDraggable);
    }
    currentDraggable = draggableBelow;
    if (currentDraggable) {
      // la lógica para procesar "flying in" del receptor
      enterDraggable(currentDraggable);
    }
  }
}
```

En el siguiente ejemplo cuando la pelota se arrastra sobre la portería, esta se resalta.

<https://plnkr.co/edit/6GjSD6KXPkxZgleD?p=preview>

Ahora tenemos el “destino” actual, sobre el que estamos flotando, en la variable `currentDraggable` durante el proceso completo y podemos usarlo para resaltar o cualquier otra cosa.

Resumen

Consideramos un algoritmo básico de Arrastrar y Soltar.

Los componentes clave:

1. Flujo de eventos: `ball.mousedown` → `document.mousemove` → `ball.mouseup` (no olvides cancelar el `ondragstart` nativo).
2. El inicio del arrastrado – recuerda la posición inicial del puntero relativo al elemento: `shiftX/shiftY` y lo mantiene durante el arrastrado.
3. Detectar elementos arrastrables bajo el puntero usando `document.elementFromPoint`.

Podemos poner mucho sobre esta base.

- Con `mouseup` podemos intelectualmente finalizar el arrastre: cambiar datos, mover elementos alrededor.
- Podemos resaltar los elementos sobre los que estamos volando.
- Podemos limitar el arrastrado a cierta área o dirección.
- Podemos usar delegación de eventos para `mousedown/up`. Un manejador de eventos para un área grande que compruebe `event.target` puede manejar Arrastrar y Soltar para cientos de elementos.
- Y así por el estilo.

Hay frameworks que construyen una arquitectura sobre esto: `DragZone`, `Droppable`, `Draggable` y otras clases. La mayoría de ellos hacen cosas similares a lo que hemos descrito, así que debería ser fácil entenderlos ahora. O crea el tuyo propio: como puedes ver es fácil de hacer, a veces es más fácil que adaptarse a una solución de terceros.

✓ Tareas

Control deslizante

importancia: 5

Crea un control deslizante:



Arrastra el pasador azul con el ratón y muévelo.

Detalles importantes:

- Cuando el botón del ratón es presionado, durante el arrastrado del ratón puedes ir por arriba o debajo de la barra deslizante. Ésta seguirá funcionando (es lo conveniente para el usuario).
- Si el ratón se mueve muy rápido hacia la izquierda o la derecha, el pasador se detiene exactamente en el borde.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Arrastrar super héroes por el campo

importancia: 5

Esta tarea te puede ayudar a comprobar tu entendimiento de varios aspectos de Arrastrar y Soltar, y del DOM.

Hacer que todos los elementos con clase `draggable` sean arrastrables. Como la pelota de este capítulo.

Requerimientos:

- Usa delegación de eventos para detectar el inicio del arrastrado: un solo manejador de eventos en el `document` para `mousedown`.
- Si los elementos son arrastrados a los bordes superior/inferior de la ventana: la página se desliza hacia arriba/abajo para permitir dicho arrastre.
- Sin desplazamiento horizontal (esto hace la tarea un poco más simple, añadirlo es fácil).
- Los elementos arrastrables o sus partes nunca deben dejar la ventana, incluso después de movimientos rápidos del ratón.

La demostración es demasiado grande para caber aquí, así que aquí está el enlace.

[Demo en nueva ventana](#) ↗

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Eventos de puntero

Los eventos de puntero son una forma moderna de manejar la entrada de una variedad de dispositivos señaladores, como un mouse, un lápiz, una pantalla táctil, etc.

Una breve historia

Hagamos una pequeña descripción general para que comprenda la imagen general y el lugar de los Eventos de Puntero entre otros tipos de eventos.

- Hace mucho tiempo, en el pasado, solo existían eventos de mouse.

Luego aparecieron los dispositivos táctiles. Para que el código antiguo funcione, también generan eventos de mouse. Por ejemplo, tocar genera `mousedown`.

Pero los dispositivos táctiles tienen más capacidades que un mouse. Por ejemplo, es posible tocar múltiples puntos al mismo ("multi-touch"). Los eventos del mouse no tienen las propiedades necesarias para manejar tal multi-touch.

- Por lo tanto, se introdujeron eventos táctiles, como `touchstart`, `touchend`, `touchmove`, que tienen propiedades específicas de toque (no los cubrimos en detalle aquí, porque los eventos de puntero son aún mejores).

Aún así no fue suficiente, ya que hay muchos otros dispositivos, como los lápices, que tienen sus propias funciones. Además, escribir un código que escuchara ambos eventos, los táctiles y los del mouse era engorroso.

- Para resolver estos problemas, se introdujo el nuevo estándar: *Pointer Events*. Este proporciona un conjunto único de eventos para todo tipo de dispositivos señaladores.

Al momento la especificación [Pointer Events Level 2](#) es compatible con todos los principales navegadores, mientras que [Pointer Events Level 3](#) está en proceso y es mayormente compatible con Pointer Events level 2.

A menos que codifique para navegadores viejos tales como Internet Explorer 10 o Safari 12 y versiones anteriores, ya no tiene sentido usar el mouse o los eventos táctiles: podemos cambiar a eventos de puntero.

Dicho esto, hay peculiaridades importantes, uno debe saber usarlas correctamente y evitar sorpresas adicionales. Les prestaremos atención en este artículo.

Tipos de eventos de puntero

Los eventos de puntero se llaman de forma similar a los eventos del mouse:

Evento de puntero	Evento de mouse similar
<code>pointerdown</code>	<code>mousedown</code>
<code>pointerup</code>	<code>mouseup</code>
<code>pointermove</code>	<code>mousemove</code>
<code>pointerover</code>	<code>mouseover</code>
<code>pointerout</code>	<code>mouseout</code>
<code>pointerenter</code>	<code>mouseenter</code>
<code>pointerleave</code>	<code>mouseleave</code>
<code>pointercancel</code>	-
<code>gotpointercapture</code>	-
<code>lostpointercapture</code>	-

Como podemos ver, para cada `mouse<event>`, hay un `pointer<event>` que juega un papel similar. También hay 3 eventos de puntero adicionales que no tienen una contraparte correspondiente de `mouse` ..., pronto hablaremos sobre ellos.

i Reemplazando *mouse* con *pointer* en nuestro código

Podemos reemplazar los eventos `mouse<event>` con `pointer<event>` en nuestro código y esperar que las cosas sigan funcionando bien con el mouse.

El soporte para dispositivos táctiles también mejorará “mágicamente”, pero probablemente necesitemos agregar la regla `touch-action: none` en CSS. Vea los detalles a continuación en la sección sobre `pointercancel`.

Propiedades de los eventos de puntero

Los eventos de puntero tienen las mismas propiedades que los eventos del mouse, como `clientX/Y`, `target`, etc., más algunos adicionales:

- `pointerId` – el identificador único del puntero que causa el evento.
Generado por el navegador. Permite manejar múltiples punteros, como una pantalla táctil con lápiz y multitáctil (explicado a continuación).
- `pointerType` – el tipo de dispositivo señalador. Debe ser una cadena, uno de los siguientes: “mouse”, “pen” o “touch”.
Podemos usar esta propiedad para reaccionar de manera diferente en varios tipos de punteros.
- `isPrimary` – `true` para el puntero principal (el primer dedo en multitáctil).

Para punteros que miden un área de contacto y presión, p. Ej. un dedo en la pantalla táctil, las propiedades adicionales pueden ser útiles:

- `width` – el ancho del área donde el puntero toca el dispositivo. Donde no sea compatible, como en el mouse, siempre es `1`.
- `height` – el alto del área donde el puntero toca el dispositivo. Donde no sea compatible, siempre es `1`.
- `pressure` – la presión de la punta del puntero, en el rango de 0 a 1. Para dispositivos que no soportan presión, debe ser `0.5` (presionada) o `0`.
- `tangentialPressure` – la presión tangencial normalizada.
- `tiltX`, `tiltY`, `twist` – propiedades específicas del lápiz que describen cómo se lo coloca en relación con la superficie.

Estas propiedades no son muy compatibles en todos los dispositivos, por lo que rara vez se utilizan. Puede encontrar los detalles en la [especificación](#) si lo necesita.

Multi-touch (Multitáctil)

Una de las cosas con las que los eventos del mouse no son compatibles es la propiedad multitáctil: un usuario puede tocarlos en varios lugares a la vez en su teléfono o tableta, realizar gestos especiales.

Los eventos de puntero permiten manejar multitáctiles con la ayuda de las propiedades `pointerId` e `isPrimary`.

Esto es lo que sucede cuando un usuario toca una pantalla en un lugar y luego coloca otro dedo en otro lugar:

1. En el primer toque:
 - `pointerdown` with `isPrimary=true` y algún `pointerId`.
2. Para el segundo dedo y toques posteriores (asumiendo que el primero sigue tocando):
 - `pointerdown` con `isPrimary=false` y un diferente `pointerId` por cada dedo.

Tenga en cuenta: el `pointerId` no se asigna a todo el dispositivo, sino a cada dedo que se toca. Si usamos 5 dedos para tocar simultáneamente la pantalla, tenemos 5 eventos `pointerdown` con coordenadas respectivas y diferentes `pointerId`.

Los eventos asociados con el primer dedo siempre tienen `isPrimary = true`.

Podemos rastrear el toque de varios dedos usando sus respectivos `pointerId`. Cuando el usuario mueve un dedo y luego lo quita, obtenemos los eventos `pointermove` y `pointerup` con el mismo `pointerId` que teníamos en `pointerdown`.

Evento: `pointercancel`

El evento `pointercancel` se dispara cuando hay una interacción de puntero en curso, y luego sucede algo que hace que se anule, de modo que no se generan más eventos de puntero.

Algunas causas son:

- Se deshabilitó el hardware del dispositivo de puntero.
- La orientación del dispositivo cambió (tableta rotada).
- El navegador decidió manejar la interacción por sí solo, considerándolo un gesto del mouse o una acción de zoom y panorámica u otra cosa.

Demostraremos `pointercancel` en un ejemplo práctico para ver cómo nos afecta.

Digamos que estamos implementando arrastrar y soltar (drag'n'drop) en una pelota, como al principio del artículo [Arrastrar y Soltar con eventos del ratón](#).

A continuación, se muestra el flujo de acciones del usuario y los eventos correspondientes:

1. El usuario presiona sobre una imagen para comenzar a arrastrar
 - `pointerdown` el evento se dispara
2. Luego comienzan a mover el puntero (arrastrando la imagen)
 - `pointermove` se dispara, tal vez varias veces
3. ¡Sorpresa! El navegador tiene soporte nativo de arrastrar y soltar para imágenes, que se dispara y se hace cargo del proceso de arrastrar y soltar, generando así el evento `pointercancel`.
 - El navegador ahora maneja arrastrar y soltar la imagen por sí solo. El usuario puede incluso arrastrar la imagen de la bola fuera del navegador, a su programa de correo o al administrador de archivos.
 - No más eventos `pointermove` para nosotros.

Así que el problema es que el navegador “secuestra” la interacción: `pointercancel` se dispara y no se generan más eventos de `pointermove`.

Nos gustaría implementar nuestro propio arrastrar y soltar, así que digámosle al navegador que no se haga cargo.

Evitar las acciones predeterminadas del navegador para prevenir `pointercancel`.

Necesitaremos dos cosas:

1. Evitar que suceda la función nativa de arrastrar y soltar:
 - Puede hacerlo configurando `ball.ondragstart = () => false`, tal como se describe en el artículo [Arrastrar y Soltar con eventos del ratón](#).
 - Eso funciona bien para eventos de mouse.
2. Para los dispositivos táctiles, también existen acciones del navegador relacionadas con el tacto (además de arrastrar y soltar). También tendremos problemas con ellos.
 - Podemos prevenirlos configurando `#ball{touch-action: none}` en CSS.
 - Entonces nuestro código comenzará a funcionar en dispositivos táctiles.

Después de hacer eso, los eventos funcionarán según lo previsto, el navegador no secuestrará el proceso y no emitirá ningún `pointercancel`.

Ahora podemos agregar el código para mover realmente la bola, y nuestro método de arrastrar y soltar funcionará en dispositivos de mouse y dispositivos táctiles.

Captura del puntero

La captura de puntero es una característica especial de los eventos de puntero.

La idea es muy simple, pero puede verse extraña al principio, porque no existe algo así para ningún otro tipo de evento.

El método principal es:

- `elem.setPointerCapture(pointerId)` – vincula el `pointerId` dado a `elem`. Después del llamado todos los eventos de puntero con el mismo `pointerId` tendrán `elem` como objetivo (como si pasara sobre `elem`), no

importa dónde en el documento realmente ocurrió.

En otras palabras, `elem.setPointerCapture(pointerId)` redirecciona todos los subsecuentes eventos con el `pointerId` dado hacia `elem`.

El vínculo se deshace::

- automáticamente cuando ocurren los eventos `pointerup` o `pointercancel`,
- automáticamente cuando `elem` es quitado del documento,
- cuando `elem.releasePointerCapture(pointerId)` es llamado.

La captura de puntero puede utilizarse para simplificar el tipo de interacciones de arrastrar y soltar.

Recordemos el problema que encontramos al hacer un control deslizante personalizado en el artículo [Arrastrar y Soltar con eventos del ratón](#).

Hacemos un elemento deslizante con la corredera y botón (`thumb`) dentro.

Funciona así:

1. Primero, el usuario presiona en el deslizante – se dispara `pointerdown`
2. Entonces mueve el puntero – se dispara `pointermove` y mueve el botón `thumb`
 - ...mientras el puntero se mueve, puede salirse del control deslizante: que vaya por debajo o por encima de él. El botón debe moverse estrictamente horizontal, permaneciendo alineado con el puntero.

Entonces, para rastrear todos los movimientos del puntero, incluyendo aquellos por arriba o por debajo, tenemos que asignar el controlador de evento `pointermove` al `document` entero.

Esta solución se ve un poco “sucía”. Uno de los problemas es que los movimientos alrededor del documento pueden causar efectos colaterales, disparar otros eventos, completamente sin relación al deslizante.

La captura de puntero provee un medio de vincular `pointermove` a `thumb` y evitar cualquiera de tales problemas:

- Podemos llamar `thumb.setPointerCapture(event.pointerId)` en el controlador `pointerdown`,
- Entonces futuros eventos de puntero hasta `pointerup/cancel` serán redirigidos a `thumb`.
- Cuando ocurre `pointerup` (arrastre finalizado), el vínculo se deshace automáticamente, no necesitamos atender eso.

Entonces, incluso si el usuario mueve el puntero alrededor de todo el documento, los controladores de eventos serán llamados sobre `thumb`. Además las propiedades de coordenadas de los eventos, tales como `clientX/clientY` aún serán correctas, la captura solo afecta a `target/currentTarget`.

Aquí está el código esencial:

```
thumb.onpointerdown = function(event) {
  // reorienta todos los eventos de puntero (hasta pointerup) a thumb
  thumb.setPointerCapture(event.pointerId);
};

thumb.onpointermove = function(event) {
  // mueve el control deslizante: escucha a thumb, ya que todos los eventos se redireccionan a él
  let newLeft = event.clientX - slider.getBoundingClientRect().left;
  thumb.style.left = newLeft + 'px';
};

// nota: no es necesario llamar a thumb.releasePointerCapture,
// esto sucede con el pointerup automáticamente
```

Finalizando, la captura de puntero nos brinda dos beneficios:

1. El código se vuelve más claro, ya no necesitamos agregar o quitar controladores para el `document` entero. El vínculo se deshace automáticamente.
2. Si hay cualquier controlador `pointermove` en el documento, no serán disparados accidentalmente mientras el usuario está arrastrando el deslizante.

Eventos de captura de puntero

Hay dos eventos de puntero asociados:

- `gotpointercapture` se dispara cuando un elemento usa `setPointerCapture` para permitir la captura.
- `lostpointercapture` se dispara cuando se libera la captura: ya sea explícitamente con la llamada a `releasePointerCapture`, o automáticamente con `pointerup/pointercancel`.

Resumen

Los eventos de puntero permiten manejar eventos de mouse, toque y lápiz simultáneamente con una simple pieza de código.

Los eventos de puntero extienden los eventos del mouse. Podemos reemplazar `mouse` con `pointer` en los nombres de los eventos y esperar que nuestro código continúe funcionando para el mouse, con mejor soporte para otros tipos de dispositivos.

Para arrastrar y soltar, y complejas interacciones que el navegador pudiera decidir secuestrar y manejarlas por su cuenta, recuerde cancelar la acción predeterminada sobre eventos y establecer `touch-events: none` en CSS para los elementos que involucramos.

Las habilidades adicionales de los eventos Pointer son:

- Soporte multitáctil usando `pointerId` y `isPrimary`.
- Propiedades específicas del dispositivo, como `pressure`, `width/height` y otras.
- Captura de puntero: podemos redirigir todos los eventos de puntero a un elemento específico hasta `pointerup/pointercancel`.

Al momento los eventos de puntero son compatibles con todos los navegadores principales, por lo que podemos cambiarlos de forma segura si no se necesitan IE10 y Safari 12. E incluso con esos navegadores, existen polyfills que permiten la compatibilidad con eventos de puntero.

Keyboard: keydown and keyup

Before we get to keyboard, please note that on modern devices there are other ways to “input something”. For instance, people use speech recognition (especially on mobile devices) or copy/paste with the mouse.

So if we want to track any input into an `<input>` field, then keyboard events are not enough. There's another event named `input` to track changes of an `<input>` field, by any means. And it may be a better choice for such task. We'll cover it later in the chapter [Eventos: change, input, cut, copy, paste](#).

Keyboard events should be used when we want to handle keyboard actions (virtual keyboard also counts). For instance, to react on arrow keys `Up` and `Down` or hotkeys (including combinations of keys).

Teststand

To better understand keyboard events, you can use the [teststand](#).

Keydown and keyup

The `keydown` events happens when a key is pressed down, and then `keyup` – when it's released.

event.code and event.key

The `key` property of the event object allows to get the character, while the `code` property of the event object allows to get the “physical key code”.

For instance, the same key `Z` can be pressed with or without `Shift`. That gives us two different characters: lowercase `z` and uppercase `Z`.

The `event.key` is exactly the character, and it will be different. But `event.code` is the same:

Key	event.key	event.code
<code>Z</code>	<code>z (lowercase)</code>	<code>KeyZ</code>

Key	event.key	event.code
Shift+Z	Z (uppercase)	KeyZ

If a user works with different languages, then switching to another language would make a totally different character instead of "Z". That will become the value of `event.key`, while `event.code` is always the same: "KeyZ".

"KeyZ" and other key codes

Every key has the code that depends on its location on the keyboard. Key codes described in the [UI Events code specification](#).

For instance:

- Letter keys have codes `"Key<letter>": "KeyA", "KeyB"` etc.
- Digit keys have codes: `"Digit<number>": "Digit0", "Digit1"` etc.
- Special keys are coded by their names: `"Enter", "Backspace", "Tab"` etc.

There are several widespread keyboard layouts, and the specification gives key codes for each of them.

Read the [alphanumeric section of the spec](#) for more codes, or just press a key in the [teststand](#) above.

Case matters: "KeyZ", not "keyZ"

Seems obvious, but people still make mistakes.

Please evade mistypes: it's `KeyZ`, not `keyZ`. The check like `event.code=="keyZ"` won't work: the first letter of "Key" must be uppercase.

What if a key does not give any character? For instance, `Shift` or `F1` or others. For those keys, `event.key` is approximately the same as `event.code`:

Key	event.key	event.code
F1	F1	F1
Backspace	Backspace	Backspace
Shift	Shift	ShiftRight or ShiftLeft

Please note that `event.code` specifies exactly which key is pressed. For instance, most keyboards have two `Shift` keys: on the left and on the right side. The `event.code` tells us exactly which one was pressed, and `event.key` is responsible for the "meaning" of the key: what it is (a "Shift").

Let's say, we want to handle a hotkey: `Ctrl+Z` (or `Cmd+Z` for Mac). Most text editors hook the "Undo" action on it. We can set a listener on `keydown` and check which key is pressed.

There's a dilemma here: in such a listener, should we check the value of `event.key` or `event.code`?

On one hand, the value of `event.key` is a character, it changes depending on the language. If the visitor has several languages in OS and switches between them, the same key gives different characters. So it makes sense to check `event.code`, it's always the same.

Like this:

```
document.addEventListener('keydown', function(event) {
  if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
    alert('Undo!')
  }
});
```

On the other hand, there's a problem with `event.code`. For different keyboard layouts, the same key may have different characters.

For example, here are US layout ("QWERTY") and German layout ("QWERTZ") under it (from Wikipedia):

~	!	@	#	\$	%	^	&	*	()	-	=	Backspace
Tab	Q	W	E	R	T	Y	U	I	O	P	{	}	
Caps Lock	A	S	D	F	G	H	J	K	L	:	"	'	Enter
Shift	Z	X	C	V	B	N	M	<	>	?	/	Shift	
Ctrl	Win Key	Alt							Alt	Win Key	Menu	Ctrl	

°	!	"	§	\$	%	&	/	()	=	?	~	Backspace
^	1	2	3	4	5	6	7	{	8	[9]	0
↔	Q	W	E	R	T	Z	U	I	O	P	Ü	*	
↕	A	S	D	F	G	H	J	K	L	Ö	Ä	'	Enter
↕	>	Y	X	C	V	B	N	M	;	:	-	Shift	
Strg	Win	Alt							Alt Gr	Win	Menu	Strg	

For the same key, US layout has “Z”, while German layout has “Y” (letters are swapped).

Literally, `event.code` will equal `KeyZ` for people with German layout when they press `Y`.

If we check `event.code == 'KeyZ'` in our code, then for people with German layout such test will pass when they press `Y`.

That sounds really odd, but so it is. The [specification](#) explicitly mentions such behavior.

So, `event.code` may match a wrong character for unexpected layout. Same letters in different layouts may map to different physical keys, leading to different codes. Luckily, that happens only with several codes, e.g. `keyA`, `keyQ`, `keyZ` (as we’ve seen), and doesn’t happen with special keys such as `Shift`. You can find the list in the [specification](#).

To reliably track layout-dependent characters, `event.key` may be a better way.

On the other hand, `event.code` has the benefit of staying always the same, bound to the physical key location, even if the visitor changes languages. So hotkeys that rely on it work well even in case of a language switch.

Do we want to handle layout-dependant keys? Then `event.key` is the way to go.

Or we want a hotkey to work even after a language switch? Then `event.code` may be better.

Auto-repeat

If a key is being pressed for a long enough time, it starts to “auto-repeat”: the `keydown` triggers again and again, and then when it’s released we finally get `keyup`. So it’s kind of normal to have many `keydown` and a single `keyup`.

For events triggered by auto-repeat, the event object has `event.repeat` property set to `true`.

Default actions

Default actions vary, as there are many possible things that may be initiated by the keyboard.

For instance:

- A character appears on the screen (the most obvious outcome).
- A character is deleted (`Delete` key).
- The page is scrolled (`PageDown` key).
- The browser opens the “Save Page” dialog (`Ctrl+S`)
- ...and so on.

Preventing the default action on `keydown` can cancel most of them, with the exception of OS-based special keys. For instance, on Windows `Alt+F4` closes the current browser window. And there’s no way to stop it by preventing the default action in JavaScript.

For instance, the `<input>` below expects a phone number, so it does not accept keys except digits, `+`, `(` or `-`:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-';
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please" type="tel">
```

Phone, please

Please note that special keys, such as `Backspace`, `Left`, `Right`, `Ctrl+V`, do not work in the input. That's a side-effect of the strict filter `checkPhoneKey`.

Let's relax it a little bit:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-' ||
    key == 'ArrowLeft' || key == 'ArrowRight' || key == 'Delete' || key == 'Backspace';
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please" type="tel">
```

Phone, please

Now arrows and deletion works well.

...But we still can enter anything by using a mouse and right-click + Paste. So the filter is not 100% reliable. We can just let it be like that, because most of time it works. Or an alternative approach would be to track the `input` event – it triggers after any modification. There we can check the new value and highlight/modify it when it's invalid.

Legacy

In the past, there was a `keypress` event, and also `keyCode`, `charCode`, `which` properties of the event object.

There were so many browser incompatibilities while working with them, that developers of the specification had no way, other than deprecating all of them and creating new, modern events (described above in this chapter). The old code still works, as browsers keep supporting them, but there's totally no need to use those any more.

Mobile Keyboards

When using virtual/mobile keyboards, formally known as IME (Input-Method Editor), the W3C standard states that a `KeyboardEvent`'s `e.keyCode` should be `229` and `e.key` should be `"Unidentified"`.

While some of these keyboards might still use the right values for `e.key`, `e.code`, `e.keyCode` ... when pressing certain keys such as arrows or backspace, there's no guarantee, so your keyboard logic might not always work on mobile devices.

Summary

Pressing a key always generates a keyboard event, be it symbol keys or special keys like `Shift` or `Ctrl` and so on. The only exception is `Fn` key that sometimes presents on a laptop keyboard. There's no keyboard event for it, because it's often implemented on lower level than OS.

Keyboard events:

- `keydown` – on pressing the key (auto-repeats if the key is pressed for long),
- `keyup` – on releasing the key.

Main keyboard event properties:

- `code` – the “key code” (`"KeyA"` , `"ArrowLeft"` and so on), specific to the physical location of the key on keyboard.
- `key` – the character (`"A"` , `"a"` and so on), for non-character keys, such as `Esc` , usually has the same value as `code` .

In the past, keyboard events were sometimes used to track user input in form fields. That's not reliable, because the input can come from various sources. We have `input` and `change` events to handle any input (covered later in the chapter [Eventos: change, input, cut, copy, paste](#)). They trigger after any kind of input, including copy-pasting or speech recognition.

We should use keyboard events when we really want keyboard. For example, to react on hotkeys or special keys.

✓ Tareas

Extended hotkeys

importancia: 5

Create a function `runOnKeys(func, code1, code2, ... code_n)` that runs `func` on simultaneous pressing of keys with codes `code1` , `code2` , ..., `code_n` .

For instance, the code below shows `alert` when `"Q"` and `"W"` are pressed together (in any language, with or without CapsLock)

```
runOnKeys(
  () => alert("Hello!"),
  "KeyQ",
  "KeyW"
);
```

[Demo en nueva ventana](#) ↗

[A solución](#)

Desplazamiento

El evento `scroll` permite reaccionar al desplazamiento de una página o elemento. Hay bastantes cosas buenas que podemos hacer aquí.

Por ejemplo:

- Mostrar/ocultar controles o información adicional según el lugar del documento en el que se encuentre el/la usuario/a.
- Cargar más datos cuando el/la usuario/a se desplaza hacia abajo hasta el final del documento.

Aquí hay una pequeña función para mostrar el desplazamiento actual:

```
window.addEventListener('scroll', function() {
  document.getElementById('showScroll').innerHTML = window.pageYOffset + 'px';
});
```

El evento `scroll` funciona tanto en `window` como en los elementos desplazables.

Evitar el desplazamiento

¿Qué hacemos para que algo no se pueda desplazar?

No podemos evitar el desplazamiento utilizando `event.preventDefault()` oyendo al evento `onscroll` , porque este se activa *después* de que el desplazamiento haya ocurrido.

Pero podemos prevenir el desplazamiento con `event.preventDefault()` en un evento que cause el desplazamiento, por ejemplo en el evento `keydown` para `pageUp` y `pageDown`.

Si añadimos un manejador de eventos a estos eventos y un `event.preventDefault()` en el manejador, entonces el desplazamiento no se iniciará.

Hay muchas maneras de iniciar un desplazamiento, la más fiable es usar CSS, la propiedad `overflow`.

Aquí hay algunas tareas que puede resolver o revisar para ver aplicaciones de `onscroll`.

✓ Tareas

Página sin fin

importancia: 5

Crear una página interminable. Cuando un visitante la desplace hasta el final, se auto-añadirá la fecha y hora actual al texto (así el visitante podrá seguir desplazándose)

Así:

Scroll me

Date: Sun Jun 13 2021 15:00:35 GMT+0300 (Moscow Standard Time)

Date: Sun Jun 13 2021 15:00:35 GMT+0300 (Moscow Standard Time)

Date: Sun Jun 13 2021 15:00:35 GMT+0300 (Moscow Standard Time)

Date: Sun Jun 13 2021 15:00:35 GMT+0300 (Moscow Standard Time)

Por favor tenga en cuenta dos características importantes del desplazamiento:

1. **El scroll es “elástico”**. En algunos navegadores/dispositivos podemos desplazarnos un poco más allá del inicio o final del documento (se muestra un espacio vacío abajo, y luego el documento “rebota” automáticamente a la normalidad).
2. **El scroll es impreciso**. Cuando nos desplazamos hasta el final de la página, podemos estar de hecho como a 0-50px del fondo del documento real.

Así que, “desplazarse hasta el final” debería significar que el visitante no está a más de 100px del final del documento.

P.D. En la vida real podemos querer mostrar “más mensajes” o “más bienes”.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Botón para subir/bajar

importancia: 5

Crea un botón “ir arriba” para ayudar con el desplazamiento de la página.

Debería funcionar así:

- Mientras que la página no se desplace hacia abajo al menos la altura de la ventana... es invisible.
- Cuando la página se desplaza hacia abajo más que la altura de la ventana – aparece una flecha “hacia arriba” en la esquina superior izquierda. Si la página se desplaza hacia atrás desaparece.
- Cuando se hace click en la flecha, la página se desplaza hacia arriba hasta el tope.

Así (esquina superior izquierda, desplácese para ver):

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115 116
117 118 119 120 121 122 123 124 125 126 127 128 129 130
131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158
159 160 161 162 163 164 165 166 167 168 169 170 171 172
173 174 175 176 177 178 179 180 181 182 183 184 185 186
```

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Cargar imágenes visibles

importancia: 4

Digamos que tenemos un cliente con baja velocidad de conexión y queremos cuidar su tarifa de datos.

Para ello decidimos no mostrar las imágenes inmediatamente, sino sustituirlas por marcadores de posición, como este:

```

```

Así que, inicialmente todas las imágenes son `placeholder.svg`. Cuando la página se desplaza a la posición donde el usuario puede ver la imagen – cambiamos `src` a `data-src`, y así la imagen se carga.

Aquí hay un ejemplo en `iframe`:

Text and pictures are from <https://wikipedia.org>.

All images with data-src load when become visible.

Solar system

The Solar System is the gravitationally bound system comprising the Sun and the objects that orbit it, either directly or indirectly. Of those objects that orbit the Sun directly, the largest eight are the planets, with the remainder being significantly smaller objects, such as dwarf planets and small Solar System bodies. Of the objects that orbit the Sun indirectly, the moons, two are larger than the smallest planet, Mercury.

The Solar System formed 4.6 billion years ago from the gravitational collapse of a giant interstellar molecular cloud. The vast majority of the system's mass is in the Sun, with most of the remaining mass contained in Jupiter. The four smaller inner planets, Mercury, Venus, Earth and Mars, are terrestrial planets, being primarily composed of rock and metal. The four outer planets are giant planets, being substantially more massive than the terrestrials. The two largest, Jupiter and Saturn, are gas giants, being composed mainly of hydrogen and helium; the two outermost planets,

Desplázate para ver las imágenes cargadas “bajo demanda”.

Requerimientos:

- Cuando la página se carga, las imágenes que están en pantalla deben cargarse inmediatamente, antes de cualquier desplazamiento.
- Algunas imágenes pueden ser regulares, sin `data-src`. El código no debe tocarlas.
- Una vez que una imagen se carga, no debe recargarse más cuando haya desplazamiento arriba/abajo.

P.D. Si puedes, haz una solución más avanzada para “precargar” las imágenes que están más abajo/después de la posición actual.

Post P.D. Sólo se debe manejar el desplazamiento vertical, no el horizontal.

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Formularios y controles

Propiedades especiales y eventos de formularios `<form>` y controles: `<input>`, `<select>` y otros.

Propiedades y Métodos de Formularios

Los formularios y controles, como `<input>`, tienen muchos eventos y propiedades especiales.

Trabajar con formularios será mucho más conveniente cuando los aprendamos.

Navegación: Formularios y elementos

Los formularios del documento son miembros de la colección especial `document.forms`.

Esa es la llamada “Colección nombrada”: es ambas cosas, nombrada y ordenada. Podemos usar el nombre o el número en el documento para conseguir el formulario.

```
document.forms.my; // el formulario con name="my"
document.forms[0]; // el primer formulario en el documento
```

Cuando tenemos un formulario, cualquier elemento se encuentra disponible en la colección nombrada `form.elements`.

Por ejemplo:

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  // obtención del formulario
  let form = document.forms.my; // elemento <form name="my">

  // get the element
  let elem = form.elements.one; // elemento <input name="one">

  alert(elem.value); // 1
</script>
```

Puede haber múltiples elementos con el mismo nombre. Esto es típico en el caso de los botones de radio y checkboxes.

En ese caso `form.elements[name]` es una *colección*. Por ejemplo:

```
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0]); // [object HTMLInputElement]
</script>
```

Estas propiedades de navegación no dependen de la estructura de las etiquetas. Todos los controles, sin importar qué tan profundos se encuentren en el formulario, están disponibles en `form.elements`.

Fieldsets como “sub-formularios”

Un formulario puede tener uno o varios elementos `<fieldset>` dentro. Estos también tienen la propiedad `elements` que enlista los controles del formulario dentro de ellos.

Por ejemplo:

```
<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>

  <script>
    alert(form.elements.login); // <input name="login">

    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement

    // podemos obtener el input por su nombre tanto desde el formulario como desde el fieldset
    alert(fieldset.elements.login == form.elements.login); // true
  </script>
</body>
```

Notación corta: `form.name`

Hay una notación corta: podemos acceder el elemento como `form[index/name]`.

En otras palabras, en lugar de `form.elements.login` podemos escribir `form.login`.

Esto también funciona, pero tiene un error menor: si accedemos un elemento, y cambiamos su `name`, se mantendrá disponible mediante el nombre anterior (así como mediante el nuevo).

Esto es fácil de ver en un ejemplo:

```
<form id="form">
  <input name="login">
</form>

<script>
  alert(form.elements.login == form.login); // true, el mismo <input>

  form.login.name = "username"; // cambiamos el nombre el <input>

  // form.elements actualiza el nombre:
  alert(form.elements.login); // undefined
  alert(form.elements.username); // input

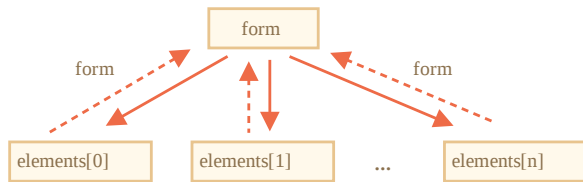
  // form permite ambos nombres: el nuevo y el viejo
  alert(form.username == form.login); // true
</script>
```

Esto usualmente no es un problema, porque raramente se cambian los nombres de los elementos de un formulario.

Referencia inversa: `element.form`

Para cualquier elemento, el formulario está disponible como `element.form`. Así que un formulario referencia todos los elementos, y los elementos referencian el formulario.

Aquí la imagen:



Por ejemplo:

```

<form id="form">
  <input type="text" name="login">
</form>

<script>
  // form -> element
  let login = form.login;

  // element -> form
  alert(login.form); // HTMLFormElement
</script>

```

Elementos del formulario

Hablemos de los controles de los formularios.

input y textarea

Podemos acceder sus valores como `input.value` (cadena) o `input.checked` (booleano) para casillas de verificación (checkboxes).

De esta manera:

```

input.value = "New value";
textarea.value = "New text";

input.checked = true; // para checkboxes o radios

```

⚠ Usa `textarea.value`, no `textarea.innerHTML`

Observa que incluso aunque `<textarea>...</textarea>` contenga su valor como HTML anidado, nunca deberíamos usar `textarea.innerHTML` para acceder a él.

Esto solo guarda el HTML que había inicialmente en la página, no su valor actual.

select y option

Un elemento `<select>` tiene 3 propiedades importantes:

1. `select.options` – la colección de subelementos `<option>`,
2. `select.value` – el valor del `<option>` seleccionado actualmente, y
3. `select.selectedIndex` – el número del `<option>` seleccionado actualmente.

Ellas proveen tres formas diferentes de asignar un valor para un elemento `<select>`:

1. Encontrar el elemento `<option>` correspondiente (por ejemplo entre `select.options`) y asignar a su `option.selected` un `true`.
2. Si conocemos un nuevo valor: Asignar tal valor a `select.value`.
3. Si conocemos el nuevo número de opción: Asignar tal número a `select.selectedIndex`.

Aquí hay un ejemplo de los tres métodos:

```

<select id="select">
  <option value="apple">Apple</option>

```

```

<option value="pear">Pear</option>
<option value="banana">Banana</option>
</select>

<script>
// las tres líneas hacen lo mismo
select.options[2].selected = true;
select.selectedIndex = 2;
select.value = 'banana';
// Recuerda que las opciones comienzan en cero, así que index 2 significa la tercera opción.
</script>

```

A diferencia de la mayoría de controles, `<select>` permite seleccionar múltiples opciones a la vez si tiene el atributo `multiple`. Esta característica es raramente utilizada.

En ese caso, necesitamos usar la primera forma: Añade/elimina la propiedad `selected` de los subelementos `<option>`.

Podemos obtener su colección como `select.options`, por ejemplo:

```

<select id="select" multiple>
  <option value="blues" selected>Blues</option>
  <option value="rock" selected>Rock</option>
  <option value="classic">Classic</option>
</select>

<script>
// obtener todos los valores seleccionados del multi-select
let selected = Array.from(select.options)
  .filter(option => option.selected)
  .map(option => option.value);

alert(selected); // blues, rock
</script>

```

La especificación completa del elemento `<select>` está disponible en la especificación <https://html.spec.whatwg.org/multipage/forms.html#the-select-element>.

new Option

En la [especificación](#) hay una sintaxis muy corta para crear elementos `<option>`:

```
option = new Option(text, value, defaultSelected, selected);
```

Esta sintaxis es opcional. Podemos usar `document.createElement('option')` y asignar atributos manualmente. Aún puede ser más corta, aquí los parámetros:

- `text` – el texto dentro del option,
- `value` – el valor del option,
- `defaultSelected` – si es `true`, entonces se le crea el atributo HTML `selected`,
- `selected` – si es `true`, el option se selecciona.

La diferencia entre `defaultSelected` y `selected` es que `defaultSelected` asigna el atributo HTML, el que podemos obtener usando `option.getAttribute('selected')`, mientras que `selected` hace que el option esté o no seleccionado.

En la práctica, uno debería usualmente establecer *ambos* valores en `true` o `false`. O simplemente omitirlos, quedarán con el predeterminado `false`.

Por ejemplo, aquí creamos un nuevo Option “unselected”:

```

let option = new Option("Text", "value");
// crea <option value="value">Text</option>

```

El mismo elemento, pero seleccionado:

```
let option = new Option("Text", "value", true, true);
```

Los elementos Option tienen propiedades:

option.selected

Es el option seleccionado.

option.index

El número del option respecto a los demás en su `<select>`.

option.text

El contenido del option (visto por el visitante).

Referencias

- Especificación: <https://html.spec.whatwg.org/multipage/forms.html> .

Resumen

Navegación de formularios:

document.forms

Un formulario está disponible como `document.forms[name/index]`.

form.elements

Los elementos del formulario están disponibles como `form.elements[name/index]`, o puedes usar solo `form[name/index]`. La propiedad `elements` también funciona para los `<fieldset>`.

element.form

Los elementos referencian a su formulario en la propiedad `form`.

El valor está disponible con `input.value`, `textarea.value`, `select.value` etc. Para checkboxes y radios, usa `input.checked` para determinar si el valor está seleccionado.

Para `<select>` también podemos obtener el valor con el índice `select.selectedIndex` o a través de la colección de opciones `select.options`.

Esto es lo básico para empezar a trabajar con formularios. Conoceremos muchos ejemplos más adelante en el tutorial.

En el siguiente capítulo vamos a hablar sobre los eventos `focus` y `blur` que pueden ocurrir en cualquier elemento, pero son manejados mayormente en formularios.

✓ Tareas

Añade una opción al select

importancia: 5

Tenemos un `<select>`:

```
<select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>
```

Utiliza JavaScript para:

1. Mostrar el valor y el texto del option seleccionado.
2. Añadir un option: `<option value="classic">Classic</option>`.
3. Seleccionarlo.

Nota, si haz hecho todo bien, tu alert debería mostrar `blues`.

[A solución](#)

Focusing: focus/blur

An element receives the focus when the user either clicks on it or uses the `Tab` key on the keyboard. There's also an `autoFocus` HTML attribute that puts the focus onto an element by default when a page loads and other means of getting the focus.

Focusing on an element generally means: "prepare to accept the data here", so that's the moment when we can run the code to initialize the required functionality.

The moment of losing the focus ("blur") can be even more important. That's when a user clicks somewhere else or presses `Tab` to go to the next form field, or there are other means as well.

Losing the focus generally means: "the data has been entered", so we can run the code to check it or even to save it to the server and so on.

There are important peculiarities when working with focus events. We'll do the best to cover them further on.

Events focus/blur

The `focus` event is called on focusing, and `blur` – when the element loses the focus.

Let's use them for validation of an input field.

In the example below:

- The `blur` handler checks if the field has an email entered, and if not – shows an error.
- The `focus` handler hides the error message (on `blur` it will be checked again):

```
<style>
  .invalid { border-color: red; }
  #error { color: red }
</style>

Your email please: <input type="email" id="input">

<div id="error"></div>

<script>
input.onblur = function() {
  if (!input.value.includes('@')) { // not email
    input.classList.add('invalid');
    error.innerHTML = 'Please enter a correct email.'
  }
};

input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // remove the "error" indication, because the user wants to re-enter something
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
</script>
```

Your email please:

Modern HTML allows us to do many validations using input attributes: `required`, `pattern` and so on. And sometimes they are just what we need. JavaScript can be used when we want more flexibility. Also we could automatically send the changed value to the server if it's correct.

Methods focus/blur

Methods `elem.focus()` and `elem.blur()` set/unset the focus on the element.

For instance, let's make the visitor unable to leave the input if the value is invalid:

```
<style>
.error {
  background: red;
}
</style>

Your email please: <input type="email" id="input">
<input type="text" style="width:220px" placeholder="make email invalid and try to focus here">

<script>
input.onblur = function() {
  if (!this.value.includes('@')) { // not email
    // show the error
    this.classList.add("error");
    // ...and put the focus back
    input.focus();
  } else {
    this.classList.remove("error");
  }
};
</script>
```

Your email please: make email invalid and try to focus he

It works in all browsers except Firefox ([bug](#)).

If we enter something into the input and then try to use `Tab` or click away from the `<input>`, then `onblur` returns the focus back.

Please note that we can't "prevent losing focus" by calling `event.preventDefault()` in `onblur`, because `onblur` works *after* the element lost the focus.

⚠ JavaScript-initiated focus loss

A focus loss can occur for many reasons.

One of them is when the visitor clicks somewhere else. But also JavaScript itself may cause it, for instance:

- An `alert` moves focus to itself, so it causes the focus loss at the element (`blur` event), and when the `alert` is dismissed, the focus comes back (`focus` event).
- If an element is removed from DOM, then it also causes the focus loss. If it is reinserted later, then the focus doesn't return.

These features sometimes cause `focus/blur` handlers to misbehave – to trigger when they are not needed.

The best recipe is to be careful when using these events. If we want to track user-initiated focus-loss, then we should avoid causing it ourselves.

Allow focusing on any element: `tabindex`

By default, many elements do not support focusing.

The list varies a bit between browsers, but one thing is always correct: `focus/blur` support is guaranteed for elements that a visitor can interact with: `<button>`, `<input>`, `<select>`, `<a>` and so on.

On the other hand, elements that exist to format something, such as `<div>`, ``, `<table>` – are unfocusable by default. The method `elem.focus()` doesn't work on them, and `focus/blur` events are never triggered.

This can be changed using HTML-attribute `tabindex`.

Any element becomes focusable if it has `tabindex`. The value of the attribute is the order number of the element when `Tab` (or something like that) is used to switch between them.

That is: if we have two elements, the first has `tabindex="1"`, and the second has `tabindex="2"`, then pressing `Tab` while in the first element – moves the focus into the second one.

The switch order is: elements with `tabindex` from 1 and above go first (in the `tabindex` order), and then elements without `tabindex` (e.g. a regular `<input>`).

Elements without matching `tabindex` are switched in the document source order (the default order).

There are two special values:

- `tabindex="0"` puts an element among those without `tabindex`. That is, when we switch elements, elements with `tabindex=0` go after elements with `tabindex ≥ 1`.

Usually it's used to make an element focusable, but keep the default switching order. To make an element a part of the form on par with `<input>`.

- `tabindex="-1"` allows only programmatic focusing on an element. The `Tab` key ignores such elements, but method `elem.focus()` works.

For instance, here's a list. Click the first item and press `Tab`:

Click the first item and press `Tab`. Keep track of the order. Please note that many subsequent `Tabs` can move the focus

```
<ul>
  <li tabindex="1">One</li>
  <li tabindex="0">Zero</li>
  <li tabindex="2">Two</li>
  <li tabindex="-1">Minus one</li>
</ul>
```

```
<style>
  li { cursor: pointer; }
  :focus { outline: 1px dashed green; }
</style>
```

Click the first item and press `Tab`. Keep track of the order. Please note that many subsequent `Tabs` can move the focus out of the `iframe` in the example.

- One
- Zero
- Two
- Minus one

The order is like this: 1 - 2 - 0. Normally, `` does not support focusing, but `tabindex` full enables it, along with events and styling with `:focus`.

The property `elem.tabIndex` works too

We can add `tabindex` from JavaScript by using the `elem.tabIndex` property. That has the same effect.

Delegation: `focusin/focusout`

Events `focus` and `blur` do not bubble.

For instance, we can't put `onfocus` on the `<form>` to highlight it, like this:

```
<!-- on focusing in the form -- add the class -->
<form onfocus="this.className='focused'">
  <input type="text" name="name" value="Name">
  <input type="text" name="surname" value="Surname">
```

```
</form>

<style> .focused { outline: 1px solid red; } </style>
```

Name	Surname
------	---------

The example above doesn't work, because when user focuses on an `<input>`, the `focus` event triggers on that input only. It doesn't bubble up. So `form.onfocus` never triggers.

There are two solutions.

First, there's a funny historical feature: `focus/blur` do not bubble up, but propagate down on the capturing phase.

This will work:

```
<form id="form">
  <input type="text" name="name" value="Name">
  <input type="text" name="surname" value="Surname">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  // put the handler on capturing phase (last argument true)
  form.addEventListener("focus", () => form.classList.add('focused'), true);
  form.addEventListener("blur", () => form.classList.remove('focused'), true);
</script>
```

Name	Surname
------	---------

Second, there are `focusin` and `focusout` events – exactly the same as `focus/blur`, but they bubble.

Note that they must be assigned using `elem.addEventListener`, not `on<event>`.

So here's another working variant:

```
<form id="form">
  <input type="text" name="name" value="Name">
  <input type="text" name="surname" value="Surname">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  form.addEventListener("focusin", () => form.classList.add('focused'));
  form.addEventListener("focusout", () => form.classList.remove('focused'));
</script>
```

Name	Surname
------	---------

Summary

Events `focus` and `blur` trigger on an element focusing/losing focus.

Their specials are:

- They do not bubble. Can use capturing state instead or `focusin/focusout`.
- Most elements do not support focus by default. Use `tabindex` to make anything focusable.

The current focused element is available as `document.activeElement`.

✓ Tareas

Editable div

importancia: 5

Create a `<div>` that turns into `<textarea>` when clicked.

The textarea allows to edit the HTML in the `<div>`.

When the user presses `Enter` or it loses focus, the `<textarea>` turns back into `<div>`, and its content becomes HTML in `<div>`.

[Demo en nueva ventana](#) ↗

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Edit TD on click

importancia: 5

Make table cells editable on click.

- On click – the cell should become “editable” (textarea appears inside), we can change HTML. There should be no resize, all geometry should remain the same.
- Buttons OK and CANCEL appear below the cell to finish/cancel the editing.
- Only one cell may be editable at a moment. While a `<td>` is in “edit mode”, clicks on other cells are ignored.
- The table may have many cells. Use event delegation.

The demo:

Click on a table cell to edit it. Press OK or CANCEL when you finish.

Bagua Chart: Direction, Element, Color, Meaning

Northwest Metal Silver Elders	North Water Blue Change	Northeast Earth Yellow Direction
West Metal Gold Youth	Center All Purple Harmony	East Wood Blue Future
Southwest Earth Brown Tranquility	South Fire Orange Fame	Southeast Wood Green Romance

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Keyboard-driven mouse

importancia: 4

Focus on the mouse. Then use arrow keys to move it:

[Demo en nueva ventana](#) ↗

P.S. Don't put event handlers anywhere except the `#mouse` element.

P.P.S. Don't modify HTML/CSS, the approach should be generic and work with any element.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Eventos: change, input, cut, copy, paste

Veamos varios eventos que acompañan la actualización de datos.

Evento: change

El evento `change` se activa cuando el elemento finaliza un cambio.

Para ingreso de texto significa que el evento ocurre cuando se pierde foco en el elemento.

Por ejemplo, mientras estamos escribiendo en el siguiente cuadro de texto – no hay evento. Pero cuando movemos el focus (enfoque) a otro lado, por ejemplo hacemos click en un botón, entonces ocurre el evento `change` :

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">
```


Para otros elementos: `select`, `input type=checkbox/radio` ocurre justo después de cambiar la opción seleccionada:

```
<select onchange="alert(this.value)">
  <option value="">Select something</option>
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
  <option value="3">Option 3</option>
</select>
```

Evento: input

El evento `input` ocurre cada vez que un valor es modificado por el usuario.

A diferencia de los eventos de teclado, ocurre con el cambio a cualquier valor, incluso aquellos no que involucran acciones de teclado: copiar/pegar con el mouse o usar reconocimiento de voz para dictar texto.

Por ejemplo:

```
<input type="text" id="input" oninput: <span id="result"></span>
<script>
  input.oninput = function() {
    result.innerHTML = input.value;
  };
</script>
```

 oninput:

Si queremos manejar cualquier modificación en un `<input>` entonces este evento es la mejor opción.

Por otro lado, el evento `input` no se activa con entradas del teclado u otras acciones que no involucren modificar un valor, e.g presionar las flechas de dirección `←` `→` mientras se está en el input.

No se puede evitar nada en `oninput`

El evento `input` ocurre después de que el valor es modificado.

Por lo tanto no podemos usar `event.preventDefault()` para evitar modificaciones, en este caso ya es demasiado tarde por lo que no habría efecto.

Eventos: cut, copy, paste

Estos eventos ocurren al cortar/copiar/pegar un valor.

Estos pertenecen a la clase [ClipboardEvent](#) y dan acceso a los datos copiados/pegados.

También podemos usar `event.preventDefault()` para cancelar la acción y que nada sea copiado/pegado. Por ejemplo, el siguiente código evita tales eventos y muestra qué es lo que estamos intentando cortar/copiar/pegar:

```
<input type="text" id="input">
<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));
    return false;
  };
</script>
```

Por favor ten en cuenta que no solo es posible copiar/pegar texto, sino cualquier cosa. Por ejemplo, podemos copiar un archivo en el gestor de archivos del SO y pegarlo.

Esto es porque `clipboardData` implementa la interfaz `DataTransfer`, usada comúnmente para “arrastrar y soltar” y “copiar y pegar”. Ahora esto está fuera de nuestro objetivo, pero puedes encontrar sus métodos [en la especificación](#).

ClipboardAPI: restricciones para seguridad del usuario

El portapapeles es algo a nivel “global” del SO. Por cuestiones de seguridad, la mayoría de los navegadores dan acceso al portapapeles solamente bajo determinadas acciones del usuario, por ejemplo al manejar eventos `onclick`.

Además está prohibido generar eventos “personalizados” del portapapeles con `dispatchEvent` en todos los navegadores excepto Firefox.

Resumen

Eventos de modificación de datos:

Evento	Descripción	Especiales
<code>change</code>	Un valor fue cambiado.	Para ingreso de texto, ocurre al perderse el enfoque
<code>input</code>	Para entrada de texto en cada cambio	Ocurre inmediatamente a diferencia de <code>change</code> .
<code>cut/copy/paste</code>	Acciones cortar/copiar/pegar	La acción puede ser cancelada. La propiedad <code>event.clipboardData</code> brinda acceso a leer/escribir del portapeles.

Tareas

Calculadora de depósito

importancia: 5

Crea una interfaz que permita ingresar una suma de depósito bancario y porcentaje, luego calcula cuánto será después de un periodo de tiempo determinado.

Acá una demostración:

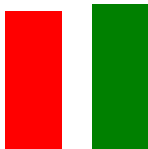
Calculadora de depósito.

Depósito inicial

¿Cuántos meses?

¿Interés anual?

Inicial:	Final:
10000	10500



Cualquier modificación debe ser procesada de inmediato.

La fórmula es:

```
// initial: la suma inicial de dinero
// interest: e.g. 0.05 significa 5% anual
// years: cuántos años esperar
let result = Math.round(initial * (1 + interest * years));
```

[Abrir un entorno controlado para la tarea.](#)

[A solución](#)

Formularios: evento y método submit

El evento `submit` se activa cuando el formulario es enviado, normalmente se utiliza para validar el formulario antes de ser enviado al servidor o bien para abortar el envío y procesarlo con JavaScript.

El método `form.submit()` permite iniciar el envío del formulario mediante JavaScript. Podemos utilizarlo para crear y enviar nuestros propios formularios al servidor.

Veamos más detalles sobre ellos.

Evento: submit

Mayormente un formulario puede enviarse de dos maneras:

1. La primera – Haciendo click en `<input type="submit">` o en `<input type="image">`.
2. La segunda – Pulsando la tecla `Enter` en un campo del formulario.

Ambas acciones causan que el evento `submit` sea activado en el formulario. El handler puede comprobar los datos, y si hay errores, mostrarlos e invocar `event.preventDefault()`, entonces el formulario no será enviado al servidor.

En el formulario de abajo:

1. Ve al campo tipo texto y pulsa la tecla `Enter`.
2. Haz click en `<input type="submit">`.

Ambas acciones muestran `alert` y el formulario no es enviado debido a la presencia de `return false`:

```
<form onsubmit="alert('submit!');return false">
  Primero: Enter en el campo de texto <input type="text" value="texto"><br>
```


Segundo: Click en "submit": `<input type="submit" value="Submit">`
`</form>`

Primero: Enter en el campo de texto

Segundo: Click en "submit":

i Relación entre `submit` y `click`

Cuando un formulario es enviado utilizando `Enter` en un campo tipo texto, un evento `click` se genera en el `<input type="submit">`

Muy curioso, dado que no hubo ningún click en absoluto.

Aquí esta la demo:

```
<form onsubmit="return false">
  <input type="text" size="30" value="Sitúa el cursor aquí y pulsa Enter">
  <input type="submit" value="Submit" onclick="alert('click')">
</form>
```

Método: `submit`

Para enviar un formulario al servidor manualmente, podemos usar `form.submit()`.

Entonces el evento `submit` no será generado. Se asume que si el programador llama `form.submit()`, entonces el script ya realizó todo el procesamiento relacionado.

A veces es usado para crear y enviar un formulario manualmente, como en este ejemplo:

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// el formulario debe estar en el document para poder enviarlo
document.body.append(form);

form.submit();
```

✓ Tareas

Formulario modal

importancia: 5

Crea una función `showPrompt(html, callback)` que muestre un formulario con el mensaje `html`, un campo `input` y botones `OK/CANCELAR`.

- Un usuario debe escribir algo en el campo de texto y pulsar `Enter` o el botón OK, entonces `callback(value)` es llamado con el valor introducido.
- En caso contrario, si el usuario pulsa `Esc` o CANCELAR, entonces `callback(null)` es llamado.

En ambos casos se finaliza el proceso se y borra el formulario.

Requisitos:

- El formulario debe estar en el centro de la ventana.

- El formulario es *modal*. Es decir que no habrá interacción con el resto de la página, siempre que sea posible, hasta que el usuario lo cierre.
- Cuando se muestra el formulario, el foco debe estar en el `<input>` del usuario.
- Las teclas `Tab` / `Shift+Tab` deben alternar el foco entre los diferentes campos del formulario, no se permite cambiar el foco a otros elementos de la página.

Ejemplo de uso:

```
showPrompt("Escribe algo<br>...inteligente :)", function(value) {
  alert(value);
});
```

Demo en el iframe:

Pulsa el botón de abajo

Click aquí para mostrar el formulario

P.S. El código fuente tiene el HTML/CSS para el formulario con posición fija. Pero tú decides cómo haces el modal.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

El documento y carga de recursos

Page: DOMContentLoaded, load, beforeunload, unload

El ciclo de vida de una página HTML tiene tres eventos importantes:

- `DOMContentLoaded` – el navegador HTML está completamente cargado y el árbol DOM está construido, pero es posible que los recursos externos como `` y hojas de estilo aún no se hayan cargado.
- `load` – no solo se carga el HTML, sino también todos los recursos externos: imágenes, estilos, etc.
- `beforeunload/unload` – el usuario sale de la página.

Cada evento puede ser útil:

- Evento `DOMContentLoaded` – DOM está listo, por lo que el controlador puede buscar nodos DOM, inicializar la interfaz.
- Evento `load` – se cargan recursos externos, por lo que se aplican estilos, se conocen tamaños de imagen, etc.
- Evento `beforeunload` – el usuario se va: podemos comprobar si el usuario guardó los cambios y preguntarle si realmente quiere irse.
- Evento `unload` – el usuario casi se fue, pero aún podemos iniciar algunas operaciones, como enviar estadísticas.

Explore los detalles de estos eventos.

DOMContentLoaded

El evento `DOMContentLoaded` ocurre en el objeto `document`.

Debemos usar `addEventListener` para capturarlo:

```
document.addEventListener("DOMContentLoaded", ready);
// no "document.onDOMContentLoaded = ..."
```

Por ejemplo:

```
<script>
function ready() {
  alert('DOM is ready');

  // la imagen aún no está cargada (a menos que se haya almacenado en caché), por lo que el tamaño es 0x0
  alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
}

document.addEventListener("DOMContentLoaded", ready);
</script>


```

En el ejemplo, el controlador del evento `DOMContentLoaded` se ejecuta cuando el documento está cargado, por lo que puede ver todos los elementos, incluido el `` que está después de él.

Pero no espera a que se cargue la imagen. Entonces, `alert` muestra los tamaños en cero.

A primera vista, el evento `DOMContentLoaded` es muy simple. El árbol DOM está listo – aquí está el evento. Sin embargo, hay algunas peculiaridades.

DOMContentLoaded y scripts

Cuando el navegador procesa un documento HTML y se encuentra con una etiqueta `<script>`, debe ejecutarla antes de continuar construyendo el DOM. Esa es una precaución, ya que los scripts pueden querer modificar el DOM, e incluso hacer `document.write` en él, por lo que `DOMContentLoaded` tiene que esperar.

Entonces `DOMContentLoaded` siempre ocurre después de tales scripts:

```
<script>
document.addEventListener("DOMContentLoaded", () => {
  alert("DOM listo!");
});
</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<script>
  alert("Librería cargada, línea de script ejecutada");
</script>
```

En el ejemplo anterior, primero vemos “Biblioteca cargada ...” y luego “¡DOM listo!” (se ejecutan todos los scripts).

⚠ Scripts que no bloquean DOMContentLoaded

Hay dos excepciones a esta regla:

1. Scripts con el atributo `async`, que cubriremos [un poco más tarde](#), no bloquea el `DOMContentLoaded`.
2. Los scripts que se generan dinámicamente con `document.createElement('script')` y luego se agregan a la página web, tampoco bloquean este evento.

DOMContentLoaded y estilos

Las hojas de estilo externas no afectan a DOM, por lo que `DOMContentLoaded` no las espera.

Pero hay una trampa. Si tenemos un script después del estilo, entonces ese script debe esperar hasta que se cargue la hoja de estilo:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // el script no se ejecuta hasta que se cargue la hoja de estilo
  alert(getComputedStyle(document.body).marginTop);
</script>
```

La razón de esto es que el script puede querer obtener coordenadas y otras propiedades de elementos dependientes del estilo, como en el ejemplo anterior. Naturalmente, tiene que esperar a que se carguen los estilos.

Como `DOMContentLoaded` espera los scripts, ahora también espera a los estilos que están antes que ellos.

Autocompletar del navegador integrado

Firefox, Chrome y Opera autocompletan formularios en `DOMContentLoaded`.

Por ejemplo, si la página tiene un formulario con nombre de usuario y contraseña, y el navegador recuerda los valores, entonces en `DOMContentLoaded` puede intentar completarlos automáticamente (si el usuario lo aprueba).

Entonces, si `DOMContentLoaded` es pospuesto por scripts de largo tiempo de carga, el autocompletado también espera. Probablemente haya visto eso en algunos sitios (si usa la función de autocompletar del navegador): los campos de inicio de sesión/contraseña no se autocompletan inmediatamente, sino con retraso hasta que la página se carga por completo. En realidad es el retraso hasta el evento `DOMContentLoaded`.

window.onload

El evento `load` en el objeto `window` se activa cuando se carga toda la página, incluidos estilos, imágenes y otros recursos. Este evento está disponible a través de la propiedad `onload`.

El siguiente ejemplo muestra correctamente los tamaños de las imágenes, porque `window.onload` espera todas las imágenes:

```
<script>
window.onload = function() { // similar a window.addEventListener('load', (event) => {
    alert('Página cargada');

    // la imagen es cargada al mismo tiempo
    alert(`Tamaño de imagen: ${img.offsetWidth}x${img.offsetHeight}`);
  }
};
</script>


```

window.onunload

Cuando un visitante abandona la página, el evento `unload` se activa en `window`. Podemos hacer algo allí que no implique un retraso, como cerrar ventanas emergentes relacionadas.

La excepción notable es el envío de análisis.

Supongamos que recopilamos datos sobre cómo se usa la página: clicks del mouse, desplazamientos, áreas de página visitadas, etc.

Naturalmente, el evento `unload` sucede cuando el usuario nos deja y nos gustaría guardar los datos en nuestro servidor.

Existe un método especial `navigator.sendBeacon(url, data)` para tales necesidades, descrito en la especificación <https://w3c.github.io/beacon/>.

Este envía los datos en segundo plano. La transición a otra página no se retrasa: el navegador abandona la página, pero aún realiza `sendBeacon`.

Así es como se usa:

```
let analyticsData = { /* objeto con datos recopilados */ };

window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
});
```

- La solicitud se envía como POST.
- Podemos enviar no solo una cadena, sino también formularios y otros formatos, como se describe en el capítulo <info: fetch>, pero generalmente es un objeto string.

- Los datos están limitados por 64 kb.

Cuando finaliza la solicitud `sendBeacon`, es probable que el navegador ya haya abandonado el documento, por lo que no hay forma de obtener la respuesta del servidor (que suele estar vacía para análisis).

También hay una bandera `keepalive` para hacer tales solicitudes “after-page-left” en el método `[fetch](info: fetch)` para solicitudes de red genéricas. Puede encontrar más información en el capítulo [<info: fetch-api>](#).

Si queremos cancelar la transición a otra página, no podemos hacerlo aquí. Pero podemos usar otro evento: `onbeforeunload`.

window.onbeforeunload

Si un visitante inició la navegación fuera de la página o intenta cerrar la ventana, el controlador `beforeunload` solicita una confirmación adicional.

Si cancelamos el evento, el navegador puede preguntar al visitante si está seguro.

Puede probarlo ejecutando este código y luego recargando la página:

```
window.onbeforeunload = function() {  
  return false;  
};
```

Por razones históricas, devolver una cadena no vacía también cuenta como cancelar el evento. Hace algún tiempo, los navegadores solían mostrarlo como un mensaje, pero como dice la [especificación moderna](#), no deberían.

Aquí hay un ejemplo:

```
window.onbeforeunload = function() {  
  return "Hay cambios sin guardar. ¿Salir ahora?";  
};
```

El comportamiento se modificó, porque algunos webmasters abusaron de este controlador de eventos mostrando mensajes engañosos y molestos. Entonces, en este momento, los navegadores antiguos aún pueden mostrarlo como un mensaje, pero aparte de eso, no hay forma de personalizar el mensaje que se muestra al usuario.

readyState

¿Qué sucede si configuramos el controlador `DOMContentLoaded` después de cargar el documento?

Naturalmente, nunca se ejecutará.

Hay casos en los que no estamos seguros de si el documento está listo o no. Nos gustaría que nuestra función se ejecute cuando se cargue el DOM, ya sea ahora o más tarde.

La propiedad `document.readyState` nos informa sobre el estado de carga actual.

Hay 3 valores posibles:

- `"loading"` – el documento se está cargando.
- `"interactive"` – el documento fue leído por completo.
- `"complete"` – el documento se leyó por completo y todos los recursos (como imágenes) también se cargaron.

Entonces podemos verificar `document.readyState` y configurar un controlador o ejecutar el código inmediatamente si está listo.

Como esto:

```
function work() { /*...*/ }  
  
if (document.readyState === 'loading') {  
  // cargando todavía, esperar el evento  
  document.addEventListener('DOMContentLoaded', work);  
}
```

```
} else {  
  // DOM está listo!  
  work();  
}
```

También existe el evento `readystatechange` que se activa cuando cambia el estado, por lo que podemos imprimir todos estos estados así:

```
// estado actual  
console.log(document.readyState);  
  
//imprimir los cambios de estado  
document.addEventListener('readystatechange', () => console.log(document.readyState));
```

El evento `readystatechange` es una mecánica alternativa para rastrear el estado de carga del documento, apareció hace mucho tiempo. Hoy en día, rara vez se usa.

Veamos el flujo de eventos completo para ver si están completados.

Aquí hay un documento con `<iframe>`, `` y controladores que registran eventos:

```
<script>  
  log('initial readyState:' + document.readyState);  
  
  document.addEventListener('readystatechange', () => log('readyState:' + document.readyState));  
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));  
  
  window.onload = () => log('window onload');  
</script>  
  
<iframe src="iframe.html" onload="log('iframe onload')"></iframe>  
  
  
<script>  
  img.onload = () => log('img onload');  
</script>
```

El ejemplo práctico está [en el sandbox](#) .

La salida típica:

1. [1] readyState inicial: loading
2. [2] readyState: interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState: complete
7. [4] window onload

Los números entre corchetes denotan el tiempo aproximado en el que ocurre. Los eventos etiquetados con el mismo dígito ocurren aproximadamente al mismo tiempo (+ – unos pocos ms).

- `document.readyState` se convierte en `interactive` justo antes de `DOMContentLoaded` . Estas dos cosas realmente significan lo mismo.
- `document.readyState` se convierte en `complete` cuando se cargan todos los recursos (`iframe` e `img`). Aquí podemos ver que ocurre aproximadamente al mismo tiempo que `img.onload` (`img` es el último recurso) y `window.onload` . Cambiar al estado `complete` significa lo mismo que “`window.onload`”. La diferencia es que `window.onload` siempre funciona después de todos los demás controladores `load` .

Resumen

Eventos de carga de página:

- El evento `DOMContentLoaded` se activa en el `document` cuando el DOM está listo. Podemos aplicar JavaScript a elementos en esta etapa.
 - Secuencias de comandos como `<script> ... </script>` o `<script src = " ... " > </script>` bloquean `DOMContentLoaded`, el navegador espera a que se ejecuten.
 - Las imágenes y otros recursos también pueden seguir cargándose.
- El evento `load` en `window` se activa cuando se cargan la página y todos los recursos. Rara vez lo usamos, porque generalmente no hay necesidad de esperar tanto.
- El evento `beforeunload` en `window` se activa cuando el usuario quiere salir de la página. Si cancelamos el evento, el navegador pregunta si el usuario realmente quiere irse (por ejemplo, tenemos cambios sin guardar).
- El evento `unload` en `window` se dispara cuando el usuario finalmente se está yendo, en el controlador solo podemos hacer cosas simples que no impliquen demoras o preguntas al usuario. Debido a esa limitación, rara vez se usa. Podemos enviar una solicitud de red con `navigator.sendBeacon`.
- `document.readyState` es el estado actual del documento, los cambios se pueden rastrear con el evento `readystatechange`:
 - `loading` – el documento está cargando.
 - `interactive` – el documento se analiza, ocurre aproximadamente casi al mismo tiempo que `DOMContentLoaded`, pero antes.
 - `complete` – el documento y los recursos se cargan, ocurre aproximadamente casi al mismo tiempo que `window.onload`, pero antes.

Scripts: async, defer

En los sitios web modernos los scripts suelen ser más “pesados” que el HTML, el tamaño de la descarga es grande y el tiempo de procesamiento es mayor.

Cuando el navegador carga el HTML y se encuentra con una etiqueta `<script>...</script>`, no puede continuar construyendo el DOM. Debe ejecutar el script en el momento. Lo mismo sucede con los scripts externos `<script src="..."></script>`, el navegador tiene que esperar hasta que el script sea descargado, ejecutarlo y solo después procesa el resto de la página.

Esto nos lleva a dos importantes problemas:

1. Los scripts no pueden ver los elementos del DOM que se encuentran debajo de él por lo que no pueden agregar controladores de eventos, etc.
2. Si hay un script muy pesado en la parte superior de la página, este “bloquea la página”. Los usuarios no pueden ver el contenido de la página hasta que sea descargado y ejecutado.

```
<p>...contenido previo al script...</p>

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- Esto no es visible hasta que el script sea cargado -->
<p>...contenido posterior al script...</p>
```

Hay algunas soluciones para eso. Por ejemplo podemos poner el script en la parte inferior de la página por lo que podrá ver los elementos sobre él y no bloqueará la visualización del contenido de la página.

```
<body>
  ...todo el contenido está arriba del script...

  <script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
</body>
```

Pero esta solución está lejos de ser perfecta. Por ejemplo el navegador solo se dará cuenta del script (y podrá empezar a descargarlo) después de descargar todo el documento HTML. Para documentos HTML extensos eso puede ser un retraso notable.

Este tipo de cosas son imperceptibles para las personas que usan conexiones muy rápidas, pero muchas personas en el mundo todavía tienen velocidades de internet lentas y utilizan una conexión de internet móvil que está lejos de ser

perfecta.

Afortunadamente hay dos atributos de `<script>` que resuelven ese problema para nosotros: `defer` y `async`.

defer

El atributo `defer` indica al navegador que no espere por el script. En lugar de ello, debe seguir procesando el HTML, construir el DOM. El script carga “en segundo plano” y se ejecuta cuando el DOM esta completo.

Aquí está el mismo ejemplo de arriba pero con `defer`:

```
<p>...contenido previo script...</p>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- Inmediatamente visible -->
<p>...contenido posterior al script...</p>
```

En otras palabras:

- Los scripts con `defer` nunca bloquean la página.
- Los scripts con `defer` siempre se ejecutan cuando el DOM esta listo (pero antes del evento `DOMContentLoaded`).

Los siguientes ejemplos demuestran la segunda parte:

```
<p>...contenido previo a los scripts...</p>

<script>
  document.addEventListener('DOMContentLoaded', () => alert("¡DOM listo después del defer!"));
</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<p>...contenido posterior a los scripts...</p>
```

1. El contenido de la página se muestra inmediatamente.
2. `DOMContentLoaded` espera por el script diferido. Solo se dispara cuando el script es descargado y ejecutado.

******Los scripts diferidos mantienen su orden relativo, tal cual los scripts regulares.

Digamos que tenemos dos scripts diferidos, `long.js` (largo) y luego `small.js` (corto):

```
<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>
<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>
```

Los navegadores analizan la página en busca de scripts y los descarga en paralelo para mejorar el rendimiento. Entonces en el ejemplo superior ambos scripts se descargan en paralelo, el `small.js` probablemente lo haga primero.

...Pero el atributo `defer`, además de decirle al navegador “no bloquear”, asegura que el orden relativo se mantenga. Entonces incluso si `small.js` se carga primero, aún espera y se ejecuta después de `long.js`.

Por ello es importante para casos donde necesitamos cargar una librería JavaScript y entonces un script que depende de ella.

i El atributo `defer` es solo para scripts externos

El atributo `defer` es ignorado si el `<script>` no tiene el atributo `src`.

async

El atributo `async` es de alguna manera como `defer`. También hace el script no bloqueante. Pero tiene importantes diferencias de comportamiento.

El atributo `async` significa que el script es completamente independiente:

- El navegador no se bloquea con scripts `async` (como `defer`).
- Otros scripts no esperan por scripts `async`, y scripts `async` no espera por ellos.
- `DOMContentLoaded` y los scripts asincrónicos no se esperan entre sí:
 - `DOMContentLoaded` puede suceder antes que un script asincrónico (si un script asincrónico termina de cargar una vez la página está completa)
 - ...o después de un script asincrónico (si tal script asincrónico es pequeño o está en cache)

En otras palabras, los scripts `async` cargan en segundo plano y se ejecutan cuando están listos. El DOM y otros scripts no esperan por ellos, y ellos no esperan por nada. Un script totalmente independiente que se ejecuta en cuanto se ha cargado. Tan simple como es posible, ¿cierto?

Aquí hay un ejemplo similar al que vimos con `defer`: Dos scripts `long.js` y `small.js`, pero ahora con `async` en lugar de `defer`.

Los unos no esperan por lo otros. El que cargue primero (probablemente `small.js`), se ejecuta primero.

```
<p>...contenido previo a los scripts...</p>

<script>
  document.addEventListener('DOMContentLoaded', () => alert("¡DOM listo!"));
</script>

<script async src="https://javascript.info/article/script-async-defer/long.js"></script>
<script async src="https://javascript.info/article/script-async-defer/small.js"></script>

<p>...contenido posterior a los scripts...</p>
```

- El contenido de la página se muestra inmediatamente: `async` no lo bloquea.
- El evento `DOMContentLoaded` puede suceder antes o después de `async`, no hay garantías aquí.
- Un script más pequeño `small.js` que esté segundo probablemente cargue antes que uno más largo `long.js`, entonces se ejecutará primero. Aunque podría ser que `long.js` cargue primero si está en caché y ejecute primero. A eso lo llamamos “load-first order”, se ejecuta primero el que cargue antes.

Los scripts asincrónicos son excelentes cuando incluimos scripts de terceros (contadores, anuncios, etc) en la página debido a que ellos no dependen de nuestros scripts y nuestros scripts no deberían esperar por ellos.

```
<!-- Google Analytics is usually added like this -->
<script async src="https://google-analytics.com/analytics.js"></script>
```

Scripts dinámicos

Hay otra manera importante de agregar un script a la página.

Podemos crear un script y agregarlo dinámicamente al documento usando JavaScript:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
document.body.append(script); // (*)
```

El script comienza a cargar tan pronto como es agregado al documento `(*)`.

Los scripts dinámicos se comportan como `async` por defecto

Esto es:

- Ellos no esperan a nadie y nadie espera por ellos.

- El script que carga primero se ejecuta primero (`load-first order`)

Esto puede ser cambiado si explícitamente establecemos `script.async=false` . Así los scripts serán ejecutados en el orden del documento, tal como en `defer` .

En este ejemplo, la función `loadScript(src)` añade un script y también establece `async` a `false` .

Entonces `long.js` siempre ejecuta primero (por haber sido agregado primero):

```
function loadScript(src) {  
  let script = document.createElement('script');  
  script.src = src;  
  script.async = false;  
  document.body.append(script);  
}  
  
// long.js se ejecuta primero a causa del async=false  
loadScript("/article/script-async-defer/long.js");  
loadScript("/article/script-async-defer/small.js");
```

Sin `script.async=false` , los scripts se ejecutarían de forma predeterminada, en el orden de carga primero (probablemente `small.js` primero).

De nuevo, como con `defer` , el orden importa si queremos cargar una librería y luego otro script que depende de ella.

Resumen

Ambos, `async` y `defer` , tienen algo en común: la descarga de tales scripts no bloquean el renderizado de la página. Por lo cual el usuario puede leer el contenido de la página y familiarizarse con la página inmediatamente.

Pero hay algunas diferencias esenciales entre ellos:

	Orden	DOMContentLoaded
<code>async</code>	<i>Load-first order.</i> El orden del documento no importa. El que carga primero ejecuta primero	Irrelevante. Puede cargar y ejecutarse mientras el documento no ha sido completamente descargado, eso puede pasar si el script es pequeño o está en cache y el documento es suficientemente extenso.
<code>defer</code>	<i>Document order</i> (como van en el documento).	Ejecutan después de que el documento es cargado y analizado (espera si es necesario), justo antes de <code>DOMContentLoaded</code> .

En la práctica, `defer` es usado para scripts que necesitan todo el DOM y/o el orden de ejecución es importante.

Y `async` es usado para scripts independientes, como contadores y anuncios donde el orden de ejecución no importa.



La página sin scripts debe ser utilizable

Por favor ten en cuenta que si estas usando `defer` o `async` , el usuario verá la página antes de que el script sea cargado.

En tal caso algunos componentes gráficos probablemente no estén listos.

No olvides poner alguna señal de “cargando” y los deshabilitar botones que aún no estén funcionando. Permite al usuario ver claramente qué puede hacer en la página y qué está listo y qué no.

Carga de recursos: onload y onerror

El navegador nos permite hacer seguimiento de la carga de recursos externos: scripts, iframes, imágenes y más.

Hay dos eventos para eso:

- `onload` – cuando cargó exitosamente,
- `onerror` – cuando un error ha ocurrido.

Cargando un script

Digamos que tenemos que cargar un script de terceros y llamar una función que se encuentra dentro.

Podemos cargarlo dinámicamente de esta manera:

```
let script = document.createElement("script");
script.src = "my.js";

document.head.append(script);
```

...pero ¿cómo podemos ejecutar la función que esta dentro del script? Necesitamos esperar hasta que el script haya cargado, y solo después podemos llamarlo.

Por favor tome nota:

Para nuestros scripts podemos usar [JavaScript modules](#) aquí, pero no está adoptado ampliamente por bibliotecas de terceros.

script.onload

El evento `load` se dispara después de que script sea cargado y ejecutado.

Por ejemplo:

```
let script = document.createElement('script');

// podemos cargar cualquier script desde cualquier dominio
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"
document.head.append(script);

script.onload = function() {
  // el script crea una variable "_"
  alert( _.VERSION ); // muestra la versión de la librería
};
```

Entonces en `onload` podemos usar variables, ejecutar funciones, etc.

...¿y si la carga falla? Por ejemplo: no hay tal script (error 404) en el servidor o el servidor está caído (no disponible).

script.onerror

Los errores que ocurren durante la carga de un script pueden ser rastreados en el evento `error`.

Por ejemplo, hagamos una petición a un script que no existe:

```
let script = document.createElement('script');
script.src = "https://example.com/404.js"; // no hay tal script
document.head.append(script);

script.onerror = function() {
  alert("Error al cargar " + this.src); // Error al cargar https://example.com/404.js
};
```

Por favor nota que como no podemos obtener detalles del error HTTP aquí, no podemos saber if fue un error 404 o algo diferente. Solo el error de carga.

Importante:

Los eventos `onload/onerror` rastrean solamente la carga de ellos mismos.

Los errores que pueden ocurrir durante el procesamiento y ejecución están fuera del alcance para esos eventos. Eso es: si un script es cargado de manera exitosa, incluso si tiene errores de programación adentro, el evento `onload` se dispara. Para rastrear los errores del script un puede usar el manejador global `window.onerror`;

Otros recursos

Los eventos `load` y `error` también funcionan para otros recursos, básicamente para cualquiera que tenga una `src` externa.

Por ejemplo:

```
let img = document.createElement("img");
img.src = "https://js.cx/clipart/train.gif"; // (*)

img.onload = function () {
  alert(`Image loaded, size ${img.width}x${img.height}`);
};

img.onerror = function () {
  alert("Error occurred while loading image");
};
```

Sin embargo, hay algunas notas:

- La mayoría de recursos empiezan a cargarse cuando son agregados al documento. Pero `` es una excepción, comienza la carga cuando obtiene una fuente `“src”` `(*)`.
- Para `<iframe>`, el evento `iframe.onload` se dispara cuando el `iframe` ha terminado de cargar, tanto para una carga exitosa como en caso de un error.

Esto es por razones históricas.

Política de origen cruzado

Hay una regla: los scripts de un sitio no pueden acceder al contenido de otro sitio. Por ejemplo: un script de `https://facebook.com` no puede leer la bandeja de correos del usuario en `https://gmail.com`.

O para ser más precisos, un origen (el trío dominio/puerto/protocolo) no puede acceder al contenido de otro. Entonces, incluso si tenemos un sub-dominio o solo un puerto distinto, son considerados orígenes diferentes sin acceso al otro.

Esta regla también afecta a recursos de otros dominios.

Si usamos un script de otro dominio y tiene un error, no podemos obtener detalles del error.

Por ejemplo, tomemos un script `error.js` que consta de una sola llamada a una función (con errores).

```
// error.js
noSuchFunction();
```

Ahora cargalo desde el mismo sitio donde esta alojado:

```
<script>
  window.onerror = function (message, url, line, col, errorObj) {
    alert(`${message}\n${url}, ${line}:${col}`);
  };
</script>
<script src="/article/onload-onerror/crossorigin/error.js"></script>
```

Podemos ver un buen reporte de error, como este:

```
Uncaught ReferenceError: noSuchFunction is not defined
https://javascript.info/article/onload-onerror/crossorigin/error.js, 1:1
```

Ahora carguemos el mismo script desde otro dominio:

```
<script>
  window.onerror = function (message, url, line, col, errorObj) {
    alert(`${message}\n${url}, ${line}:${col}`);
  };
</script>
<script src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>
```

El reporte es diferente, como este:

```
Script error.
, 0:0
```

Los detalles pueden variar dependiendo del navegador, pero la idea es la misma: cualquier información sobre las partes internas de un script, incluyendo el rastreo de la pila de errores, se oculta. Exactamente porque es de otro dominio.

¿Por qué necesitamos detalles de error?

Hay muchos servicios (y podemos construir uno nuestro) que escuchan los errores globales usando `window.onerror`, guardan los errores y proveen una interfaz para acceder a ellos y analizarlos. Eso es grandioso ya que podemos ver los errores originales ocasionados por nuestros usuarios. Pero si el script viene desde otro origen no hay mucha información sobre los errores como acabamos de ver.

También se aplican políticas similares de origen cruzado (CORS) a otros tipos de recursos.

Para permitir el acceso de origen cruzado, la etiqueta `<script>` necesita tener el atributo `crossorigin`, además el servidor remoto debe proporcionar cabeceras especiales.

Hay 3 niveles de acceso de origen cruzado:

1. **Sin el atributo `crossorigin`** – acceso prohibido.
2. **`crossorigin="anonymous"`** – acceso permitido si el servidor responde con la cabecera `Access-Control-Allow-Origin` con `*` o nuestro origen. El navegador no envía la información de la autorización y cookies al servidor remoto.
3. **`crossorigin="use-credentials"`** – acceso permitido si el servidor envía de vuelta la cabecera `Access-Control-Allow-Origin` con nuestro origen y `Access-Control-Allow-Credentials: true`. El navegador envía la información de la autorización y las cookies al servidor remoto.

i Por favor tome nota:

Puedes leer más sobre accesos de origen cruzado en el capítulo [Fetch: Cross-Origin Requests](#). Este describe el método `fetch` para requerimientos de red, pero la política es exactamente la misma.

Cosas como las “cookies” están fuera de nuestro alcance, pero podemos leer sobre ellas en [Cookies](#), [document.cookie](#).

En nuestro caso no teníamos ningún atributo de origen cruzado (`cross-origin`). Por lo que se prohibió el acceso de origen cruzado. Vamos a agregarlo.

Podemos elegir entre `"anonymous"` (no se envían las cookies, una sola cabecera es necesaria en el lado del servidor) y `"use-credentials"` (envía las cookies, dos cabeceras son necesarias en el lado del servidor).

Si no nos importan las `cookies`, entonces `"anonymous"` es el camino a seguir:

```
<script>
  window.onerror = function(message, url, line, col, errorObj) {
    alert(`${message}\n${url}, ${line}:${col}`);
  };
</script>
<script crossorigin="anonymous" src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>
```

Ahora, asumiendo que el servidor brinda una cabecera `Access-Control-Allow-Origin`, todo está bien. Podemos tener el reporte completo del error.

Resumen

Las imágenes ``, estilos externos, scripts y otros recursos proveen los eventos `load` y `error` para rastrear sus cargas:

- `load` se ejecuta cuando la carga ha sido exitosa,
- `error` se ejecuta cuando una carga ha fallado.

La única excepción es el `<iframe>`: por razones históricas siempre dispara el evento `load`, incluso si no encontró la página.

El evento `readystatechange` también funciona para recursos, pero es muy poco usado debido a que los eventos `load/error` son mas simples.

✓ Tareas

Cargando imágenes con una un función de retorno (callback)

importancia: 4

Normalmente, las imágenes son cargadas cuando son creadas. Entonces, cuando nosotros agregamos `` a la página el usuario no ve la imagen inmediatamente. El navegador necesita cargarlo primero.

Para mostrar una imagen inmediatamente, podemos crearlo “en avance”, como esto:

```
let img = document.createElement('img');
img.src = 'my.jpg';
```

El navegador comienza a cargar la imagen y lo guarda en el cache. Después cuando la misma imagen aparece en el documento (no importa cómo) la muestra inmediatamente.

Crear una función `preloadImages(sources, callback)` que cargue todas las imágenes desde una lista de fuentes (`sources`) y, cuando estén listas, ejecutar la función de retorno (`callback`).

Por ejemplo: esto puede mostrar una alerta (`alert`) después de que la imagen sea cargada:

```
function loaded() {
  alert("Imágenes cargadas")
}

preloadImages(["1.jpg", "2.jpg", "3.jpg"], loaded);
```

En caso de un error, la función debería seguir asumiendo que la imagen ha sido “cargada”.

En otras palabras, la función de retorno (`callback`) es ejecutada cuando todas las imágenes han sido cargadas o no.

La función es útil, por ejemplo, cuando planeamos mostrar una galería con muchas imágenes desplazables y estar seguros de que todas las imágenes están cargadas.

En el documento fuente puedes encontrar enlaces para probar imágenes y también el código para verificar si han sido cargadas o no. Debería devolver `300`.

[Abrir un entorno controlado para la tarea.](#) ↗

[A solución](#)

Temas diversos

Mutation observer

`MutationObserver` es un objeto incorporado que observa un elemento DOM y dispara un callback cuando hay cambios en él.

Primero veremos su sintaxis, luego exploraremos un caso de la vida real para ver dónde puede ser útil.

Sintaxis

`MutationObserver` es fácil de usar.

Primero creamos un observador con una función callback:

```
let observer = new MutationObserver(callback);
```

Y luego lo vinculamos a un nodo DOM:

```
observer.observe(node, config);
```

`config` es un objeto con opciones booleanas “a qué clase de cambios reaccionar”:

- `childList` – cambios en los hijos directos de `node`,
- `subtree` – en todos los descendientes de `node`,
- `attributes` – atributos de `node`,
- `attributeFilter` – un array de nombres de atributos, para observar solamente a los seleccionados,
- `characterData` – establece si debe observar cambios de texto en `node.data` o no,

Algunas otras opciones:

- `attributeOldValue` – si es `true`, tanto el valor viejo como el nuevo del atributo son pasados al callback (ver abajo), de otro modo pasa solamente el nuevo (necesita la opción `attributes`),
- `characterDataOldValue` – si es `true`, tanto el valor viejo como el nuevo de `node.data` son pasados al callback (ver abajo), de otro modo pasa solamente el nuevo (necesita la opción `characterData`).

Entonces, después de cualquier cambio, el `callback` es ejecutado: los cambios son pasados en el primer argumento como una lista objetos [MutationRecord](#) ↗, y el observador en sí mismo como segundo argumento.

Los objetos [MutationRecord](#) ↗ tienen como propiedades:

- `type` – tipo de mutación, uno de:
 - `"attributes"`: atributo modificado,
 - `"characterData"`: dato modificado, usado para nodos de texto,
 - `"childList"`: elementos hijos agregados o quitados,
- `target` – dónde ocurrió el cambio: un elemento para `"attributes"`, o un nodo de texto para `"characterData"`, o un elemento para una mutación de `"childList"`,
- `addedNodes/removedNodes` – nodos que fueron agregados o quitados,
- `previousSibling/nextSibling` – los nodos “hermanos”, previos y siguientes a los nodos agregados y quitados,
- `attributeName/attributeNamespace` – el nombre o namespace (para XML) del atributo cambiado,
- `oldValue` – el valor previo, solamente cambios de atributo o cambios de texto si se establece la opción correspondiente `attributeOldValue/characterDataOldValue`.

Por ejemplo, aquí hay un `<div>` con un atributo `contentEditable`. Ese atributo nos permite poner el foco en él y editarlo.

```
<div contentEditable id="elem">Click and <b>edit</b>, please</div>

<script>
let observer = new MutationObserver(mutationRecords => {
  console.log(mutationRecords); // console.log(los cambios)
});
```

```
});

// observa todo exceptuando atributos
observer.observe(elem, {
  childList: true, // observa hijos directos
  subtree: true, // y descendientes inferiores también
  characterDataOldValue: true // pasa el dato viejo al callback
});
</script>
```

Si ejecutamos este código en el navegador, el foco en el `<div>` dado y el cambio en texto dentro de `edit`, `console.log` mostrará una mutación:

```
mutationRecords = [{
  type: "characterData",
  oldValue: "edit",
  target: <text node>,
  // otras propiedades vacías
}];
```

Si hacemos operaciones de edición más complejas, como eliminar el `edit`, el evento de mutación puede contener múltiples registros de mutación:

```
mutationRecords = [{
  type: "childList",
  target: <div#elem>,
  removedNodes: [<b>],
  nextSibling: <text node>,
  previousSibling: <text node>
  // otras propiedades vacías
}, {
  type: "characterData"
  target: <text node>
  // ...detalles de mutación dependen de cómo el navegador maneja tal eliminación
  // puede unir dos nodos de texto adyacentes "edit " y ", please" en un nodo
  // o puede dejarlos como nodos de texto separados
}];
```

Así, `MutationObserver` permite reaccionar a cualquier cambio dentro del subárbol DOM.

Uso para integración

¿Cuándo puede ser práctico esto?

Imagina la situación cuando necesitas añadir un script de terceros que contiene funcionalidad útil pero que también hace algo no deseado, por ejemplo añadir publicidad `<div class="ads">Unwanted ads</div>`.

Naturalmente el script de terceras partes no proporciona mecanismos para removerlo.

Usando `MutationObserver` podemos detectar cuándo aparece el elemento no deseado en nuestro DOM y removerlo.

Hay otras situaciones, como cuando un script de terceras partes agrega algo en nuestro documento y quisiéramos detectarlo para adaptar nuestra página y cambiar el tamaño de algo dinámicamente, etc.

`MutationObserver` permite implementarlo.

Uso para arquitectura

Hay también situaciones donde `MutationObserver` es bueno desde el punto de vista de la arquitectura.

Digamos que estamos haciendo un sitio web acerca de programación. Naturalmente, los artículos y otros materiales pueden contener fragmentos de código.

Tal fragmento en un markup HTML se ve como esto:


```
...
<pre class="language-javascript"><code>
  // aquí el código
  let hello = "world";
</code></pre>
...
```

Para mejorar la legibilidad y al mismo tiempo embellecerlo, usaremos una librería JavaScript de “highlighting” para resaltar elementos de nuestro sitio, por ejemplo [Prism.js](#) [↗](#). Para obtener sintaxis resaltada para el fragmento de arriba en Prism, llamamos a `Prism.highlightElem(pre)`, que examina el contenido de tales elementos y les agrega tags y styles especiales para obtener sintaxis resaltada con color, similares a los que ves en esta página.

¿Exactamente cuándo ejecutar tal método de highlighting? Bien, podemos hacerlo en el evento `DOMContentLoaded`, o poner el script al final de la página. En el momento en que tenemos nuestro DOM listo buscamos los elementos `pre[class*="language"]` y llamamos `Prism.highlightElem` en ellos:

```
// resaltar todos los fragmentos de código en la página
document.querySelectorAll('pre[class*="language"]').forEach(Prism.highlightElem);
```

Todo es simple hasta ahora, ¿verdad? Buscamos fragmentos de código en HTML y los resaltamos.

Continuemos. Digamos que vamos a buscar dinámicamente material desde un servidor. Estudiaremos métodos para ello [más adelante](#) en el tutorial. Por ahora solamente importa que buscamos un artículo HTML desde un servidor web y lo mostramos bajo demanda:

```
let article = /* busca contenido nuevo desde un servidor */
articleElem.innerHTML = article;
```

El nuevo elemento HTML `article` puede contener fragmentos de código. Necesitamos llamar `Prism.highlightElem` en ellos, de otro modo no se resaltarían.

¿Dónde y cuándo llamar `Prism.highlightElem` en un artículo cargado dinámicamente?

Podríamos agregar el llamado al código que carga un “article”, como esto:

```
let article = /* busca contenido nuevo desde un servidor */
articleElem.innerHTML = article;

let snippets = articleElem.querySelectorAll('pre[class*="language-"]');
snippets.forEach(Prism.highlightElem);
```

...Pero imagina que tenemos muchos lugares en el código donde cargamos contenido: artículos, cuestionarios, entradas de foros. ¿Necesitamos poner el llamado al “highlighting” en todos lugares? Eso no es muy conveniente.

¿Y si el contenido es cargado por un módulo de terceras partes? Por ejemplo tenemos un foro, escrito por algún otro, que carga contenido dinámicamente y quisiéramos añadirle sintaxis resaltada. A nadie le gusta emparchar scripts de terceras partes.

Afortunadamente hay otra opción.

Podemos usar `MutationObserver` para detectar automáticamente cuándo los fragmentos de código son insertados en la página y resaltarlos.

Entonces manejaremos la funcionalidad de “highlighting” en un único lugar, liberándonos de la necesidad de integrarlo.

Demo de highlight dinámico

Aquí el ejemplo funcionando.

Si ejecutas el código, este comienza a observar el elemento debajo y resalta cualquier fragmento de código que aparezca allí:

```
let observer = new MutationObserver(mutations => {
```

```

for(let mutation of mutations) {
  // examine nodos nuevos, ¿hay algo para resaltar?

  for(let node of mutation.addedNodes) {
    // seguimos elementos solamente, saltamos los otros nodos (es decir nodos de texto)
    if (!(node instanceof HTMLElement)) continue;

    // verificamos que el elemento insertado sea un fragmento de código
    if (node.matches('pre[class*="language-"]')) {
      Prism.highlightElement(node);
    }

    // ¿o tal vez haya un fragmento de código en su sub-árbol?
    for(let elem of node.querySelectorAll('pre[class*="language-"]')) {
      Prism.highlightElement(elem);
    }
  }
}

});

let demoElem = document.getElementById('highlight-demo');

observer.observe(demoElem, {childList: true, subtree: true});

```

Aquí, abajo, hay un elemento HTML y JavaScript que lo llena dinámicamente usando `innerHTML`.

Por favor ejecuta el código anterior (arriba, que observa aquel elemento) y luego el código de abajo. Verás cómo `MutationObserver` detecta y resalta el fragmento.

A demo-element with `id="highlight-demo"`, run the code above to observe it.

El siguiente código llena su `innerHTML`, lo que causa que `MutationObserver` reaccione y resalte su contenido:

```

let demoElem = document.getElementById('highlight-demo');

// inserta contenido con fragmentos de código
demoElem.innerHTML = `A code snippet is below:


```
<code> let hello = "world!"; </code></pre>
<div>Another one:</div>
<div>


```
<code>.class { margin: 5px; } </code></pre>
</div>
`;

```


```


```

Ahora tenemos un `MutationObserver` que puede rastrear todo el “highlighting” en los elementos observados del `document` entero. Podemos agregar o quitar fragmentos de código en el HTML sin siquiera pensar en ello.

Métodos adicionales

Hay un método para detener la observación del nodo:

- `observer.disconnect()` – detiene la observación.

Cuando detenemos la observación, algunos cambios todavía podrían quedar sin ser procesados por el observador. En tales casos usamos

- `observer.takeRecords()` – obtiene una lista de registros de mutaciones sin procesar, aquellos que ocurrieron pero el callback no manejó.

Estos métodos pueden ser usados juntos, como esto:

```

// obtener una lista de mutaciones sin procesar
// debe ser llamada antes de la desconexión,
// si te interesa las posibles mutaciones recientes sin manejar
let mutationRecords = observer.takeRecords();

```

```
// detener el rastreo de cambios
observer.disconnect();
...
```

i Lo registros devueltos por `observer.takeRecords()` son quitados de la cola de procesamiento

El callback no será llamado en registros devueltos por `observer.takeRecords()`.

i Interacción con la recolección de basura

Los observadores usan internamente referencias débiles a nodos. Esto es: si un nodo es quitado del DOM y se hace inalcanzable, se vuelve basura para ser recolectada.

El mero hecho de que un nodo DOM sea observado no evita la recolección de basura.

Resumen

`MutationObserver` puede reaccionar a cambios en el DOM: atributos, contenido de texto y añadir o quitar elementos.

Podemos usarlo para rastrear cambios introducidos por otras partes de nuestro código o bien para integrarlo con scripts de terceras partes.

`MutationObserver` puede rastrear cualquier cambio. Las opciones de `config` permiten establecer qué se va a observar, se usa para optimización y no desperdiciar recursos en llamados al callback innecesarios.

Selection y Range

En este capítulo cubriremos la selección en el documento, así como la selección en campos de formulario, como `<input>`.

JavaScript puede obtener la selección existente, seleccionar/deseleccionar tanto en su totalidad como parcialmente, eliminar la parte seleccionada del documento, envolverla en una etiqueta, etc.

Tu podrías usar ahora mismo las recetas que hay al final, En la sección "Resumen". Pero será mucho más beneficioso para ti si lees todo el capítulo. Los objetos subyacentes `Range` y `Selection` son fáciles de captar y, por lo tanto, no necesitará recetas para que hagan lo que desees.

Range

El concepto básico de selección [Range](#), es básicamente un par de "puntos límite": inicio y fin del rango.

Cada punto es representado como un nodo DOM principal con el desplazamiento relativo desde su inicio. Si el nodo principal es un nodo de elemento, entonces el desplazamiento es un número secundario, para un nodo de texto es la posición en el texto. Ejemplos a continuación.

Seleccionemos algo.

Primero, podemos crear un rango (el constructor no tiene parámetros):

```
let range = new Range();
```

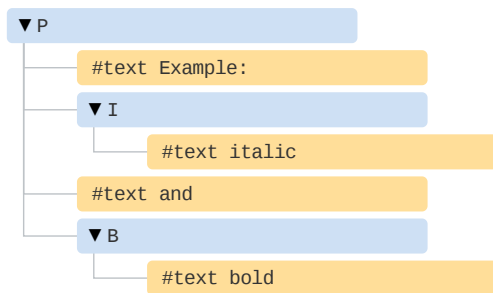
Entonces podemos establecer los límites de selección usando `range.setStart(node, offset)` y `range.setEnd(node, offset)`.

Por ejemplo, considere este fragmento de HTML:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
```

Example: *italic* and **bold**

Aquí está su estructura DOM, tenga en cuenta que aquí los nodos de texto son importantes para nosotros:



Seleccionamos "Example: *italic*". Son los dos primeros hijos de `<p>` (contando nodos de texto):



- The starting point has `<p>` as the parent node, and `0` as the offset.
So we can set it as `range.setStart(p, 0)`.
- The ending point also has `<p>` as the parent node, but `2` as the offset (it specifies the range up to, but not including offset).
So we can set it as `range.setEnd(p, 2)`.

Here's the demo. If you run it, you can see that the text gets selected:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

<script>
  let range = new Range();

  range.setStart(p, 0);
  range.setEnd(p, 2);

  // toString de un rango devuelve su contenido como texto (sin etiquetas)
  alert(range); // Ejemplo: italic

  // aplicar este rango para la selección de documentos (explicado más adelante)
  document.getSelection().addRange(range);
</script>
```

- `range.setStart(p, 0)` – establece el comienzo en el hijo 0 de `<p>` (ese es el nodo de texto "Example:").
- `range.setEnd(p, 2)` – abarca el rango hasta (pero sin incluir) el segundo hijo de `<p>` (ese es el nodo de texto " and ", pero como el final no está incluido, el último nodo seleccionado es `<i>`).

Aquí hay un banco de pruebas más flexible en el que probar más variantes:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

From <input id="start" type="number" value=1> - To <input id="end" type="number" value=4>
<button id="button">Click to select</button>
<script>
  button.onclick = () => {
```

```
let range = new Range();

range.setStart(p, start.value);
range.setEnd(p, end.value);

// aplicar la selección, explicado más adelante
document.getSelection().removeAllRanges();
document.getSelection().addRange(range);
};
</script>
```

Example: *italic* and **bold**

From – To

Ej. seleccionando de 1 a 4 da como rango `<i>italic</i>` and `bold`.

`<p>Example: <i>italic</i> and bold</p>`

No tenemos que usar el mismo nodo en `setStart` y `setEnd`. Un rango puede abarcar muchos nodos no relacionados. Solo es importante que el final sea posterior al comienzo.

Seleccionar partes de nodos de texto

Seleccionemos el texto parcialmente, así:

`<p>Example: <i>italic</i> and bold</p>`

Eso también es posible, solo necesitamos establecer el inicio y el final como un desplazamiento relativo en los nodos de texto.

Necesitamos crear un rango, que:

- comienza desde la posición 2 en `<p>` primer hijo (tomando todas menos dos primeras letras de “**Example:**”)
- termina en la posición 3 de `` primer hijo (tomando las primeras tres letras de “**bold**”, pero no más):

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

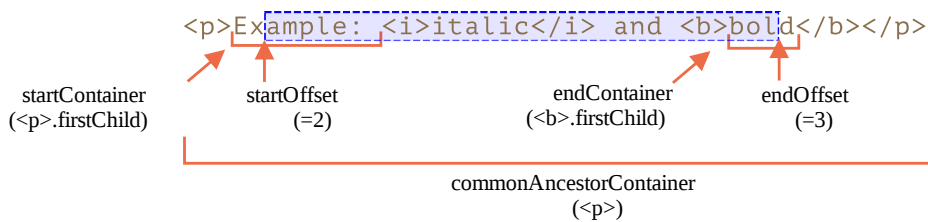
<script>
  let range = new Range();

  range.setStart(p.firstChild, 2);
  range.setEnd(p.querySelector('b').firstChild, 3);

  alert(range); // amplio: italic and bol

  // use este rango para la selección (explicado más adelante)
  window.getSelection().addRange(range);
</script>
```

El objeto de rango tiene las siguientes propiedades:



- `startContainer`, `startOffset` – nodo y desplazamiento del inicio,
 - en el ejemplo anterior: primer nodo de texto dentro de `<p>` y 2.
- `endContainer`, `endOffset` – nodo y desplazamiento del final,
 - en el ejemplo anterior: primer nodo de texto dentro de `` y 3.
- `collapsed` – booleano, `true` si el rango comienza y termina en el mismo punto (por lo que no hay contenido dentro del rango),
 - en el ejemplo anterior: `false`
- `commonAncestorContainer` – el ancestro común más cercano de todos los nodos dentro del rango,
 - en el ejemplo anterior: `<p>`

Métodos de Range

Hay muchos métodos convenientes para manipular rangos.

Establecer inicio de rango:

- `setStart(node, offset)` establecer inicio en: posición `offset` en `node`
- `setStartBefore(node)` establecer inicio en: justo antes `node`
- `setStartAfter(node)` establecer inicio en: justo después `node`

Establecer fin de rango (métodos similares):

- `setEnd(node, offset)` establecer final en: posición `offset` en `node`
- `setEndBefore(node)` establecer final en: justo antes `node`
- `setEndAfter(node)` establecer final en: justo después `node`

Como quedó demostrado, `node` puede ser un nodo de texto o de elemento: para nodos de texto `offset` omite esa cantidad de caracteres, mientras que para los nodos de elementos esa cantidad de nodos secundarios.

Otros:

- `selectNode(node)` establecer rango para seleccionar el `node`
- `selectNodeContents(node)` establecer rango para seleccionar todo el contenido de `node`
- `collapse(toStart)` si `toStart=true` establece final=comienzo, de otra manera comienzo=final, colapsando así el rango
- `cloneRange()` crea un nuevo rango con el mismo inicio/final

Para manipular el contenido dentro del rango:

- `deleteContents()` – eliminar el contenido de rango del documento
- `extractContents()` – eliminar el contenido de rango del documento y lo retorna como `DocumentFragment`
- `cloneContents()` – clonar el contenido del rango y lo retorna como `DocumentFragment`
- `insertNode(node)` – inserta `node` en el documento al comienzo del rango
- `surroundContents(node)` --envuelve `node` alrededor del contenido del rango. Para que esto funcione, el rango debe contener etiquetas de apertura y cierre para todos los elementos dentro de él: no hay rangos parciales como `<i>abc`.

Con estos métodos podemos hacer básicamente cualquier cosa con los nodos seleccionados.

Aquí está el banco de pruebas para verlos en acción:

Haga clic en los botones para ejecutar métodos en la selección, "resetExample" para restablecerla.

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>

<p id="result"></p>
<script>
  let range = new Range();

  // Cada método demostrado se representa aquí:
  let methods = {
    deleteContents() {
      range.deleteContents()
    },
    extractContents() {
      let content = range.extractContents();
      result.innerHTML = "";
      result.append("extracted: ", content);
    },
    cloneContents() {
      let content = range.cloneContents();
      result.innerHTML = "";
      result.append("cloned: ", content);
    },
    insertNode() {
      let newNode = document.createElement('u');
      newNode.innerHTML = "NEW NODE";
      range.insertNode(newNode);
    },
    surroundContents() {
      let newNode = document.createElement('u');
      try {
        range.surroundContents(newNode);
      } catch(e) { console.log(e) }
    },
    resetExample() {
      p.innerHTML = `Example: <i>italic</i> and <b>bold</b>`;
      result.innerHTML = "";

      range.setStart(p.firstChild, 2);
      range.setEnd(p.querySelector('b').firstChild, 3);

      window.getSelection().removeAllRanges();
      window.getSelection().addRange(range);
    }
  };

  for(let method in methods) {
    document.write(`<div><button onclick="methods.${method}()">${method}</button></div>`);
  }

  methods.resetExample();
</script>
```

Haga clic en los botones para ejecutar métodos en la selección, "resetExample" para restablecerla.

Example: *italic* and **bold**

deleteContents

extractContents

cloneContents

insertNode

surroundContents

resetExample

También existen métodos para comparar rangos, pero rara vez se utilizan. Cuando los necesite, consulte el [spec](#) o [MDN manual](#).

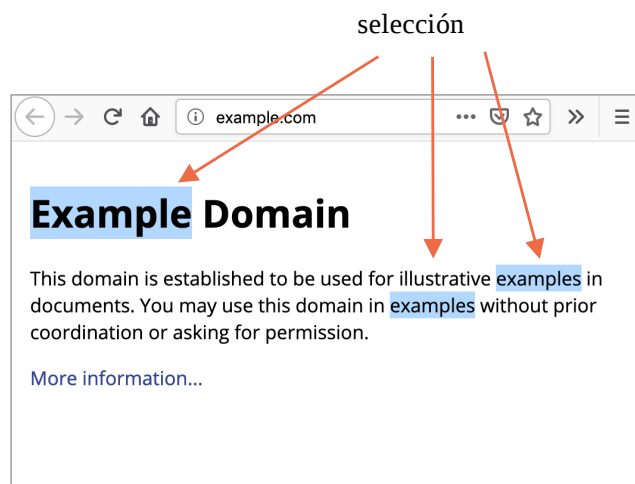
Selection

`Range` es un objeto genérico para gestionar rangos de selección. Podemos crear tales objetos, pasarlos alrededor – no seleccionan visualmente nada por sí mismos.

La selección de documentos está representada por el objeto `Selection`, que se puede obtener como `window.getSelection()` o `document.getSelection()`.

Una selección puede incluir cero o más rangos. Al menos, el [Selection API specification](#) lo dice. Sin embargo, en la práctica, solo Firefox permite seleccionar múltiples rangos en el documento usando `Ctrl+click` (`Cmd+click` para Mac).

Aquí hay una captura de pantalla de una selección con 3 rangos, realizada en Firefox:



Otros navegadores admiten un rango máximo de 1. Como veremos, algunos de los métodos de `Selection` implica que puede haber muchos rangos, pero nuevamente, en todos los navegadores excepto Firefox, hay un máximo de 1.

Propiedades de Selection

Similar a `Range`, una selección tiene un inicio, llamado “ancla(anchor)”, y un final, llamado “foco(focus)”.

Las principales propiedades de `selection` son:

- `anchorNode` – el nodo donde comienza la selección,
- `anchorOffset` – el desplazamiento en `anchorNode` donde comienza la selección,
- `focusNode` – el nodo donde termina la selección,
- `focusOffset` – el desplazamiento en `focusNode` donde termina la selección,
- `isCollapsed` – `true` si la selección no selecciona nada (rango vacío), o no existe.
- `rangeCount` – recuento de rangos en la selección, máximo “1” en todos los navegadores excepto Firefox.

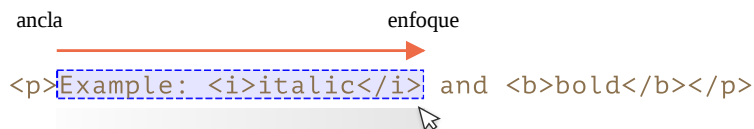
i El final de la selección puede estar en el documento antes del inicio

Hay muchas formas de seleccionar el contenido, dependiendo del agente de usuario: mouse, teclas de acceso rápido, toques en un móvil, etc.

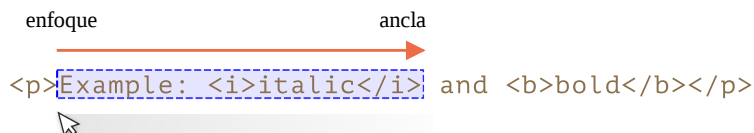
Algunos de ellos, como un mouse, permiten que se pueda crear la misma selección en dos direcciones: “de izquierda a derecha” y “de derecha a izquierda”.

Si el inicio (ancla) de la selección va en el documento antes del final (foco), se dice que esta selección tiene una dirección “hacia adelante”.

Ej. si el usuario comienza a seleccionar con el mouse y pasa de “Example” a “italic”:



De lo contrario, si van desde el final de “italic” to “Example”, la selección se dirige “hacia atrás”, su foco estará antes del ancla:



Eso es diferente de los objetos `Range` que siempre se dirigen hacia adelante: el inicio del rango no puede ser posterior a su final.

Eventos Selection

Hay eventos para realizar un seguimiento de la selección:

- `elem.onselectstart` – cuando una selección comienza en `elem`, ej. el usuario comienza a mover el mouse con el botón presionado.
 - Evitar la acción predeterminada hace que la selección no se inicie.
- `document.onselectionchange` – siempre que cambie una selección.
 - Tenga en cuenta: este controlador solo se puede configurar en `document`.

Demostración de seguimiento de selección

Aquí hay una pequeña demostración que muestra los límites de selección de forma dinámica a medida que cambia:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

From <input id="from" disabled> – To <input id="to" disabled>
<script>
  document.onselectionchange = function() {
    let selection = document.getSelection();

    let {anchorNode, anchorOffset, focusNode, focusOffset} = selection;

    // anchorNode and focusNode are text nodes usually
    from.value = `${anchorNode?.data}, offset ${anchorOffset}`;
    to.value = `${focusNode?.data}, offset ${focusOffset}`;
  };
</script>
```

Demostración de obtención de selección

Para obtener toda la selección:

- Como texto: solo llama `document.getSelection().toString()`.

- Como nodos DOM: obtenga los rangos subyacentes y llame a su método `cloneContents()` (solo el primer rango si no admitimos la selección múltiple de Firefox).

Y aquí está la demostración de cómo obtener la selección como texto y como nodos DOM:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

Cloned: <span id="cloned"></span>
<br>
As text: <span id="astext"></span>

<script>
document.onselectionchange = function() {
    let selection = document.getSelection();

    cloned.innerHTML = astext.innerHTML = "";

    // Clonar nodos DOM de rangos (admitimos selección múltiple aquí)
    for (let i = 0; i < selection.rangeCount; i++) {
        cloned.append(selection.getRangeAt(i).cloneContents());
    }

    // Obtener como texto
    astext.innerHTML += selection;
};
</script>
```

Métodos Selection

Métodos de selección para agregar/eliminar rangos:

- `getRangeAt(i)` – obtener el rango i-ésimo, comenzando desde “0”. En todos los navegadores, excepto Firefox, solo se utiliza 0.
- `addRange(rango)` – agrega un `rango` a la selección. Todos los navegadores excepto Firefox ignoran la llamada, si la selección ya tiene un rango asociado.
- `removeRange(rango)` --elimina `rango` de la selección.
- `removeAllRanges()` --elimina todos los rangos.
- `empty()` – alias para `removeAllRanges`.

Además, existen métodos convenientes para manipular el rango de selección directamente, sin `Range`:

- `collapse(node, offset)` – Reemplazar el rango seleccionado con uno nuevo que comienza y termina en el `node` dado, en posición `offset`.
- `setPosition(node, offset)` – alias para `collapse`.
- `collapseToStart()` – colapsar (reemplazar con un rango vacío) al inicio de la selección,
- `collapseToEnd()` – colapso hasta el final de la selección,
- `extend(node, offset)` – mover el foco de la selección al `node` dado, posición `offset`,
- `setBaseAndExtent(anchorNode, anchorOffset, focusNode, focusOffset)` – reemplazar el rango de selección con el inicio dado `anchorNode/anchorOffset` y final `focusNode/focusOffset`. Se selecciona todo el contenido entre ellos.
- `selectAllChildren(node)` – seleccionar todos los hijos del `node`.
- `deleteFromDocument()` – eliminar el contenido seleccionado del documento.
- `containsNode(node, allowPartialContainment = false)` – comprueba si la selección contiene `node` (parcialmente si el segundo argumento es `true`)

Entonces, para muchas tareas podemos llamar a los métodos de `Selection`, y no es necesario acceder al objeto `Range` subyacente.

Por ejemplo, seleccionando todo el contenido del párrafo `<p>`:

```

<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

<script>
  // seleccione desde el 0 hijo de <p> hasta el último hijo
  document.getSelection().setBaseAndExtent(p, 0, p, p.childNodes.length);
</script>

```

Lo mismo usando rangos:

```

<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

<script>
  let range = new Range();
  range.selectNodeContents(p); // o selectNode(p) para seleccionar el tag <p> también

  document.getSelection().removeAllRanges(); // borrar la selección existente si la hubiera
  document.getSelection().addRange(range);
</script>

```

i Para seleccionar, primero elimine la selección existente

Si la selección ya existe, vacíelo primero con `removeAllRanges()`. Y luego agregue rangos. De lo contrario, todos los navegadores excepto Firefox ignoran los nuevos rangos.

La excepción son algunos métodos de selección, que reemplazan la selección existente, como `setBaseAndExtent`.

Selección en controles de formulario

Elementos de formulario, como `input` y `textarea` proporciona [API especial para la selección](#), sin objetos `Selection` o `Range`. Como un valor de entrada es un texto puro, no HTML, no hay necesidad de tales objetos, todo es mucho más simple.

Propiedades:

- `input.selectionStart` – posición de inicio de selección (escribible),
- `input.selectionEnd` – posición del final de la selección (escribible),
- `input.selectionDirection` – dirección de selección, una de: “adelante” “hacia atrás” o “ninguno” (si, por ejemplo, se selecciona con un doble clic del mouse),

Eventos:

- `input.onselect` – se activa cuando se selecciona algo.

Métodos:

- `input.select()` – selecciona todo en el control de texto (puede ser `textarea` en vez de `input`),
- `input.setSelectionRange(start, end, [direction])` – cambiar la selección para abarcar desde la posición `start` hasta `end`, en la dirección indicada (opcional).
- `input.setRangeText(replacement, [start], [end], [selectionMode])` – reemplace un rango de texto con el nuevo texto.

Los argumentos opcionales `start` y `end`, si se proporcionan, establecen el inicio y el final del rango; de lo contrario, se utiliza la selección del usuario.

El último argumento, `selectionMode`, determina cómo se establecerá la selección después de que se haya reemplazado el texto. Los posibles valores son:

- `"select"` – se seleccionará el texto recién insertado.
- `"start"` – el rango de selección se colapsa justo antes del texto insertado (el cursor estará inmediatamente antes).
- `"end"` – el rango de selección se colapsa justo después del texto insertado (el cursor estará justo después).

- `"preserve"` – intenta preservar la selección. Este es el predeterminado.

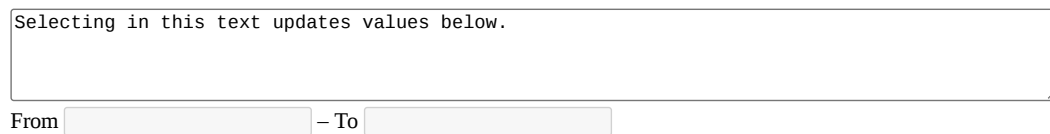
Ahora veamos estos métodos en acción.

Ejemplo: Seguimiento de selección

Por ejemplo, este código usa el evento `onselect` para rastrear la selección:

```
<textarea id="area" style="width:80%;height:60px">
Selecting in this text updates values below.
</textarea>
<br>
From <input id="from" disabled> - To <input id="to" disabled>

<script>
  area.onselect = function() {
    from.value = area.selectionStart;
    to.value = area.selectionEnd;
  };
</script>
```



Tenga en cuenta:

- `onselect` se activa cuando se selecciona algo, pero no cuando se elimina la selección.
- El evento `document.onselectionchange` no debería activarse para las selecciones dentro de un control de formulario, según el [spec](#), ya que no está relacionado con la selección y los rangos del “documento”. Algunos navegadores lo generan, pero no debemos confiar en él.

Ejemplo: cursor en movimiento

Podemos cambiar `selectionStart` y `selectionEnd`, que establece la selección.

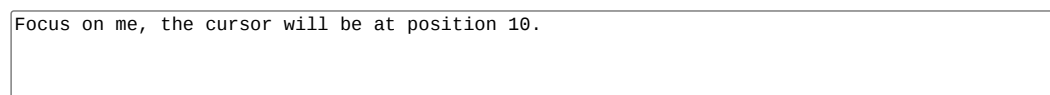
Un caso límite importante es cuando `selectionStart` y `selectionEnd` son iguales entre sí. Entonces es exactamente la posición del cursor. O, para reformular, cuando no se selecciona nada, la selección se contrae en la posición del cursor.

Entonces, al establecer `selectionStart` y `selectionEnd` en el mismo valor, movemos el cursor.

Por ejemplo:

```
<textarea id="area" style="width:80%;height:60px">
Focus on me, the cursor will be at position 10.
</textarea>

<script>
  area.onfocus = () => {
    // zero delay setTimeout to run after browser "focus" action finishes
    setTimeout(() => {
      // we can set any selection
      // if start=end, the cursor is exactly at that place
      area.selectionStart = area.selectionEnd = 10;
    });
  };
</script>
```



Ejemplo: modificar la selección

Para modificar el contenido de la selección, podemos utilizar el método `input.setRangeText()`. Por supuesto, podemos leer `selectionStart/End` y, con el conocimiento de la selección, cambiar la subcadena correspondiente

de `value` , pero `setRangeText` es más poderoso y a menudo más conveniente.

Ese es un método algo complejo. En su forma más simple de un argumento, reemplaza el rango seleccionado por el usuario y elimina la selección.

Por ejemplo, aquí la selección de usuario estará envuelta por `*...*` :

```
<input id="input" style="width:200px" value="Select here and click the button">
<button id="button">Wrap selection in stars *...*</button>

<script>
button.onclick = () => {
  if (input.selectionStart == input.selectionEnd) {
    return; // nada fue seleccionado
  }

  let selected = input.value.slice(input.selectionStart, input.selectionEnd);
  input.setRangeText(`*${selected}*`);
};
</script>
```

Select here and click the button Wrap selection in stars *...*

Con más argumentos, podemos establecer un rango `start` y `end` .

En este ejemplo, encontramos `THIS` en el texto de entrada, lo reemplazamos y mantenemos el reemplazo seleccionado:

```
<input id="input" style="width:200px" value="Replace THIS in text">
<button id="button">Replace THIS</button>

<script>
button.onclick = () => {
  let pos = input.value.indexOf("THIS");
  if (pos >= 0) {
    input.setRangeText("**THIS*", pos, pos + 4, "select");
    input.focus(); // focus to make selection visible
  }
};
</script>
```

Replace THIS in text Replace THIS

Ejemplo: insertar en el cursor

Si no se selecciona nada, o usamos el mismo `comienzo` y `final` en `setRangeText` , entonces el nuevo texto se acaba de insertar, no se elimina nada.

También podemos insertar algo “en el cursor” usando `setRangeText` .

Aquí hay un botón que se inserta `"HELLO"` en la posición del cursor y lo coloca inmediatamente después. Si la selección no está vacía, entonces se reemplaza (podemos detectarla comparando `selectionStart!=selectionEnd` y hacer otra cosa en su lugar):

```
<input id="input" style="width:200px" value="Text Text Text Text Text">
<button id="button">Insert "HELLO" at cursor</button>

<script>
button.onclick = () => {
  input.setRangeText("HELLO", input.selectionStart, input.selectionEnd, "end");
  input.focus();
};
</script>
```

Text Text Text Text Text Insert "HELLO" at cursor

Haciendo no seleccionable

Para hacer algo no seleccionable, hay tres formas:

1. Usar propiedad CSS `user-select: none`.

```
<style>
#elem {
  user-select: none;
}
</style>
<div>Selectable <div id="elem">Unselectable</div> Selectable</div>
```

Esto no permite que la selección comience en `elem`. Pero el usuario puede iniciar la selección en otro lugar e incluir `elem` en ella.

Entonces, `elem` se convertirá en parte de `document.getSelection()`, por lo que la selección realmente ocurre, pero su contenido generalmente se ignora al copiar y pegar.

2. Evita la acción predeterminada en los eventos `onselectstart` o `mousedown`.

```
<div>Selectable <div id="elem">Unselectable</div> Selectable</div>

<script>
  elem.onselectstart = () => false;
</script>
```

Esto evita que la selección se inicie en `elem`, pero el visitante puede iniciarla en otro elemento y luego extenderla a `elem`.

Eso es conveniente cuando hay otro controlador de eventos en la misma acción que activa la selección (por ejemplo, `mousedown`). Así que deshabilitamos la selección para evitar conflictos, permitiendo que se copien los contenidos de `elem`.

3. También podemos borrar la selección post-factum después de que suceda con `document.getSelection().Empty()`. Eso se usa con poca frecuencia, ya que provoca un parpadeo no deseado cuando la selección aparece o desaparece.

Referencias

- [DOM spec: Range](#)
- [Selection API](#)
- [HTML spec: APIs for the text control selections](#)

Resumen

Cubrimos dos API diferentes para las selecciones:

1. Para el documento: objetos `Selection` y `Range`.
2. Para `input`, `textarea`: métodos y propiedades adicionales.

La segunda API es muy simple, ya que funciona con texto.

Las recetas más utilizadas probablemente sean:

1. Obteniendo la selección:

```
let selection = document.getSelection();

let cloned = /* elemento para clonar los nodos seleccionados para */;

// luego aplica los métodos Range a selection.getRangeAt (0)
// o, como aquí, a todos los rangos para admitir selección múltiple
for (let i = 0; i < selection.rangeCount; i++) {
```

```
cloned.append(selection.getRangeAt(i).cloneContents());
}
```

2. Configuración de la selección:

```
let selection = document.getSelection();

// directamente:
selection.setBaseAndExtent(...from...to...);

// o podemos crear un rango y:
selection.removeAllRanges();
selection.addRange(range);
```

Y finalmente, sobre el cursor. La posición del cursor en elementos editables, como `<textarea>` está siempre al principio o al final de la selección. Podemos usarlo para obtener la posición del cursor o para mover el cursor configurando `elem.selectionStart` y `elem.selectionEnd`.

Loop de eventos: microtareas y macrotareas

El flujo de ejecución de JavaScript en el navegador, así como en Node.js, está basado en un *event loop* (loop de eventos).

Entender como este loop de eventos funciona es importante para optimizaciones y en algunos casos para utilizar la arquitectura correcta.

En este capítulo primero vamos a ver detalles teóricos acerca de cómo funcionan las cosas y luego veremos aplicaciones prácticas de ese conocimiento.

Loop de eventos

El concepto de *loop de eventos* es muy simple. Existe un ciclo infinito en el que el motor de JavaScript espera por una tarea, luego ejecuta la tarea requerida y finalmente vuelve a dormir esperando por una nueva tarea.

EL algoritmo general del motor:

1. Mientras haya tareas:
 - ejecutarlas comenzando por la más antigua.
2. Dormir hasta que aparezca una tarea, luego volver a 1.

Eso es una formalización de lo que vemos cuando navegamos por una página. El motor JavaScript no hace nada la mayoría del tiempo y solo corre cuando un script/controlador/evento se activa.

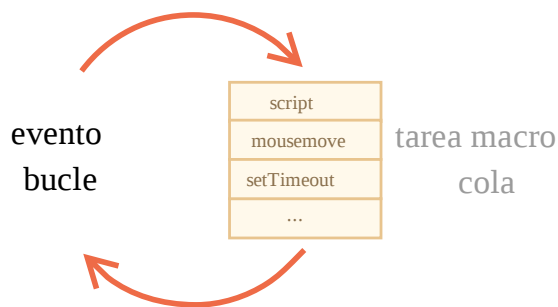
Ejemplos de tareas:

- Cuando un script externo `<script src="...">` se carga, la tarea es ejecutarlo.
- Cuando un usuario mueve el mouse, la tarea es enviar el evento `mousemove` y ejecutar el controlador.
- Cuando llega el momento de un `setTimeout` programado, la tarea es ejecutar su callback.
- ... y así sucesivamente.

Las tareas son programadas --> el motor las ejecuta --> espera por más tareas (mientras duerme y prácticamente no consume CPU).

Puede ocurrir que una tarea llegue mientras el motor está ocupado, entonces es puesta en cola.

Las tareas forman una cola, también llamada “Cola de macrotarea” (Término de v8):



Por ejemplo, mientras el motor está ocupado ejecutando un `script`, un usuario podría mover su mouse causando `mousemove` o también podría ser `setTimeout`, etc. Todas estas tareas forman una cola como se observa en la imagen arriba.

Las tareas de la cola son ejecutadas según la base “El que primero llega primero se atiende”. Cuando el motor del navegador termina con el `script`, se encarga del evento `mousemove`, continúa con `setTimeout`, etc.

Hasta ahora bastante simple, ¿no?

Dos detalles más:

1. El renderizado nunca ocurre mientras el motor ejecuta una tarea. No importa si la tarea ocupa mucho tiempo. Solo se realizan cambios a DOM una vez que la tarea finaliza.
2. Si una tarea consume demasiado tiempo, el navegador no puede hacer otras tareas, procesos o eventos por lo que después de un tiempo muestra una alerta “La página no responde” sugiriendo detener la tarea con la página completa. Esto ocurre cuando hay muchos cálculos complejos o un error en la programación que lleva a un bucle infinito.

Esa fue la teoría. Ahora veamos como podemos aplicar ese conocimiento.

Caso de uso 1: dividiendo tareas que demandan alto consumo de CPU

Digamos que tenemos una tarea con un alto consumo de CPU.

Por ejemplo, el resaltado de sintaxis (usado para colorear ejemplos de código en esta página) demanda un alto consumo de CPU. Para resaltar el código, realiza el análisis, crea muchos elementos coloreados, los agrega al documento... para una gran cantidad de texto esto lleva mucho tiempo.

Mientras el motor está ocupado con el resaltado de sintaxis, no puede hacer otras cosas relacionadas a DOM, procesar eventos de usuario, etc. Podría incluso provocar que el navegador se “congele” por un momento, lo que es inaceptable.

Podemos evitar problemas dividiendo la tarea en piezas más pequeñas. Resaltar primero 100 líneas, después programar `setTimeout` (con cero delay) para las próximas 100 líneas y así sucesivamente.

Para demostrar este enfoque y en pos de una mayor simplicidad, en lugar de resaltado de texto tomemos una función que cuenta desde 1 hasta 1000000000.

Si ejecutas el código siguiente, el navegador se va a “congelar” por un instante. Para JS desde el lado del servidor esto es claramente notable y si lo ejecutas en el navegador intenta hacer click en otros botones de la página. Verás que ningún otro evento es procesado hasta que termine el conteo.

```
let i = 0;

let start = Date.now();

function count() {

  // realiza una tarea pesada
  for (let j = 0; j < 1e9; j++) {
    i++;
  }
}
```



```

    alert("Done in " + (Date.now() - start) + 'ms');
}

count();

```

Puede que incluso se muestre una advertencia: “Un script en esta página está provocando que el navegador se ejecute con lentitud”.

Dividamos la tarea usando llamadas anidadas a `setTimeout` :

```

let i = 0;

let start = Date.now();

function count() {

    // realiza una parte de la tarea pesada(*)
    do {
        i++;
    } while (i % 1e6 !== 0);

    if (i === 1e9) {
        alert("Done in " + (Date.now() - start) + 'ms');
    } else {
        setTimeout(count); // programa una nueva llamada (**)
    }
}

count();

```

Ahora la interfaz del navegador es completamente funcional durante el conteo.

Una sola ejecución de `count` realiza parte del trabajo `(*)` y luego se reprograma a sí misma `(**)` si lo necesita:

1. La primer ejecución cuenta: `i=1...1000000`.
2. La segunda cuenta: `i=1000001...2000000`.
3. ...y así sucesivamente.

Ahora, si una tarea secundaria (por ejemplo el evento `onclick`) aparece mientras el motor está ocupado ejecutando la parte 1, entonces es puesta en lista y ejecutada cuando la parte 1 termina, antes de la siguiente parte. Retornos periódicos al ciclo de eventos entre ejecuciones de `count` brinda suficiente “aire” al motor de JavaScript para hacer algo más, para reaccionar a otras acciones del usuario.

Lo notable es que ambas variantes, con y sin división de la tarea haciendo uso de `setTimeout`, son comparables en velocidad. No hay mucha diferencia en el tiempo de conteo general.

Para acercar aún más los tiempos, hagamos una mejora.

Movamos la programación de `setTimeout` al inicio de `count()` :

```

let i = 0;

let start = Date.now();

function count() {

    // movemos la programación al principio
    if (i < 1e9 - 1e6) {
        setTimeout(count); // programamos la nueva llamada
    }

    do {
        i++;
    } while (i % 1e6 !== 0);

    if (i === 1e9) {

```

```

    alert("Done in " + (Date.now() - start) + 'ms');
  }

}

count();

```

Ahora cuando iniciamos `count()` y vemos que necesitaremos más `count()`, lo programamos inmediatamente, antes de hacer el trabajo.

Si lo ejecutas, es fácil notar que lleva bastante menos tiempo.

¿Por qué pasa esto?

Es simple: como recordarás existe un retraso mínimo en el navegador de 4ms para varias llamadas anidadas a `setTimeout`. Si configuramos `0`, es 4ms (o un poco más). Por lo que mientras antes lo programemos más rápido se ejecutará.

Finalmente hemos dividido una tarea con un alto consumo de CPU en partes y ahora no bloquea la interfaz de usuario. Y el tiempo general de ejecución no es mucho mayor.

Caso de uso 2: indicación de progreso

Otro beneficio de dividir tareas pesadas para scripts de navegadores es que podemos indicar el progreso.

Usualmente el navegador renderiza al terminar la ejecución del código que actualmente se está ejecutando. No importa si la tarea lleva demasiado tiempo.

Por un lado eso es genial porque nuestra función puede crear muchos elementos, agregarlos de a uno al documento y cambiar sus estilos... el visitante no verá ningún estado intermedio, sin finalizar. Lo cuál es importante, ¿no?

Acá hay una demostración, los cambios a `i` no se mostrarán hasta que la función finalice, por lo que veremos solo el último valor:

```

<div id="progress"></div>

<script>

  function count() {
    for (let i = 0; i < 1e6; i++) {
      i++;
      progress.innerHTML = i;
    }
  }

  count();
</script>

```

... Pero puede que queramos mostrar algo durante la tarea, por ejemplo una barra de progreso.

Si dividimos la tarea más pesada en partes más pequeñas usando `setTimeout`, entonces los cambios son aplicados entre ellos.

Esto se ve mejor:

```

<div id="progress"></div>

<script>
  let i = 0;

  function count() {

    // realiza una parte del trabajo pesado (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 != 0);
  }

```

```

    if (i < 1e7) {
      setTimeout(count);
    }

    }

    count();
  </script>

```

Ahora el `<div>` muestra el incremento en el valor `i`, una especie de barra de progreso.

Caso de uso 3: hacer algo después del evento

En un controlador de evento nosotros podemos decidir posponer alguna acción hasta que el evento aparezca y sea controlado en todos los niveles. Podemos hacer esto envolviendo el código en un `setTimeout` con retraso cero.

En el capítulo [Envío de eventos personalizados](#) vimos un ejemplo: el evento personalizado `menu-open` es distribuido en `setTimeout`, de modo que ocurre después de que el evento “click” se maneja por completo.

```

menu.onclick = function() {
  // ...

  // crea un evento personalizado con los datos del elemento de menú en el que se hizo clic
  let customEvent = new CustomEvent("menu-open", {
    bubbles: true
  });

  // envia el evento personalizado de forma asincrónica
  setTimeout(() => menu.dispatchEvent(customEvent));
};

```

Macrotareas y Microtareas

Junto con las *macrotareas* descritas en este capítulo, existen *microtareas* mencionadas en el capítulo [Microtareas \(Microtasks\)](#).

Las microtareas provienen únicamente de nuestro código. Por lo general, se crean mediante promesas: una ejecución del controlador `.then` / `catch` / `finally` se convierte en una microtarea. Las microtareas también se utilizan “bajo la cubierta” de `await`, ya que es otra forma de manejo de promesas.

Existe también una función especial `queueMicrotask(func)` que pone a `func` en la cola de microtareas.

Inmediatamente después de cada *macrotarea*, el motor ejecuta todas las tareas desde la cola de *microtareas*, antes de ejecutar cualquier otra *macrotarea* o renderización o cualquier otra cosa.

Por ejemplo:

```

setTimeout(() => alert("timeout"));

Promise.resolve()
  .then(() => alert("promise"));

alert("code");

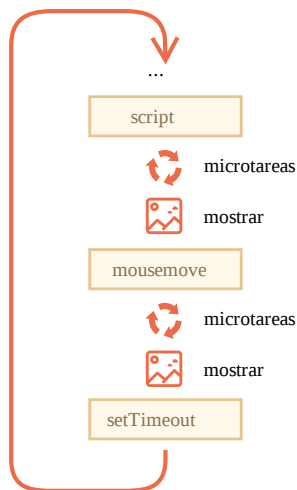
```

¿Cuál será el orden en este caso?

1. `code` se muestra primero porque es una llamada sincrónica regular.
2. `promise` aparece segundo, porque `.then` pasa a través de la cola de microtareas y se ejecuta después del código actual.
3. `timeout` aparece última, porque es una macrotarea.

La imagen del loop de eventos completa se ve así (el orden es de arriba a abajo, es decir: primero el script, luego las microtareas, el renderizado, etc.):

evento
bucle



Todas las microtareas se completan antes de que se lleve a cabo cualquier otro manejo o renderizado o cualquier otra macro tarea.

Eso es importante, ya que garantiza que el entorno de la aplicación es básicamente el mismo (sin cambios de coordenadas del mouse, sin nuevos datos de red, etc.) entre las microtareas.

Si quisiéramos ejecutar una función de forma asíncrona (después del código actual), pero antes de que se procesen los cambios o se manejen nuevos eventos, podemos programarla con `queueMicrotask`.

Aquí hay un ejemplo con la “barra de progreso de conteo”, similar al que se mostró anteriormente, pero se usa `queueMicrotask` en lugar de `setTimeout`. Puedes ver que se renderiza al final. Al igual que el código sincrónico:

```
<div id="progress"></div>

<script>
  let i = 0;

  function count() {

    // realiza una parte del trabajo pesado (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 !== 0);

    if (i < 1e6) {
      queueMicrotask(count);
    }

  }

  count();
</script>
```

Resumen

El algoritmo más detallado del loop de eventos (aunque aún simplificado en comparación con la [especificación](#)):

1. Retirar de la cola y ejecutar la tarea más antigua de la cola *macro tareas* (por ejemplo, “script”).
2. Ejecutar todas las *micro tareas*:
 - Mientras la cola de microtareas no esté vacía:
 - Retirar de la cola y ejecutar la microtarea más antigua.
3. Renderizar los cambios si los hubiera.
4. Si la cola de macro tareas está vacía, esperar hasta que aparezca una macro tarea.
5. Ejecutar el paso 1.

Para programar una nueva *macrotarea*:

- Usar `setTimeout(f)` con un retraso de cero.

Eso puede usarse para dividir una gran tarea de cálculo en partes, para que el navegador pueda reaccionar a los eventos del usuario y mostrar el progreso entre ellos.

Además, se utiliza en los controladores de eventos para programar una acción después de que el evento se haya manejado por completo.

Para programar una nueva *microtarea*

- Usar `queueMicrotask(f)`.
- También se usan promesas para que los controladores pasen por la cola de microtarear.

No hay gestión de eventos de red o de UI entre las microtarear: se ejecutan inmediatamente una tras otra.

Por lo tanto, es posible que desee `queueMicrotask` para ejecutar una función de forma asíncrona, pero dentro del estado del entorno.

Web Workers

Para cálculos largos y pesados que no deberían bloquear el ciclo de eventos, podemos usar [Web Workers](#) .

Esa es una forma de ejecutar código en otro hilo paralelo.

Los Web Workers pueden intercambiar mensajes con el proceso principal, pero tienen sus propias variables y su propio ciclo de eventos.

Los Web Workers no tienen acceso a DOM, por lo que son útiles principalmente para cálculos, para utilizar varios núcleos de CPU simultáneamente.

Soluciones

Recorriendo el DOM

DOM children

Hay muchas maneras, por ejemplo:

El nodo `<div>` del DOM:

```
document.body.firstChild
// o
document.body.children[0]
// o (el primer nodo es un espacio, así que tomamos el segundo)
document.body.childNodes[1]
```

El nodo `` del DOM:

```
document.body.lastElementChild
// o
document.body.children[1]
```

El segundo `` (con Pete):

```
// obtener <ul>, y luego obtener su último elemento hijo
document.body.lastElementChild.lastElementChild
```

[A formulación](#)

La pregunta de los hermanos

1. Sí, verdadero. El elemento `elem.lastChild` siempre es el último, no tiene `nextSibling`.
2. No, falso, porque `elem.children[0]` es el primer hijo *entre elementos* pero pueden existir nodos que no son elementos antes que él. `previousSibling` puede ser un nodo texto.

Ten en cuenta: para ambos casos, si no hay hijos habrá un error.

Si no hay hijos, `elem.lastChild` es `null`, entonces no podemos acceder a `elem.lastChild.nextSibling`. Y la colección `elem.children` es vacía (como un array vacío `[]`).

[A formulación](#)

Seleccionar todas las celdas diagonales

Usaremos las propiedades de las `filas` y las `celdas` para acceder a las celdas de la tabla diagonal

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Buscar: getElement*, querySelector*

Buscar elementos

Hay muchas maneras de resolverlo.

Aquí hay algunas de ellas:

```
// 1. La tabla con `id="age-table"`.
let table = document.getElementById('age-table')

// 2. Todos los elementos `label` dentro de esa tabla
table.getElementsByTagName('label')
// or
document.querySelectorAll('#age-table label')

// 3. El primer `td` en la tabla (con la palabra "Age")
table.rows[0].cells[0]
// or
table.getElementsByTagName('td')[0]
// or
table.querySelector('td')

// 4. El `form` con name="search"
// suponiendo que sólo hay un elemento con name="search" en el documento
let form = document.getElementsByName('search')[0]
//o, utilizando el form específicamente
document.querySelector('form[name="search"]')

// 5. El primer input en el form.
form.getElementsByTagName('input')[0]
// o
form.querySelector('input')

// 6. El último input en el form.
let inputs = form.querySelectorAll('input') // encontrar todos los inputs
inputs[inputs.length-1] // obtener el último
```

[A formulación](#)

Propiedades del nodo: tipo, etiqueta y contenido

Contar los descendientes

Hagamos un ciclo sobre ``:

```
for (let li of document.querySelectorAll('li')) {  
  ...  
}
```

En el ciclo, necesitamos introducir el texto dentro de cada `li`.

Podemos leer el texto del primer nodo hijo de `li`, que es el nodo de texto:

```
for (let li of document.querySelectorAll('li')) {  
  let title = li.firstChild.data;  
  
  // el título es el texto en <li> antes de cualquier otro nodo  
}
```

Entonces podemos obtener el número de descendientes como

`li.getElementsByTagName('li').length`.

[Abrir la solución en un entorno controlado.](#) ↗

A formulación

¿Qué hay en `nodeType`?

Aquí hay una trampa.

En el momento de la ejecución de `<script>`, el último nodo DOM es exactamente `<script>`, porque el navegador aún no procesó el resto de la página.

Entonces el resultado es `1` (nodo de elemento).

```
<html>  
  
<body>  
  <script>  
    alert(document.body.lastChild.nodeType);  
  </script>  
</body>  
  
</html>
```

A formulación

Etiqueta en comentario

La respuesta: **BODY**.

```
<script>  
  let body = document.body;  
  
  body.innerHTML = "<!--" + body.tagName + "-->";  
  
  alert( body.firstChild.data ); // BODY  
</script>
```

¿Qué está pasando paso a paso?

1. El contenido de `<body>` se reemplaza con el comentario. El comentario es `<!--BODY-->`, porque `body.tagName == "BODY"`. Como recordamos, `tagName` siempre está en mayúsculas en HTML.
2. El comentario es ahora el único nodo hijo, así que lo obtenemos en `body.firstChild`.
3. La propiedad `data` del comentario es su contenido (dentro de `<!-- ... -->`): `"BODY"`.

A formulación

¿Dónde está el "document" en la jerarquía?

Podemos ver a qué clase pertenece, imprimiéndola, así:

```
alert(document); // [object HTMLDocument]
```

O:

```
alert(document.constructor.name); // HTMLDocument
```

Entonces, `document` es una instancia de la clase `HTMLDocument`.

¿Cuál es su lugar en la jerarquía?

Sí, podríamos examinar las especificaciones, pero sería más rápido averiguarlo manualmente.

Recorramos la cadena de prototype través de `__proto__`.

Como sabemos, los métodos de una clase están en el `prototype` del constructor. Por ejemplo, `HTMLDocument.prototype` tiene métodos para documentos.

Además, hay una referencia a la función constructor dentro de `prototype`:

```
alert(HTMLDocument.prototype.constructor === HTMLDocument); // true
```

Para obtener un nombre de la clase como string, podemos usar `constructor.name`. Hagámoslo para toda la cadena prototype de `document`, hasta la clase `Node`:

```
alert(HTMLDocument.prototype.constructor.name); // HTMLDocument
alert(HTMLDocument.prototype.__proto__.constructor.name); // Document
alert(HTMLDocument.prototype.__proto__.__proto__.constructor.name); // Node
```

Esa es la jerarquía.

También podríamos examinar el objeto usando `console.dir(document)` y ver estos nombres abriendo `__proto__`. La consola los toma del `constructor` internamente.

A formulación

Atributos y propiedades

Obtén en atributo

```
<!DOCTYPE html>
<html>
<body>

  <div data-widget-name="menu">Elige el género</div>
```



```

<script>
  // obteniéndolo
  let elem = document.querySelector('[data-widget-name]');

  // leyendo el valor
  alert(elem.dataset.widgetName);
  // o
  alert(elem.getAttribute('data-widget-name'));
</script>
</body>
</html>

```

A formulación

Haz los enlaces externos naranjas

Primero, necesitamos encontrar todos los enlaces externos.

Hay dos.

El primero es encontrar todos los enlaces usando `document.querySelectorAll('a')` y luego filtrar lo que necesitamos:

```

let links = document.querySelectorAll('a');

for (let link of links) {
  let href = link.getAttribute('href');
  if (!href) continue; // no atributo

  if (!href.includes('://')) continue; // no protocolo

  if (href.startsWith('http://internal.com')) continue; // interno

  link.style.color = 'orange';
}

```

Tenga en cuenta: nosotros usamos `link.getAttribute('href')`. No `link.href`, porque necesitamos el valor del HTML.

...Otra forma más simple sería agregar las comprobaciones al selector CSS:

```

// busque todos los enlaces que tengan: // en href
//pero href no comienza con http://internal.com
let selector = 'a[href*="://"]:not([href^="http://internal.com"])';
let links = document.querySelectorAll(selector);

links.forEach(link => link.style.color = 'orange');

```

[Abrir la solución en un entorno controlado.](#)

A formulación

Modifying the document

createTextNode vs innerHTML vs textContent

Answer: **1 and 3.**

Both commands result in adding the `text` "as text" into the `elem`.

Here's an example:

```

<div id="elem1"></div>
<div id="elem2"></div>
<div id="elem3"></div>
<script>
  let text = '<b>text</b>';

  elem1.append(document.createTextNode(text));
  elem2.innerHTML = text;
  elem3.textContent = text;
</script>

```

A formulación

Clear the element

First, let's see how *not* to do it:

```

function clear(elem) {
  for (let i=0; i < elem.childNodes.length; i++) {
    elem.childNodes[i].remove();
  }
}

```

That won't work, because the call to `remove()` shifts the collection `elem.childNodes`, so elements start from the index `0` every time. But `i` increases, and some elements will be skipped.

The `for...of` loop also does the same.

The right variant could be:

```

function clear(elem) {
  while (elem.firstChild) {
    elem.firstChild.remove();
  }
}

```

And also there's a simpler way to do the same:

```

function clear(elem) {
  elem.innerHTML = '';
}

```

A formulación

Why does "aaa" remain?

The HTML in the task is incorrect. That's the reason of the odd thing.

The browser has to fix it automatically. But there may be no text inside the `<table>`: according to the spec only table-specific tags are allowed. So the browser shows `"aaa"` before the `<table>`.

Now it's obvious that when we remove the table, it remains.

The question can be easily answered by exploring the DOM using the browser tools. You'll see `"aaa"` before the `<table>`.

The HTML standard specifies in detail how to process bad HTML, and such behavior of the browser is correct.

A formulación

Create a list

Please note the usage of `textContent` to assign the `` content.

[Abrir la solución en un entorno controlado.](#)

A formulación

Create a tree from the object

The easiest way to walk the object is to use recursion.

1. [The solution with innerHTML](#).
2. [The solution with DOM](#).

A formulación

Show descendants in a tree

To append text to each `` we can alter the text node `data`.

[Abrir la solución en un entorno controlado.](#)

A formulación

Create a calendar

We'll create the table as a string: `"<table>...</table>"`, and then assign it to `innerHTML`.

The algorithm:

1. Create the table header with `<th>` and weekday names.
2. Create the date object `d = new Date(year, month-1)`. That's the first day of `month` (taking into account that months in JavaScript start from `0`, not `1`).
3. First few cells till the first day of the month `d.getDay()` may be empty. Let's fill them in with `<td></td>`.
4. Increase the day in `d: d.setDate(d.getDate()+1)`. If `d.getMonth()` is not yet the next month, then add the new cell `<td>` to the calendar. If that's a Sunday, then add a newline `"</tr><tr>"`.
5. If the month has finished, but the table row is not yet full, add empty `<td>` into it, to make it square.

[Abrir la solución en un entorno controlado.](#)

A formulación

Colored clock with setInterval

First, let's make HTML/CSS.

Each component of the time would look great in its own ``:

```
<div id="clock">
  <span class="hour">hh</span>:<span class="min">mm</span>:<span class="sec">ss</span>
</div>
```

Also we'll need CSS to color them.

The `update` function will refresh the clock, to be called by `setInterval` every second:

```
function update() {
  let clock = document.getElementById('clock');
  let date = new Date(); // (*)
  let hours = date.getHours();
  if (hours < 10) hours = '0' + hours;
  clock.children[0].innerHTML = hours;

  let minutes = date.getMinutes();
  if (minutes < 10) minutes = '0' + minutes;
  clock.children[1].innerHTML = minutes;

  let seconds = date.getSeconds();
  if (seconds < 10) seconds = '0' + seconds;
  clock.children[2].innerHTML = seconds;
}
```

In the line `(*)` we every time check the current date. The calls to `setInterval` are not reliable: they may happen with delays.

The clock-managing functions:

```
let timerId;

function clockStart() { // run the clock
  if (!timerId) { // only set a new interval if the clock is not running
    timerId = setInterval(update, 1000);
  }
  update(); // (*)
}

function clockStop() {
  clearInterval(timerId);
  timerId = null; // (**)
}
```

Please note that the call to `update()` is not only scheduled in `clockStart()`, but immediately run in the line `(*)`. Otherwise the visitor would have to wait till the first execution of `setInterval`. And the clock would be empty till then.

Also it is important to set a new interval in `clockStart()` only when the clock is not running. Otherways clicking the start button several times would set multiple concurrent intervals. Even worse – we would only keep the `timerID` of the last interval, losing references to all others. Then we wouldn't be able to stop the clock ever again! Note that we need to clear the `timerID` when the clock is stopped in the line `(**)`, so that it can be started again by running `clockStart()`.

[Abrir la solución en un entorno controlado.](#) ↗

A formulación

Insert the HTML in the list

When we need to insert a piece of HTML somewhere, `insertAdjacentHTML` is the best fit.

The solution:

```
one.insertAdjacentHTML('afterend', '<li>2</li><li>3</li>');
```

A formulación

Sort the table

The solution is short, yet may look a bit tricky, so here I provide it with extensive comments:

```
let sortedRows = Array.from(table.tBodies[0].rows) // 1
    .sort((rowA, rowB) => rowA.cells[0].innerHTML.localeCompare(rowB.cells[0].innerHTML));

table.tBodies[0].append(...sortedRows); // (3)
```

The step-by-step algorithm:

1. Get all `<tr>`, from `<tbody>`.
2. Then sort them comparing by the content of the first `<td>` (the name field).
3. Now insert nodes in the right order by `.append(...sortedRows)`.

We don't have to remove row elements, just "re-insert", they leave the old place automatically.

P.S. In our case, there's an explicit `<tbody>` in the table, but even if HTML table doesn't have `<tbody>`, the DOM structure always has it.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Estilos y clases

Crear una notificación

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Elementos de tamaño y desplazamiento

¿Qué es el desplazamiento desde la parte inferior?

La solución es:

```
let scrollTop = elem.scrollHeight - elem.scrollTop - elem.clientHeight;
```

En otras palabras: (altura total) menos (parte superior desplazada) menos (parte visible) – esa es exactamente la parte inferior desplazada.

[A formulación](#)

¿Qué es el ancho de la barra de desplazamiento?

Para obtener el ancho de la barra de desplazamiento, podemos crear un elemento con el scroll, pero sin bordes ni rellenos.

Entonces la diferencia entre su ancho completo `offsetWidth` y el ancho del area interior `clientWidth` será exactamente la barra de desplazamiento:

```
// crea un div con el scroll
let div = document.createElement('div');

div.style.overflowY = 'scroll';
```

```
div.style.width = '50px';
div.style.height = '50px';

// debe ponerlo en el documento, de lo contrario los tamaños serán 0
document.body.append(div);
let scrollWidth = div.offsetWidth - div.clientWidth;

div.remove();

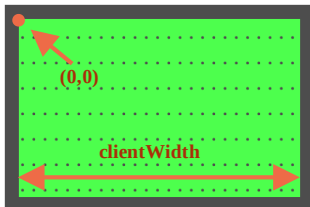
alert(scrollWidth);
```

A formulación

Coloca la pelota en el centro del campo.

La pelota tiene `position: absolute`. Significa que sus coordenadas `left/top` se miden desde el elemento posicionado más cercano, es decir `#field` (porque tiene `position: relative`).

Las coordenadas inician desde el interior de la esquina superior izquierda del campo:

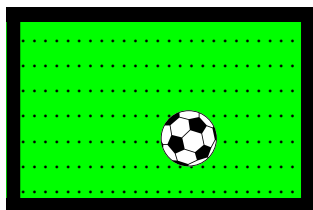


El interior del campo ancho/alto es `clientWidth/clientHeight`. Entonces el centro del campo tiene coordenadas `(clientWidth/2, clientHeight/2)`.

...Pero si configuramos `ball.style.left/top` a tales valores, entonces no la pelota en su conjunto, sino el borde superior izquierdo de la pelota estaría en el centro:

```
ball.style.left = Math.round(field.clientWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2) + 'px';
```

Así es como se ve:



Para alinear la pelota al centro con el centro del campo, deberíamos mover la pelota a la mitad de su ancho a la izquierda y a la mitad de su altura hacia arriba:

```
ball.style.left = Math.round(field.clientWidth / 2 - ball.offsetWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2 - ball.offsetHeight / 2) + 'px';
```

Ahora la pelota está finalmente centrada.

⚠️ Atención: ¡la trampa!

El código no funcionará seguramente mientras `` no tenga `width/height`:

```

```

Cuando el navegador no conoce el ancho/alto de una imagen (de un atributo o CSS), entonces este asume que es igual a `0` hasta que la imagen termine de cargarse.

Entonces el valor de `ball.offsetWidth` deberá ser `0` hasta que la imagen cargue. Eso conduce a coordenadas incorrectas en el código anterior.

Después de la primera carga, el navegador usualmente almacena en caché la imagen, y cuando se vuelva a cargar esta tendrá el tamaño inmediatamente. Pero en la primera carga el valor de `ball.offsetWidth` es `0`.

Deberíamos arreglar eso agregando `width/height` en ``:

```

```

...O indicar el tamaño en CSS:

```
#ball {  
  width: 40px;  
  height: 40px;  
}
```

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

La diferencia: CSS width versus clientWidth

Diferencias:

1. `clientWidth` es numérico, mientras `getComputedStyle(elem).width` retorna una cadena con `px` en el final.
2. `getComputedStyle` puede devolver un ancho no numérico como `"auto"` para un elemento en línea.
3. `clientWidth` es el contenido interior del área del elemento más los rellenos, mientras el ancho de CSS (con el estándar `box-sizing`) es el contenido interior del área *sin rellenos*.
4. Si hay una barra de desplazamiento y el navegador reserva espacio para esta, algunos navegadores restan ese espacio del ancho de CSS (por que no está disponible para el contenido), y otros no. La propiedad `clientWidth` es siempre la misma: el tamaño de la barra de desplazamiento se resta si está reservado.

[A formulación](#)

Coordenadas

Encuentra las coordenadas del campo en la ventana

Esquinas externas

Las esquinas externas son básicamente las que obtenemos de `elem.getBoundingClientRect()` ↗.

Las coordenadas de la esquina superior izquierda `answer1` y la esquina inferior derecha `answer2`:

```
let coords = elem.getBoundingClientRect();  
  
let answer1 = [coords.left, coords.top];  
let answer2 = [coords.right, coords.bottom];
```

Esquina interna y superior izquierda

Esta es diferente a la esquina externa por el ancho del borde. Una manera confiable de obtener la distancia es usando `clientLeft/clientTop`:

```
let answer3 = [coords.left + field.clientLeft, coords.top + field.clientTop];
```

Esquina interna e inferior derecha

En nuestro caso necesitamos sustraer la medida del borde de las coordenadas externas.

Podemos usar la forma de CSS:

```
let answer4 = [  
  coords.right - parseInt(getComputedStyle(field).borderRightWidth),  
  coords.bottom - parseInt(getComputedStyle(field).borderBottomWidth)  
];
```

Una forma alternativa puede ser agregando `clientWidth/clientHeight` a las coordenadas de la esquina superior izquierda. Probablemente sea incluso mejor:

```
let answer4 = [  
  coords.left + elem.clientLeft + elem.clientWidth,  
  coords.top + elem.clientTop + elem.clientHeight  
];
```

[Abrir la solución en un entorno controlado.](#)

A formulación

Muestra una nota cercana al elemento

En esta tarea sólo necesitamos calcular exactamente las coordenadas. Mira el código para más detalles.

Ten en cuenta: los elementos deben estar en el documento para leer `offsetHeight` y otras propiedades. Un elemento oculto (`display:none`) o fuera del documento no tiene medidas.

[Abrir la solución en un entorno controlado.](#)

A formulación

Muestra una nota cercana al elemento (absolute)

La solución realmente es muy simple:

- Usa `position:absolute` con CSS en lugar de `position:fixed` para `.note`.
- Usa la función `getCoords()` del capítulo [Coordenadas](#) para obtener las coordenadas relativas al documento.

[Abrir la solución en un entorno controlado.](#)

A formulación

Posiciona la nota adentro (absolute)

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Introducción a los eventos en el navegador

Ocultar con un click

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Ocultarse

Podemos usar `this` en el handler para referenciar “al propio elemento” aquí:

```
<input type="button" onclick="this.hidden=true" value="Click para ocultar">
```

[A formulación](#)

¿Qué handlers se ejecutan?

La respuesta: `1` y `2`.

El primer handler se activa porque no es removido por `removeEventListener`. Para remover el handler necesitamos pasar exactamente la función que fue asignada. Y en el código se pasa una función que luce igual pero es otra función.

Para remover un objeto de función necesitamos almacenar una referencia a él, así:

```
function handler() {  
  alert(1);  
}  
  
button.addEventListener("click", handler);  
button.removeEventListener("click", handler);
```

El handler `button.onclick` funciona independientemente y en adición a `addEventListener`.

[A formulación](#)

Mueve el balón por el campo

Primero necesitamos elegir un método para posicionar el balón.

No podemos usar `position:fixed` para ello, porque al desplazar la página se movería el balón del campo.

Así que deberíamos usar `position:absolute` y, para que el posicionamiento sea realmente sólido, hacer que `field` sea posicione a sí mismo.

Entonces el balón se posicionará en relación al campo:

```
#field {  
  width: 200px;
```

```

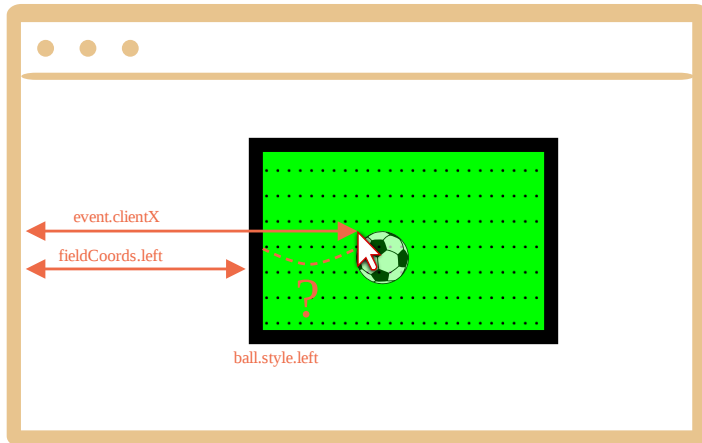
height: 150px;
position: relative;
}

#ball {
position: absolute;
left: 0; /* relativo al predecesor más cercano (field) */
top: 0;
transition: 1s all; /* Animación CSS para que left/top hagan al balón volar */
}

```

Lo siguiente es asignar el `ball.style.left/top` correcto. Ahora contienen coordenadas relativas al campo.

Aquí está la imagen:



Tenemos `event.clientX/clientY`, las cuales son las coordenadas del click relativas a la ventana.

Para obtener la coordenada `left` del click relativa al campo necesitamos restar el límite izquierdo del campo y el ancho del borde:

```

let left = event.clientX - fieldCoords.left - field.clientLeft;

```

Normalmente `ball.style.left` significa el “borde izquierdo del elemento” (el balón). Por lo que si asignamos ese `left`, entonces el borde del balón, no el centro, es el que se encontraría debajo del cursor del mouse.

Necesitamos mover la mitad del ancho del balón a la izquierda y la mitad del alto hacia arriba para que quede en el centro.

Por lo que el `left` final debería ser:

```

let left = event.clientX - fieldCoords.left - field.clientLeft - ball.offsetWidth/2;

```

La coordenada vertical es calculada usando la misma lógica.

Por favor, nota que el ancho/alto del balón se debe conocer al momento que accedemos a `ball.offsetWidth`. Se debe especificar en HTML o CSS.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Crear un menú deslizable

HTML/CSS

Primero hay que crear el HTML y CSS.

Un menú es un componente gráfico independiente en la página, por lo que es mejor colocarlo en un solo elemento del DOM.

Una lista de elementos del menú se puede diseñar como una lista `ul/li`.

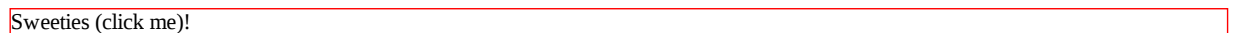
Aquí está la estructura de ejemplo:

```
<div class="menu">
  <span class="title">Sweeties (click me)!</span>
  <ul>
    <li>Cake</li>
    <li>Donut</li>
    <li>Honey</li>
  </ul>
</div>
```

Usamos `` para el título, porque `<div>` tiene un `display: block` implícito en él, y va a ocupar 100% del ancho horizontal.

Así:


```
<div style="border: solid red 1px" onclick="alert(1)">Sweeties (click me)!</div>
```



Entonces si establecemos `onclick` en él, detectará los clics a la derecha del texto.

Como `` tiene un `display: inline` implícito, ocupa exactamente el lugar suficiente para que quepa todo el texto:

```
<span style="border: solid red 1px" onclick="alert(1)">Sweeties (click me)!</span>
```



Alternar el menú

Alternar el menú debería cambiar la flecha y mostrar/ocultar la lista del menú.

Todos estos cambios son perfectamente controlados con CSS. En JavaScript debemos etiquetar el estado actual del menú agregando/eliminando la clase `.open`.

Sin él, el menú se cerrará:

```
.menu ul {
  margin: 0;
  list-style: none;
  padding-left: 20px;
  display: none;
}

.menu .title::before {
  content: '►';
  font-size: 80%;
  color: green;
}
```

...Y con `.open` la flecha cambia y aparece la lista:

```
.menu.open .title::before {
  content: '▼ ';
}

.menu.open ul {
  display: block;
}
```

[Abrir la solución en un entorno controlado.](#)

A formulación

Agregar un botón de cierre

Para agregar el botón podemos usar cualquiera de las opciones `position: absolute` (y hacer el panel `position: relative`) o `float: right`. El `float: right` tiene la ventaja de que el botón no se encima con el texto, pero `position: absolute` da más libertad. Entonces la elección es tuya.

Luego, para cada panel, el código puede ser así:

```
pane.insertAdjacentHTML("afterbegin", '<button class="remove-button">[x]</button>');
```

Luego el `<button>` se convierte en `pane.firstChild`, por lo que podemos agregarle un controlador como este:

```
pane.firstChild.onclick = () => pane.remove();
```

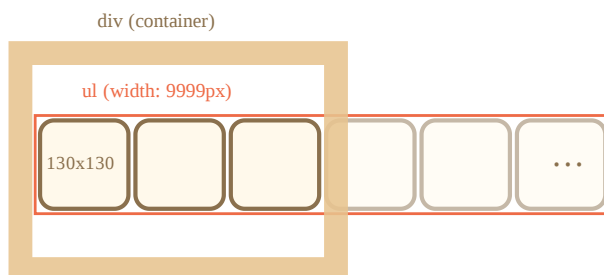
[Abrir la solución en un entorno controlado.](#)

A formulación

Carrusel

La cinta de imágenes se puede representar como una lista `ul/li` de imágenes ``.

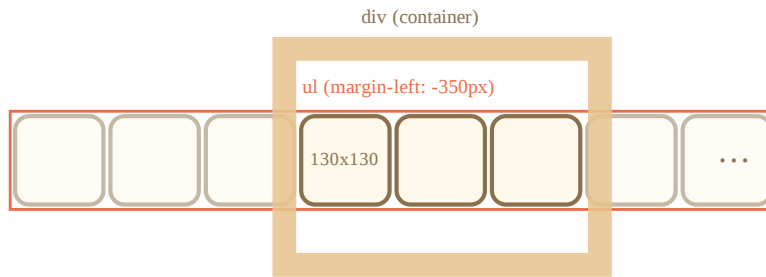
Normalmente dicha cinta es ancha pero colocamos un tamaño fijo `<div>` alrededor para “cortarla”, de modo que solo una parte de la cinta sea visible:



Para que la lista se muestre horizontalmente debemos aplicar las propiedades CSS correctas para ``, como `display: inline-block`.

Para `` también deberíamos ajustar `display`, ya que es `inline` por default. Hay espacio adicional reservado debajo de los “letter tails”, por lo que podemos usar `display: block` para eliminarlo.

Para hacer el desplazamiento, podemos cambiar ``. Hay muchas formas de hacerlo, por ejemplo, cambiando `margin-left` o (para mejor rendimiento) usando `transform: translateX()`:



El `<div>` exterior tiene un ancho fijo, por lo que se cortan las imágenes “extra”.

Todo el carrusel es un “componente gráfico” autónomo en la página, por lo que será mejor que lo envuelva en un solo elemento `<div class="carousel">` y le apliquemos estilo.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Delegación de eventos

Ocultar mensajes con delegación

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Menú de árbol

La solución tiene dos partes.

1. Envuelve cada nodo de título del árbol dentro de ``. Luego podemos aplicarles CSS-style en `:hover` y manejar los clics exactamente sobre el texto, porque el ancho de `` es exactamente el ancho del texto (no lo será si no lo tiene).
2. Establece el manejador al nodo raíz del `tree` y maneja los clics en aquellos títulos ``.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Tabla ordenable

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Comportamiento: Tooltip

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Acciones predeterminadas del navegador

¿Por qué "return false" no funciona?

Cuando el navegador lee un atributo `on*` como `onclick`, crea el controlador a partir de su contenido.

Para `onclick="handler()"` la función será:

```
function(event) {  
  handler() // el contenido de onclick  
}
```

Ahora podemos ver que el valor devuelto por `handler()` no se usa y no afecta el resultado.

La solución es simple:

```
<script>  
  function handler() {  
    alert("...");  
    return false;  
  }  
</script>  
  
<a href="https://w3.org" onclick="return handler()">w3.org</a>
```

También podemos usar `event.preventDefault()`, así:

```
<script>  
  function handler(event) {  
    alert("...");  
    event.preventDefault();  
  }  
</script>  
  
<a href="https://w3.org" onclick="handler(event)">w3.org</a>
```

A formulación

Captura enlaces en el elemento

Ese es un gran uso para el patrón de delegación de eventos.

En la vida real, en lugar de preguntar, podemos enviar una solicitud de "logging" al servidor que guarda la información sobre dónde se fue el visitante. O podemos cargar el contenido y mostrarlo directamente en la página (si está permitido).

Todo lo que necesitamos es capturar el `contents.onclick` y usar `confirm` para preguntar al usuario. Una buena idea sería usar `link.getAttribute('href')` en lugar de `link.href` para la URL. Consulte la solución para obtener más detalles.

[Abrir la solución en un entorno controlado.](#) ↗

A formulación

Galería de imágenes

La solución es asignar el controlador al contenedor y realizar un seguimiento de los clics. Si haces clic en el enlace `<a>`, cambias `src` de `#largeImg` por el `href` de la miniatura.

[Abrir la solución en un entorno controlado.](#) ↗

A formulación

Eventos del Mouse

Lista seleccionable

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Moviendo el mouse: mouseover/out, mouseenter/leave

Comportamiento mejorado de un tooltip

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Tooltip "inteligente"

El algoritmo luce simple:

1. Coloca los controladores `onmouseover/out` en el elemento. Aquí también podemos usar `onmouseenter/leave` pero son menos universales, no funcionan si introducimos delegaciones.
2. Cuando el cursor ingrese al elemento debes medir la velocidad en `mousemove`.
3. Si la velocidad es lenta hay que ejecutar `over`.
4. Si estamos saliendo del elemento, y `over` ya se había ejecutado, ahora ejecutamos `out`.

¿Pero cómo mediremos la velocidad?

La primera idea puede ser: correr una función cada `100ms` y medir la distancia entre la coordenada anterior y la actual. Si es pequeña entonces la velocidad fue rápida.

Desafortunadamente no hay manera para obtener las coordenadas actuales del mouse en JavaScript. No existe algo así como `getCurrentMouseCoordinates()`.

La única manera es registrando los eventos del mouse, como `mousemove`, y tomar las coordenadas del objeto del evento.

Entonces configuremos un `mousemove` para registrar las coordenadas y recordarlas. Y entonces las comparamos, una por cada `100ms`.

PD. Toma nota: El test de la solución usa `dispatchEvent` para ver si el tooltip funciona bien.

[Abrir la solución con pruebas en un entorno controlado.](#) ↗

[A formulación](#)

Arrastrar y Soltar con eventos del ratón

Control deslizante

Como podemos ver en el HTML/CSS, la barra de desplazamiento es un `<div>` con un fondo de color, que contiene un pasador: otro `<div>` con `position:relative`.

Para posicionar el pasador usamos `position: relative`, para proveer las coordenadas relativas a su padre, aquí es más conveniente que `position: absolute`.

En este caso implementamos un Arrastrar y Soltar horizontal limitado por el ancho.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Arrastrar super héroes por el campo

Para arrastrar el elemento podemos usar `position: fixed`, esto hace las coordenadas más fáciles de manejar. Al final deberíamos devolverla a `position: absolute` para fijar el elemento en el documento.

Cuando las coordenadas están en el tope/fondo de la ventana, usamos `window.scrollTo` para desplazarla.

Más detalles en el código, en los comentarios.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Keyboard: keydown and keyup

Extended hotkeys

We should use two handlers: `document.onkeydown` and `document.onkeyup`.

Let's create a set `pressed = new Set()` to keep currently pressed keys.

The first handler adds to it, while the second one removes from it. Every time on `keydown` we check if we have enough keys pressed, and run the function if it is so.

[Abrir la solución en un entorno controlado.](#)

[A formulación](#)

Desplazamiento

Página sin fin

El núcleo de la solución es una función que añade más fechas a la página (o carga más cosas en la vida real) mientras estamos en el final de la página.

Podemos llamarlo inmediatamente o agregarlo como un manejador de `window.onscroll`.

La pregunta más importante es: "¿Cómo detectamos que la página se desplaza hasta el fondo?"

Usaremos las coordenadas de la ventana.

El documento está representado (y contenido) dentro de la etiqueta `<html>`, que es `document.documentElement`.

Podemos obtener las coordenadas relativas a la ventana de todo el documento como `document.documentElement.getBoundingClientRect()`, la propiedad `bottom` será la coordenada relativa a la ventana del fondo del documento.

Por ejemplo, si la altura de todo el documento es `2000px`, entonces:

```
// cuando estamos en la parte superior de la página
// window-relative top = 0 (relativo a la ventana, límite superior = 0 )
document.documentElement.getBoundingClientRect().top = 0

// window-relative bottom = 2000 (relativo a la ventana, límite inferior = 2000)
// el documento es largo, así que probablemente esté más allá del fondo de la ventana
document.documentElement.getBoundingClientRect().bottom = 2000
```

Si nos desplazamos `500px` abajo, entonces:

```
// la parte superior del documento está 500px por encima de la ventana
document.documentElement.getBoundingClientRect().top = -500
// la parte inferior del documento está 500px más cerca
document.documentElement.getBoundingClientRect().bottom = 1500
```

Cuando nos desplazamos hasta el final, asumiendo que la altura de la venta es `600px`:

```
// La parte superior del documento está 1400px sobre la ventana
document.documentElement.getBoundingClientRect().top = -1400
// la parte inferior del documento está a 600px debajo de la ventana
document.documentElement.getBoundingClientRect().bottom = 600
```

Tened en cuenta que el fondo del documento `bottom` nunca puede ser `0`, porque nunca llega a la parte superior de la ventana. El límite más bajo de la coordenada `bottom` es la altura de la ventana (asumimos que es `600`), no podemos desplazarla más hacia arriba.

Podemos obtener la altura de la ventana con `document.documentElement.clientHeight`.

Para nuestra tarea, necesitamos saber cuando tenemos el final del documento a unos `100px` (esto es: `600 - 700px`, si la altura es de `600`).

Así que aquí está la función:

```
function populate() {
  while(true)
  {
    // final del documento
    let windowRelativeBottom = document.documentElement.getBoundingClientRect().bottom;

    // si el usuario no se ha desplazado lo suficiente (> 100px hasta el final)
    if (windowRelativeBottom > document.documentElement.clientHeight + 100) break;
    // vamos añadir más datos
    document.body.insertAdjacentHTML("beforeend", `<p>Date: ${new Date()}</p>`);
  }
}
```

[Abrir la solución en un entorno controlado.](#)

A formulación

Botón para subir/bajar

[Abrir la solución en un entorno controlado.](#)

A formulación

Cargar imágenes visibles

El manejador `onscroll` debería comprobar qué imágenes son visibles y mostrarlas.

También queremos que se ejecute cuando se cargue la página, para detectar las imágenes visibles inmediatamente y cargarlas.

El código debería ejecutarse cuando se cargue el documento, para que tenga acceso a su contenido.

O ponerlo en la parte inferior del `<body>`:

```
// ...el contenido de la página está arriba...

function isVisible(elem) {

  let coords = elem.getBoundingClientRect();

  let windowHeight = document.documentElement.clientHeight;

  // ¿El borde superior del elemento es visible?
  let topVisible = coords.top > 0 && coords.top < windowHeight;

  // ¿El borde inferior del elemento es visible?
  let bottomVisible = coords.bottom < windowHeight && coords.bottom > 0;

  return topVisible || bottomVisible;
}
```

La función `showVisible()` utiliza el control de visibilidad, implementado por `isVisible()`, para cargar imágenes visibles:

```
function showVisible() {
  for (let img of document.querySelectorAll('img')) {
    let realSrc = img.dataset.src;
    if (!realSrc) continue;

    if (isVisible(img)) {
      img.src = realSrc;
      img.dataset.src = '';
    }
  }
}

showVisible();
window.onscroll = showVisible;
```

P.D. La solución tiene una variante de `isVisible` que “precarga” imágenes que están dentro de 1 página por encima/debajo del desplazamiento del documento actual.

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Propiedades y Métodos de Formularios

Añade una opción al select

La solución, paso a paso:

```
<select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>

<script>
  // 1)
  let selectedOption = genres.options[genres.selectedIndex];
```

```
alert( selectedOption.value );

// 2)
let newOption = new Option("Classic", "classic");
genres.append(newOption);

// 3)
newOption.selected = true;
</script>
```

[A formulación](#)

Focusing: focus/blur

Editable div

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Edit TD on click

1. On click – replace `innerHTML` of the cell by `<textarea>` with same sizes and no border. Can use JavaScript or CSS to set the right size.
2. Set `textarea.value` to `td.innerHTML`.
3. Focus on the textarea.
4. Show buttons OK/CANCEL under the cell, handle clicks on them.

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Keyboard-driven mouse

We can use `mouse.onclick` to handle the click and make the mouse “moveable” with `position:fixed`, then `mouse.onkeydown` to handle arrow keys.

The only pitfall is that `keydown` only triggers on elements with focus. So we need to add `tabindex` to the element. As we’re forbidden to change HTML, we can use `mouse.tabIndex` property for that.

P.S. We also can replace `mouse.onclick` with `mouse.onfocus`.

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Eventos: change, input, cut, copy, paste

Calculadora de depósito

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Formularios: evento y método submit

Formulario modal

Una ventana modal puede ser implementada utilizando un `<div id="cover-div">` semi-transparente que cubra completamente la ventana, como a continuación:

```
#cover-div {  
  position: fixed;  
  top: 0;  
  left: 0;  
  z-index: 9000;  
  width: 100%;  
  height: 100%;  
  background-color: gray;  
  opacity: 0.3;  
}
```

Debido a que el `<div>` cubre toda la ventana, recibe todos los clicks, en vez de la página tras él.

También podemos evitar el scroll en la página utilizando `body.style.overflowY='hidden'`.

El formulario no debe estar en el `<div>` sino junto a él, porque no queremos que tenga `opacity`.

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)

Carga de recursos: onload y onerror

Cargando imágenes con una un función de retorno (callback)

El algoritmo:

1. Crear una `img` para cada fuente.
2. Agregar los eventos `onload/onerror` para cada imagen.
3. Incrementar el contador cuando el evento `onload` o el evento `onerror` se dispare.
4. Cuando el valor del contador es igual a la cantidad de fuentes, hemos terminado: `callback()`.

[Abrir la solución en un entorno controlado.](#) ↗

[A formulación](#)