Manipulación y eventos del DOM

Curso de Fundamentos

Introducción

Una de las capacidades más exclusivas y útiles de JavaScript es su capacidad de manipular el DOM. Pero, ¿qué *es* el DOM y cómo podemos modificarlo? Vayamos directo al grano...

Resumen de la lección

Esta sección contiene una descripción general de los temas que aprenderá en esta lección.

- Explique qué es el DOM en relación con una página web.
- Explique la diferencia entre un “nodo” y un “elemento”.
- Explique cómo orientar los nodos con “selectores”.
- Explicar los métodos básicos para encontrar, agregar, eliminar y alterar nodos DOM.
- Explique la diferencia entre una “lista de nodos” y una “matriz de nodos”.
- Explique qué es el “burbujeo” y cómo funciona.

Modelo de objeto de documento

El DOM (o Document Object Model) es una representación en forma de árbol del contenido de una página web: un árbol de “nodos” con diferentes relaciones según cómo estén dispuestos en el documento HTML. Hay muchos tipos de nodos, la mayoría de los cuales no se utilizan comúnmente. En esta lección nos centraremos en los nodos de “elemento”, que se utilizan principalmente para manipular el DOM.

```html
<div id="container">
  <div class="display"></div>
  <div class="controls"></div>
</div>
```

En el ejemplo anterior, the `<div class="display"></div>` es un "hijo" de `<div id="container"></div>` y un "hermano" de `<div class="controls"></div>` . Piense en ello como un árbol genealógico. `<div id="container"></div>` es un **padre** , con sus **hijos** en el siguiente nivel, cada uno en su propia "rama".

Apuntar a nodos con selectores

Cuando se trabaja con el DOM, se utilizan "selectores" para seleccionar los nodos con los que se desea trabajar. Se puede utilizar una combinación de selectores de estilo CSS y propiedades de relación para seleccionar los nodos que se desean. Empecemos con los selectores de estilo CSS. En el ejemplo anterior, se pueden utilizar los siguientes selectores para hacer referencia a `<div class="display"></div>` :

- `div.display`
- `.display`
- `#container > .display`
- `div#container > div.display`

También puede utilizar selectores relacionales (es decir, `firstElementChild` o `lastElementChild` , etc.) con propiedades especiales propiedad de los nodos.

```javascript
// selects the #container div (don't worry about the syntax, w
const container = document.querySelector("#container");

// selects the first child of #container => .display
const display = container.firstElementChild;
```

```
    console.log(display);  // <div class="display"></div>
```

```
// selects the .controls div
const controls = document.querySelector(".controls");

// selects the prior sibling => .display
const display = controls.previousElementSibling;
console.log(display); // <div class="display"></div>
```

Entonces, estás identificando un determinado nodo en función de sus relaciones con los nodos que lo rodean.

Métodos DOM

Cuando un navegador web analiza el código HTML, este se convierte al DOM, como se mencionó anteriormente. Una de las principales diferencias es que estos nodos son objetos JavaScript que tienen muchas propiedades y métodos asociados. Estas propiedades y métodos son las herramientas principales que vamos a utilizar para manipular nuestra página web con JavaScript.

Selectores de consultas

- `element.querySelector(selector)` - devuelve una referencia a la primera coincidencia del *selector* .
- `element.querySelectorAll(selectors)` - devuelve una “NodeList” que contiene referencias a todas las coincidencias de los *selectores* .

> Hay otras consultas más específicas que ofrecen posibles beneficios de rendimiento (marginales), pero no las analizaremos ahora.

Es importante recordar que al usar querySelectorAll, el valor de retorno **no** es una matriz. Parece una matriz y actúa como tal, pero en realidad es una “NodeList”. La gran diferencia es que faltan varios métodos de matriz en las NodeList. Una solución, si surgen problemas, es convertir la NodeList en una matriz. Puede hacerlo con `Array.from()` o con el operador spread.

Element creation

- `document.createElement(tagName, [options])` - creates a new element of tag type tagName. `[options]` in this case means you can add some optional parameters to the function. Don't worry about these at this point.

```
const div = document.createElement("div");
```

This function does NOT put your new element into the DOM - it creates it in memory. This is so that you can manipulate the element (by adding styles, classes, ids, text, etc.) before placing it on the page. You can place the element into the DOM with one of the following methods.

Append elements

- `parentNode.appendChild(childNode)` - appends *childNode* as the last child of *parentNode*.
- `parentNode.insertBefore(newNode, referenceNode)` - inserts *newNode* into *parentNode* before *referenceNode*.

Remove elements

- `parentNode.removeChild(child)` - removes *child* from *parentNode* on the DOM and returns a reference to *child*.

Altering elements

When you have a reference to an element, you can use that reference to alter the element's own properties. This allows you to do many useful alterations, like adding, removing, or altering attributes, changing classes, adding inline style information, and more.

```
// creates a new div referenced in the variable 'div'
const div = document.createElement("div");
```

Adding inline style

```
// adds the indicated style rule to the element in the div var
div.style.color = "blue";

// adds several style rules
div.style.cssText = "color: blue; background: white;";

// adds several style rules
div.setAttribute("style", "color: blue; background: white;");
```

When accessing a kebab-cased CSS property like `background-color` with JS, you will need to either use camelCase with dot notation or bracket notation. When using bracket notation, you can use either camelCase or kebab-case, but the property name must be a string.

```
// dot notation with kebab case: doesn't work as it attempts t
// equivalent to: div.style.background - color
div.style.background-color;

// dot notation with camelCase: works, accesses the div's back
div.style.backgroundColor;

// bracket notation with kebab-case: also works
div.style["background-color"];

// bracket notation with camelCase: also works
div.style["backgroundColor"];
```

Editing attributes

```
// if id exists, update it to 'theDiv', else create an id with
div.setAttribute("id", "theDiv");

// returns value of specified attribute, in this case "theDiv"
div.getAttribute("id");
```

```
// removes specified attribute
div.removeAttribute("id");
```

See MDN's section on HTML Attributes for more information on available attributes.

Working with classes

```
// adds class "new" to your new div
div.classList.add("new");

// removes "new" class from div
div.classList.remove("new");

// if div doesn't have class "active" then add it, or if it do
div.classList.toggle("active");
```

It is often standard (and cleaner) to toggle a CSS style rather than adding and removing inline CSS.

Adding text content

```
// creates a text node containing "Hello World!" and inserts i
div.textContent = "Hello World!";
```

Adding HTML content

```
// renders the HTML inside div
div.innerHTML = "<span>Hello World!</span>";
```

> Note that using textContent is preferred over innerHTML for adding text, as innerHTML should be used sparingly to avoid potential security risks. To understand the dangers of using innerHTML, watch this video about preventing the most common cross-site scripting attack.

Let’s take a minute to review what we’ve covered and give you a chance to practice this stuff before moving on. Check out this example of creating and appending a DOM element to a webpage.

```html
<!-- your HTML file: -->
<body>
  <h1>THE TITLE OF YOUR WEBPAGE</h1>
  <div id="container"></div>
</body>
```

```javascript
// your JavaScript file
const container = document.querySelector("#container");

const content = document.createElement("div");
content.classList.add("content");
content.textContent = "This is the glorious text-content!";

container.appendChild(content);
```

In the JavaScript file, first we get a reference to the `container` div that already exists in our HTML. Then we create a new div and store it in the variable `content`. We add a class and some text to the `content` div and finally append that div to `container`. After the JavaScript code is run, our DOM tree will look like this:

```html
<!-- The DOM -->
<body>
  <h1>THE TITLE OF YOUR WEBPAGE</h1>
  <div id="container">
    <div class="content">This is the glorious text-content!</d
  </div>
</body>
```

Keep in mind that the JavaScript does *not* alter your HTML, but the DOM - your HTML file will look the same, but the JavaScript changes what the browser renders.

Your JavaScript, for the most part, is run whenever the JS file is run or when the script tag is encountered in the HTML. If you are including your JavaScript at the top of your file, many of these DOM manipulation methods will not work because the JS code is being run *before* the nodes are created in the DOM. The simplest way to fix this is to include your JavaScript at the bottom of your HTML file so that it gets run after the DOM nodes are parsed and created.

Alternatively, you can link the JavaScript file in the `<head>` of your HTML document. Use the `<script>` tag with the `src` attribute containing the path to the JS file, and include the `defer` keyword to load the file *after* the HTML is parsed, as such:

```
<head>
  <script src="js-file.js" defer></script>
</head>
```

Find out more about the `defer` attribute for script tags.

Exercise

Copy the example above into files on your own computer. To make it work, you'll need to supply the rest of the HTML skeleton and either link your JavaScript file or put the JavaScript into a script tag on the page. Make sure everything is working before moving on!

Add the following elements to the container using ONLY JavaScript and the DOM methods shown above:

1. a `<p>` with red text that says "Hey I'm red!"
2. an `<h3>` with blue text that says "I'm a blue h3!"
3. a `<div>` with a black border and pink background color with the following elements inside of it:
   - another `<h1>` that says "I'm in a div"
   - a `<p>` that says "ME TOO!"

- Hint for this one: after creating the `<div>` with createElement, append the `<h1>` and `<p>` to it before adding it to the container.

Events

Now that we have a handle on manipulating the DOM with JavaScript, the next step is learning how to make that happen dynamically or on demand! Events are how you make that magic happen on your pages. Events are actions that occur on your webpage, such as mouse-clicks or key-presses. Using JavaScript, we can make our webpage listen to and react to these events.

There are three primary ways to go about this:

- You can specify function attributes directly on your HTML elements.
- You can set properties in the form of `on<eventType>`, such as `onclick` or `onmousedown`, on the DOM nodes in your JavaScript.
- You can attach event listeners to the DOM nodes in your JavaScript.

Event listeners are definitely the preferred method, but you will regularly see the others in use, so we're going to cover all three.

We're going to create three buttons that all alert "Hello World" when clicked. Try them all out using your own HTML file or using something like CodePen.

Method 1

```html
<button onclick="alert('Hello World')">Click Me</button>
```

This solution is less than ideal because we're cluttering our HTML with JavaScript. Also, we can only set one "onclick" property per DOM element, so we're unable to run multiple separate functions in response to a click event using this method.

Method 2

```html
<!-- the HTML file -->
<button id="btn">Click Me</button>
```

```
// the JavaScript file
const btn = document.querySelector("#btn");
btn.onclick = () => alert("Hello World");
```

> If you need to review the arrow syntax `() =>`, check this article about arrow functions.

This is a little better. We've moved the JS out of the HTML and into a JS file, but we still have the problem that a DOM element can only have one "onclick" property.

Method 3

```
<!-- the HTML file -->
<button id="btn">Click Me Too</button>
```

```
// the JavaScript file
const btn = document.querySelector("#btn");
btn.addEventListener("click", () => {
  alert("Hello World");
});
```

Now, we maintain separation of concerns, and we also allow multiple event listeners if the need arises. Method 3 is much more flexible and powerful, though it is a bit more complex to set up.

Note that all three of these methods can be used with named functions like so:

```
<!-- the HTML file -->
<!-- METHOD 1 -->
<button onclick="alertFunction()">CLICK ME BABY</button>
```

```javascript
// the JavaScript file
// METHOD 1
function alertFunction() {
  alert("YAY! YOU DID IT!");
}
```

```html
<!-- the HTML file -->
<!-- METHODS 2 & 3 -->
<button id="btn">CLICK ME BABY</button>
```

```javascript
// the JavaScript file
// METHODS 2 & 3
function alertFunction() {
  alert("YAY! YOU DID IT!");
}
const btn = document.querySelector("#btn");

// METHOD 2
btn.onclick = alertFunction;

// METHOD 3
btn.addEventListener("click", alertFunction);
```

Using named functions can clean up your code considerably, and is a *really* good idea if the function is something that you are going to want to do in multiple places.

With all three methods, we can access more information about the event by passing a parameter to the function that we are calling. Try this out on your own machine:

```javascript
btn.addEventListener("click", function (e) {
  console.log(e);
});
```

> When we pass in `alertFunction` or `function (e) {...}` as an argument to `addEventListener`, we call this a `callback`. A callback is simply a function that is passed into another function as an argument.

The `e` parameter in that callback function contains an object that references the **event** itself. Within that object you have access to many useful properties and methods (functions that live inside an object) such as which mouse button or key was pressed, or information about the event's **target** - the DOM node that was clicked. There's nothing magical about `e` as a name or where it comes from. JavaScript knows the parameter is an event because an event listener callback takes an `Event` object by definition. When the callback is run, the event handler passes in its own reference to the event. You can read more about the event objects on [MDN's introduction to events](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events).

Try this:

```
btn.addEventListener("click", function (e) {
  console.log(e.target);
});
```

and now this:

```
btn.addEventListener("click", function (e) {
  e.target.style.background = "blue";
});
```

Pretty cool, eh?

Attaching listeners to groups of nodes

This might seem like a lot of code if you're attaching lots of similar event listeners to many elements. There are a few ways to go about doing that more efficiently. We learned above that we can get a NodeList of all of the items matching a specific selector with `querySelectorAll('selector')`. In order to add a listener to each of them, we need to iterate through the whole list, like so:

```html
<div id="container">
  <button id="one">Click Me</button>
  <button id="two">Click Me</button>
  <button id="three">Click Me</button>
</div>
```

```javascript
// buttons is a node list. It looks and acts much like an arra
const buttons = document.querySelectorAll("button");

// we use the .forEach method to iterate through each button
buttons.forEach((button) => {
  // and for each one we add a 'click' listener
  button.addEventListener("click", () => {
    alert(button.id);
  });
});
```

This is just the tip of the iceberg when it comes to DOM manipulation and event handling, but it's enough to get you started with some exercises. In our examples so far, we have been using the 'click' event exclusively, but there are *many* more available to you.

Some useful events include:

- click
- dblclick
- keydown
- keyup

You can find a more complete list with explanations of each event on [W3Schools JavaScript Event Reference page].

Assignment

Manipulating web pages is the primary benefit of the JavaScript language! These techniques are things that you are likely to be messing with *every day* as a front-end developer, so let's practice!

1. Complete MDN's Active Learning sections on DOM manipulation to test your skills!

2. Read the following sections from JavaScript Tutorial's series on the DOM to get a broader idea of how events can be used in your pages. Note that some of the methods like `getElementById` are older and see less use today.

   As you read, remember that the general ideas can be applied to any event, not only the ones used in examples - but information specific to a certain event type can always be found by checking documentation.

   - JavaScript events
   - Page load events
   - Mouse events
   - Keyboard events
   - Event delegation
   - The dispatchEvent method
   - Custom events

Knowledge check

The following questions are an opportunity to reflect on key topics in this lesson. If you can't answer a question, click on it to review the material, but keep in mind you are not expected to memorize or master this knowledge.

- What is the DOM?
- How do you target the nodes you want to work with?
- How do you create an element in the DOM?
- How do you add an element to the DOM?

- How do you remove an element from the DOM?
- How can you alter an element in the DOM?
- When adding text to a DOM element, should you use textContent or innerHTML? Why?
- Where should you include your JavaScript tag in your HTML file when working with DOM nodes?
- How do “events” and “listeners” work?
- What are three ways to use events in your code?
- Why are event listeners the preferred way to handle events?
- What are the benefits of using named functions in your listeners?
- How do you attach listeners to groups of nodes?
- What is the difference between the return values of `querySelector` and `querySelectorAll`?
- What does a “NodeList” contain?
- Explain the difference between “capture” and “bubbling”.

Additional resources

This section contains helpful links to related content. It isn’t required, so consider it supplemental.

- Eloquent JS - DOM
- Eloquent JS - Handling Events
- Plain JavaScript is a reference of JavaScript code snippets and explanations involving the DOM, as well as other aspects of JS. If you’ve already learned jQuery, it will help you figure out how to do things without it.
- This W3Schools article offers easy-to-understand lessons on the DOM.
- JS DOM Crash Course is an extensive and well explained 4 part video series on the DOM by Traversy Media.
- Understanding The Dom is an aptly named article-based tutorial series by DigitalOcean.
- Introduction to events by MDN covers the same topics you learned in this lesson on events.

- Wes Bos's Drumkit JavaScript30 program reinforces the content covered in the assignment. In the video you will notice that a deprecated keycode keyboard event is used, replace it with the recommended code keyboard event and replace the `data-key` tags accordingly.
- Captura de eventos, propagación y video de burbujeo del programa JavaScript30 de Wes Bos.
- Comprender las devoluciones de llamadas en JavaScript para una comprensión más profunda de las devoluciones de llamadas.

Educación en codificación de alta calidad mantenida por una comunidad de código abierto.

Sobre nosotros

Acerca de

Equipo

Blog

Casos de éxito

Apoyo

Preguntas frecuentes

Contribuir

Contáctenos

Guías

Guías de la comunidad

Guías de instalación

Legal

Términos

Privacidad