UD 4.2 INTEGRACIÓN DE JAVASCRIPT EN EL NAVEGADOR

Índex

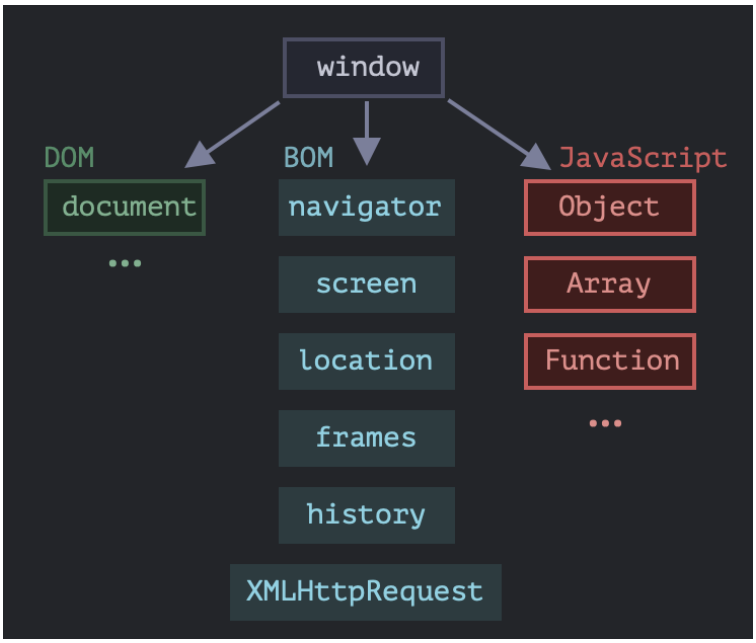
1. ENTORNO DEL NAVEGADOR, ESPECIFICACIONES

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

Una plataforma puede ser un navegador, un servidor web u otro host (“anfitrión”); 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 a esto entorno de host.

Un entorno host proporciona sus propios objetos 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, etc.

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:

- Primero, es un objeto global para el código JavaScript. Todo se ejecuta dentro de dicho objeto.
- Segundo, representa la “ventana del navegador” y proporciona métodos para controlarla.

Por ejemplo, podemos usarlo como objeto global:

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

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

Y podemos usarlo como una ventana del navegador. Para ver la altura de la ventana:

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

1.1. DOM (Modelo de Objetos del Documento)

Document Object Model, o DOM, representa todo el contenido de la página como objetos que pueden ser modificados.

El objeto document es el punto de entrada a 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.

1.2. BOM (Modelo de Objetos del Navegador)

El Modelo de Objetos del Navegador (Browser Object Model, 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: acerca del navegador actual, y navigator.platform: acerca de la plataforma (ayuda a distinguir Windows/Linux/Mac, etc.).

- El objeto location nos permite leer la URL actual y puede redirigir el navegador a una nueva.

Aquí vemos cómo podemos usar el objeto location:

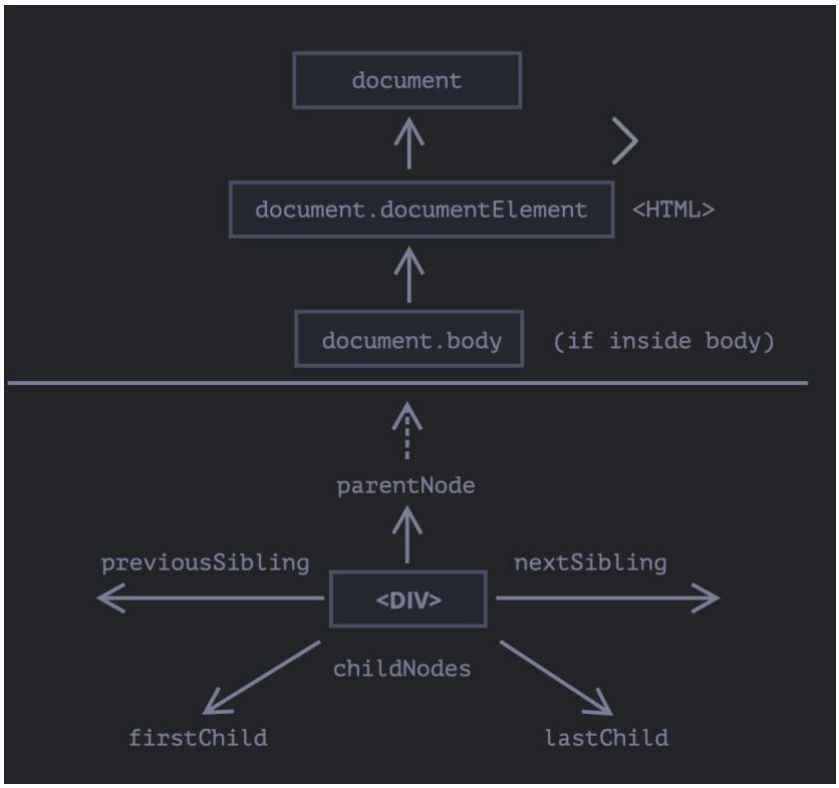
```
alert(location.href); // muestra la URL actual
if (confirm("Ir a wikipedia?")) {
  location.href = "https://wikipedia.org"; // redirigir el navegador a otra URL
}
```

2. 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.

2.1. 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.

2.2. 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.

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.

2.3.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
```

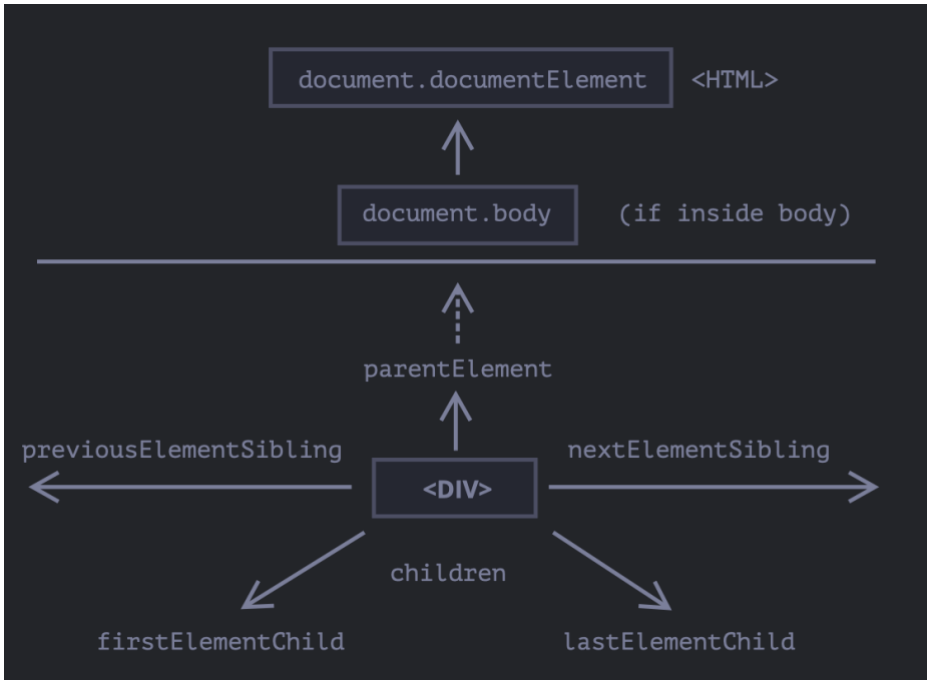
2.4. Navegación solo por elementos

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.

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>
```

3. 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.

3.1.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 guion 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>
```

3.2.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 <li> 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.

3.3. 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.

3.4. 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 2</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>
```

3.5.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.

- 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.
- 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="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>
```

4. PROPIEDADES DEL NODO: TIPO, ETIQUETA Y CONTENIDO

4.1.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.

4.2.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>
```

4.3.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>
```

4.4.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>
```

- El primer obtiene el nombre "como HTML": todas las etiquetas se convierten en etiquetas, por lo que vemos el nombre en negrita.
- El segundo 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.

4.5.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.

5. MODIFICANDO EL DOCUMENTO

La modificación del DOM es la clave para crear páginas “vivas”, dinámicas.

Aquí veremos cómo crear nuevos elementos “al vuelo” y modificar el contenido existente de la página.

5.1. Ejemplo: mostrar un mensaje

Hagamos una demostración usando un ejemplo. Añadiremos un mensaje que se vea más agradable que un alert.

Así es como se verá:

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

<div class="alert">
  <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
</div>
```

¡Hola! Usted ha leído un importante mensaje.

Eso fue el ejemplo HTML. Ahora creemos el mismo div con JavaScript (asumiendo que los estilos ya están en HTML/CSS).

5.2. Creando el mensaje

Crear el div de mensaje toma 3 pasos:

```
// 1. Crear elemento <div>
let div = document.createElement('div');


// 2. Establecer su clase a "alert"
div.className = "alert";


// 3. Agregar el contenido
div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje.";
```

Hemos creado el elemento. Pero hasta ahora solamente está en una variable llamada div, no aún en la página, y no la podemos ver.

5.3. Métodos de inserción

Para hacer que el div aparezca, necesitamos insertarlo en algún lado dentro de document. Por ejemplo, en el elemento <body>, referenciado por document.body.

Hay un método especial append para ello: document.body.append(div).

El código completo:

```
<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>¡Hola!</strong> Usted ha leído un importante.";
  document.body.append(div);
</script>
```

Aquí usamos el método append sobre document.body, pero podemos llamar append sobre cualquier elemento para poner otro elemento dentro de él. Por ejemplo, podemos añadir algo a llamando div.append(anotherElement).

5.4. Eliminación de nodos

Para quitar un nodo, tenemos el método node.remove().

Hagamos que nuestro mensaje desaparezca después de un segundo:

```
<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>¡Hola!</strong> Usted ha leído un importante mensaje.";

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

6. 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:
2. Escribir las propiedades directamente en 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.

6.1.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>
```

6.2.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 sí 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');
```

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.

También hay un método especial para eso, elem.style.removeProperty('style property'). Así, podemos quitar una propiedad:

```
document.body.style.background = 'red'; //establece background a rojo


setTimeout(() => document.body.style.removeProperty('background'), 1000); //
quitar background después de 1 segundo
```

6.3. 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.

6.4. 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>
   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>
```

7. TAMAÑO DE ELEMENTOS Y DESPLAZAMIENTO

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

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

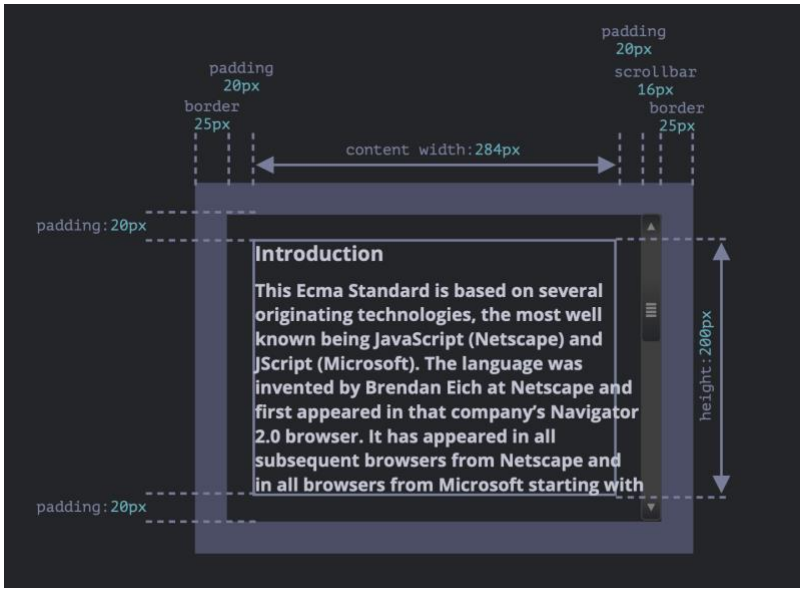
7.1. Elemento de muestra

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

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

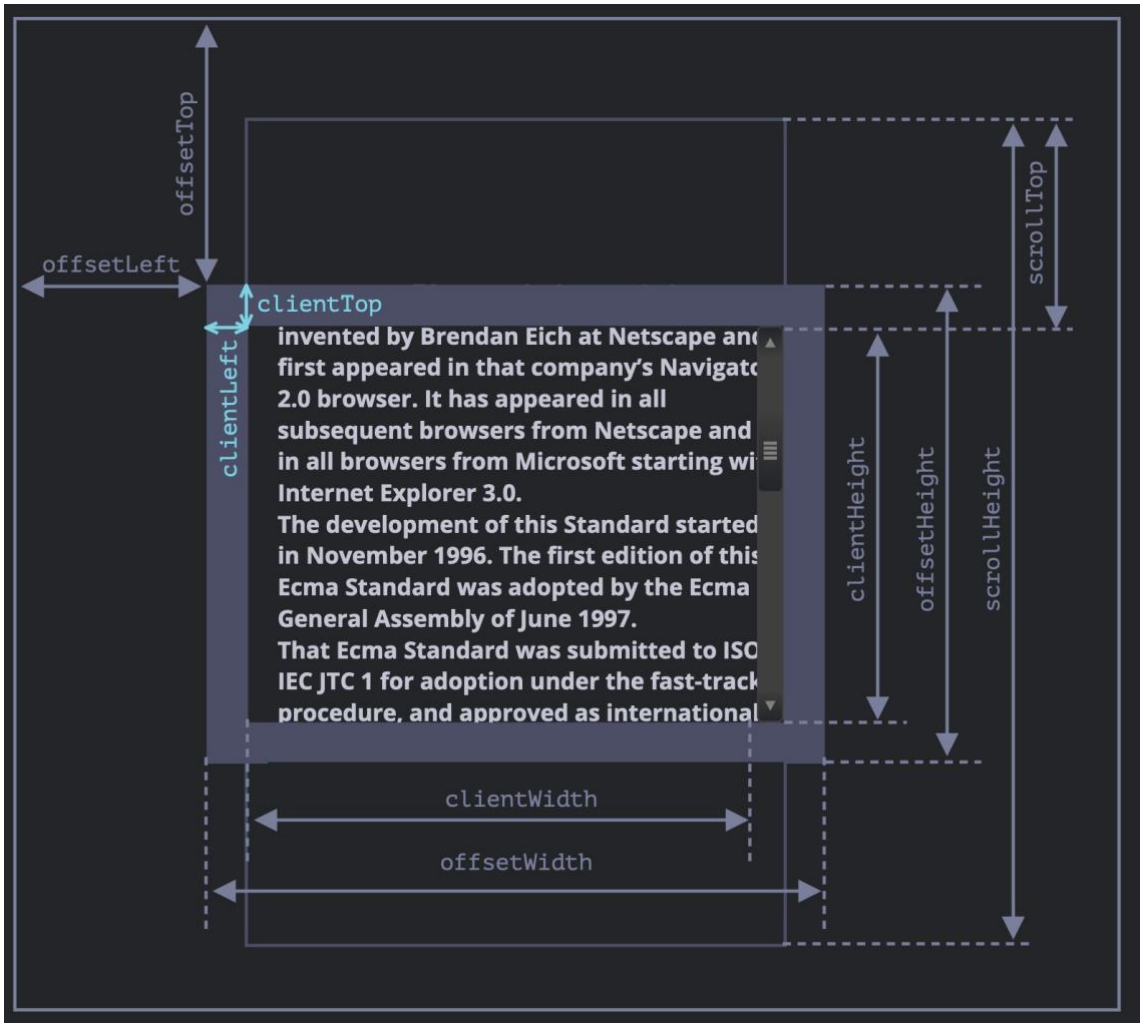
Este tiene borde, relleno y desplazamiento. El conjunto completo de funciones. No hay márgenes porque no son parte del elemento en sí, y no tienen propiedades especiales.

El elemento tiene este aspecto:



7.2.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 exterior del elemento.

7.3.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:

- Posicionado por CSS (position es absolute, relative, fixed o sticky), o…
- <td>, <th>, or <table>, o…
- <body>.

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

En el siguiente ejemplo el más interno tiene <main> como offsetParent, y offsetLeft/offsetTop lo 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 (nota: es un número, no un string "180px")
 alert(example.offsetTop); // 180
</script>
```

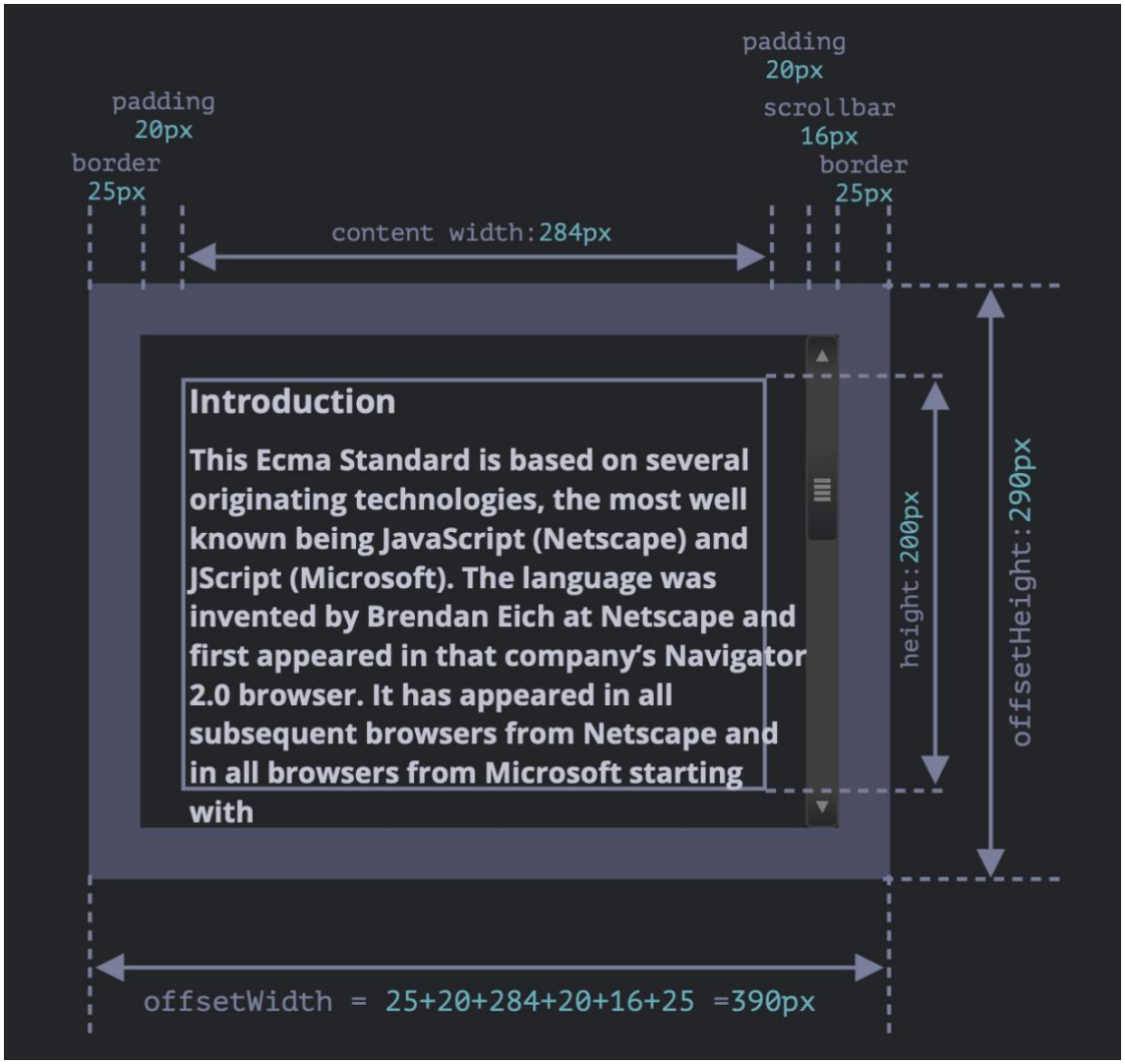
Hay varias ocasiones en la que offsetParent es null:

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

7.4.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.

7.5.clientTop/Left

Dentro del elemento, tenemos los bordes.

Para medirlos, están las propiedades clientTop y clientLeft.

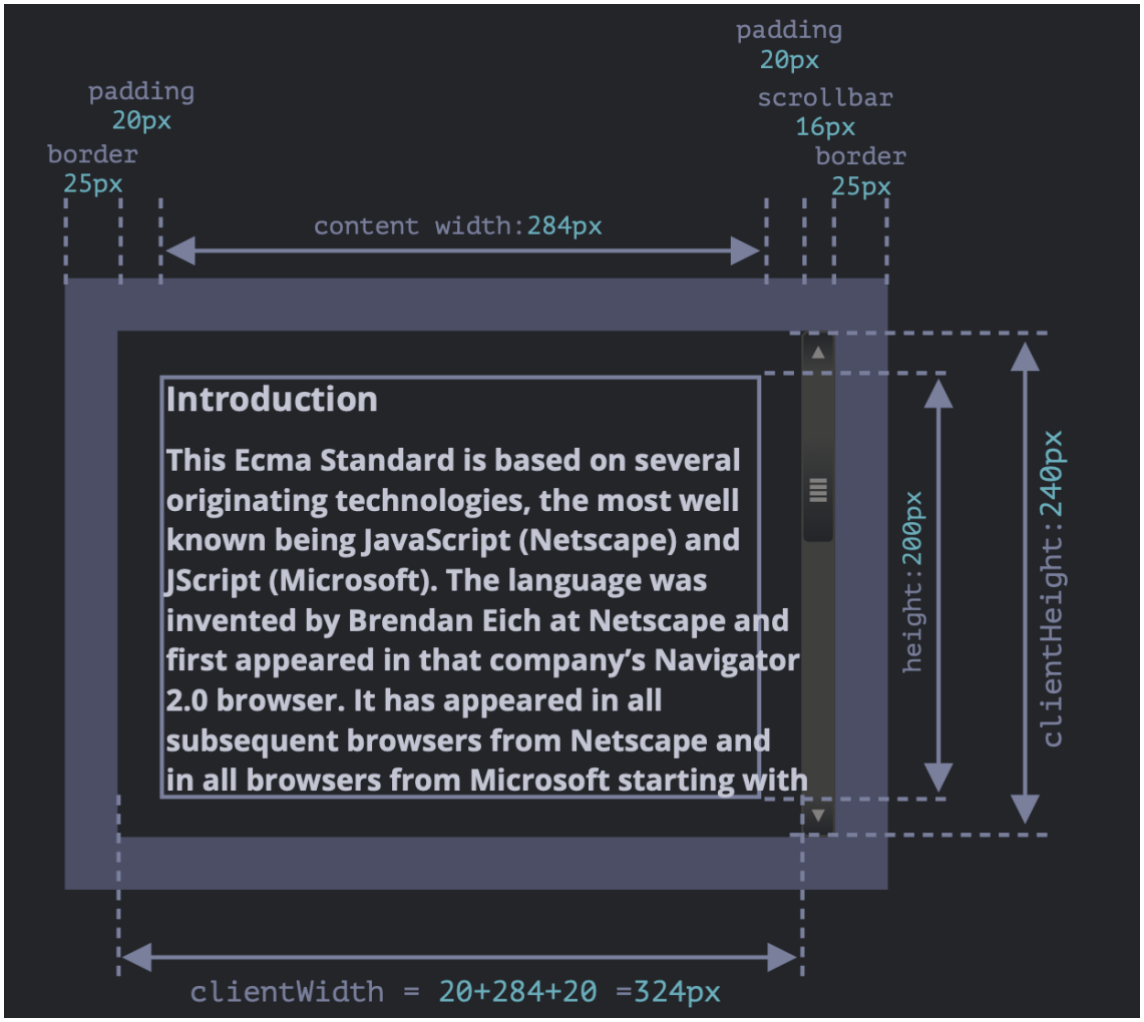
En nuestro ejemplo:

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

7.6.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:

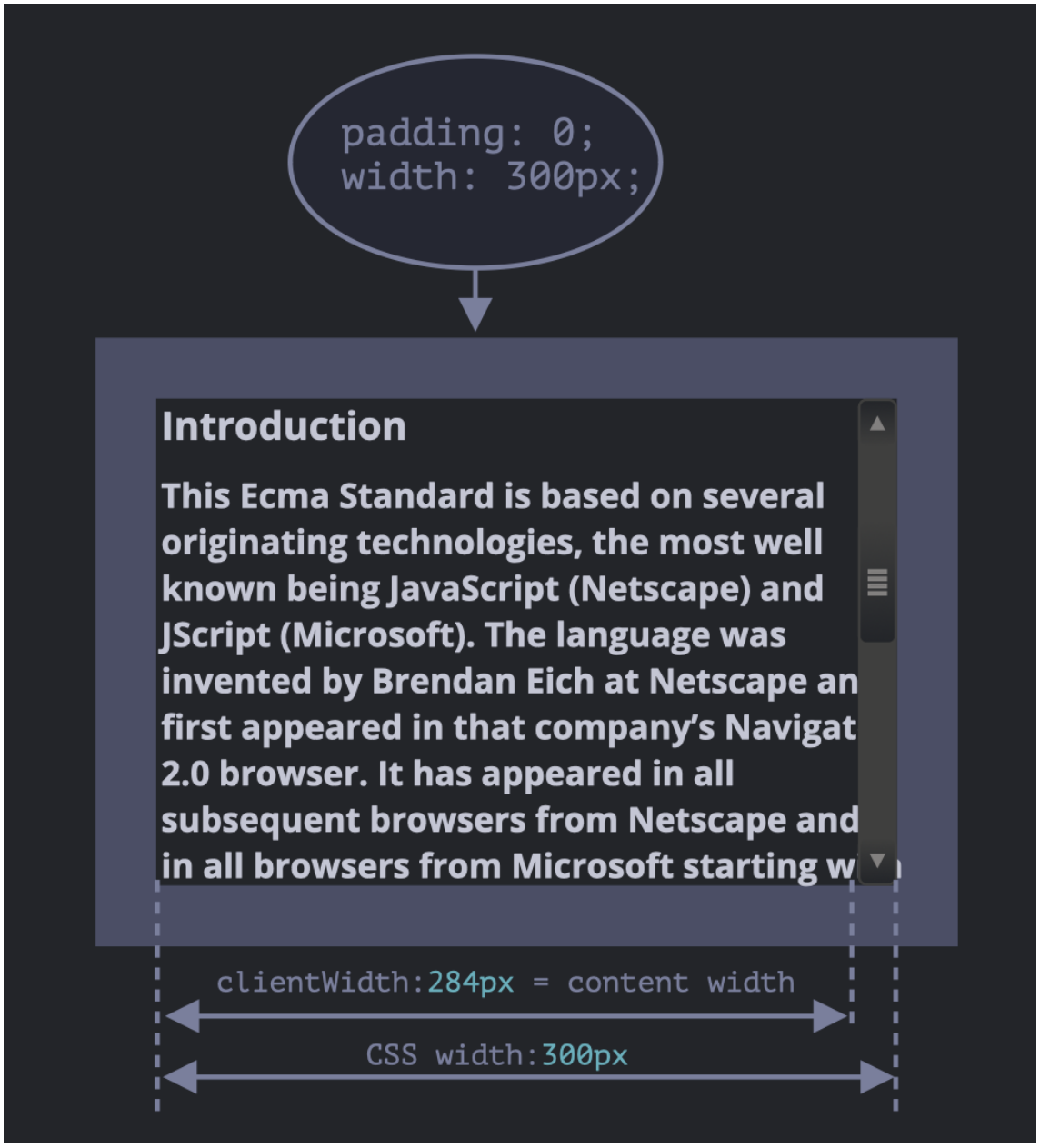


En la imagen de arriba, consideramos primero clientHeight.

No hay una barra de desplazamiento horizontal, por lo que es exactamente la suma de lo que está dentro de los bordes: CSS-height 200px más el relleno superior e inferior (2 * 20px) totaliza 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.

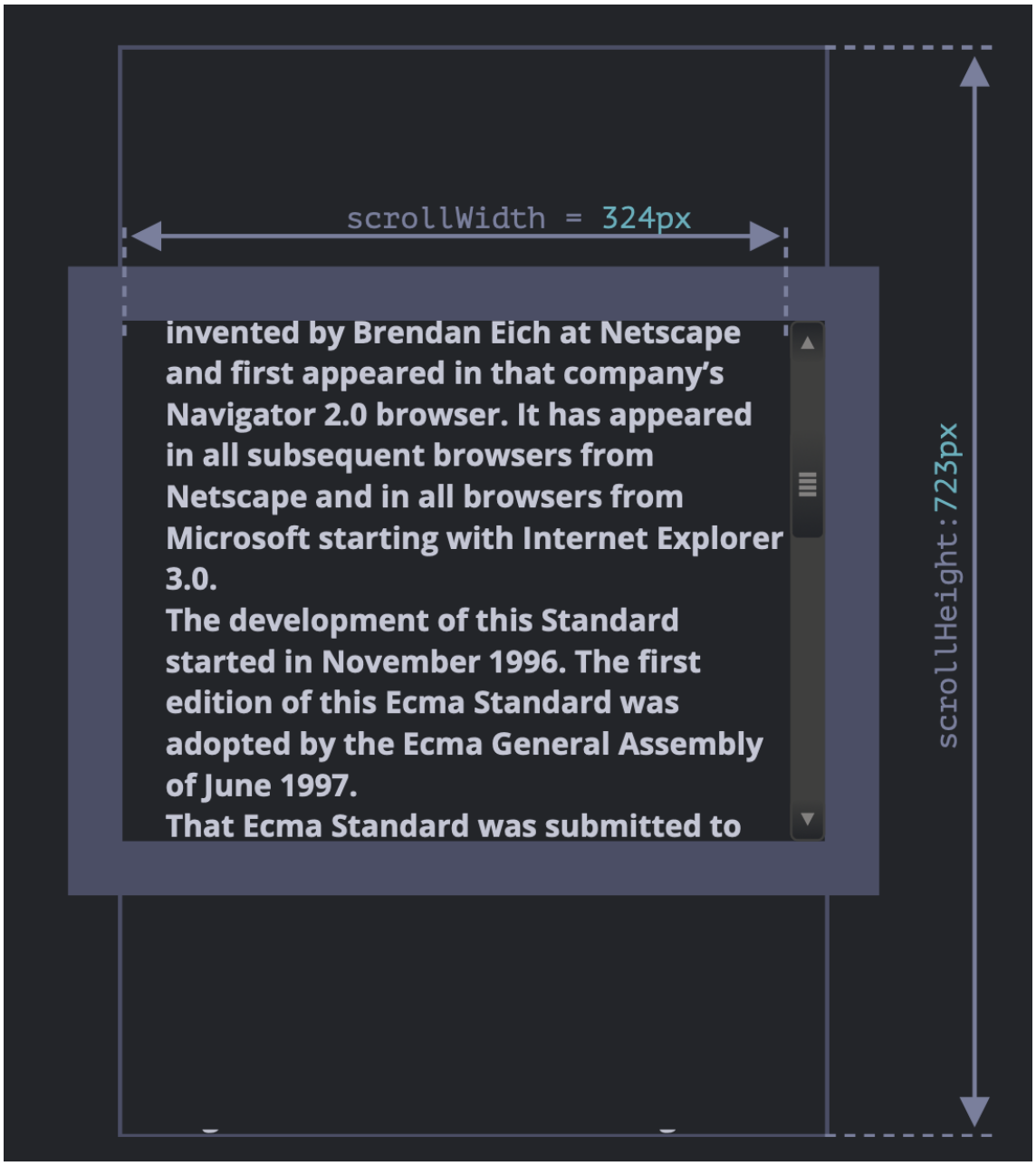
Si no hay rellenos, entonces clientWidth/Height es exactamente el área de contenido, dentro de los bordes y la barra de desplazamiento (si la hay).



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

7.7.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.

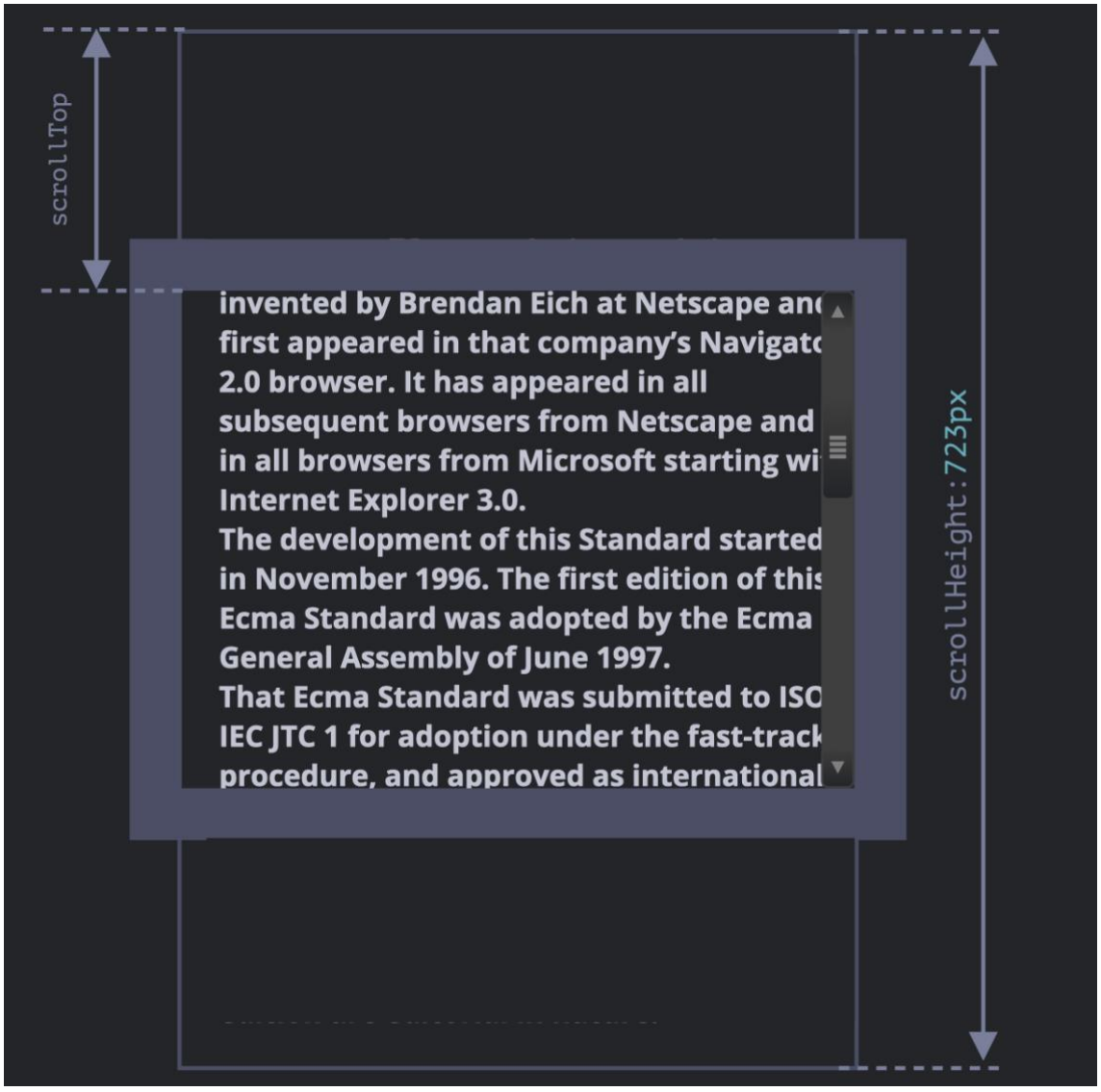
Como esto:

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

7.8. scrollLeft/scrollTop

Las propiedades scrollLeft/scrollTop 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”.

8. TAMAÑO DE VENTANA Y DESPLAZAMIENTO

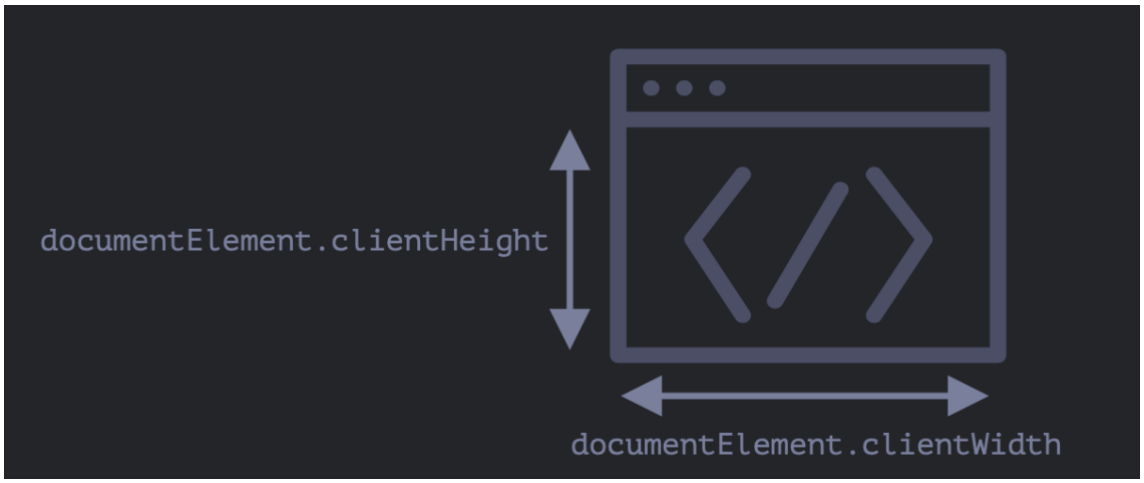
¿Cómo encontramos el ancho y el alto de la ventana del navegador? ¿Cómo obtenemos todo el ancho y la altura del documento, incluida la parte desplazada? ¿Cómo desplazamos la página usando JavaScript?

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.

8.1. Ancho/alto de la ventana

Para obtener el ancho y alto de la ventana, podemos usar clientWidth / clientHeight de document.documentElement:

Por ejemplo, la altura de su ventana:




```
alert(document.documentElement.clientHeight)
```

8.2. 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.scrollWidth / scrollHeight.

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: ' + scrollHeight);
```

¿Por qué? Mejor no preguntes. Estas inconsistencias provienen de tiempos antiguos, no una lógica “inteligente”.

8.3. 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.
```

8.4. Desplazamiento: scrollTo, scrollBy, scrollIntoView

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.