# DOM manipulation

So far, we have covered vanilla JS. Now we cover Browser API to interact with the browser using code (in this case, the HTML and CSS). We call the Browser API the “DOM,” or Document Object Model. Being able to manipulate the browser view will provide you to do:

- Build a modal that pops up over your content.
- Wipe parts of the page and add your own content.
- Update content on the page with fresh or new data.
- Track form usage and provide user feedback.
- Trigger CSS animations.
- Build charts, graphs and reports.
- etc.

Working with the DOM requires:

1. How to Get Access to HTML with Your JavaScript
2. How to Manipulate HTML and CSS with Your JavaScript
3. How to Create and Add Your Own HTML or CSS to a Page
4. How to Listen for User Interactions on the Page

Let’s use the following HTML Document:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript and the Browser API</title>
    <meta charset="UTF-8" />
  </head>
  <body>
    <div id="content">
      <h1>Header 1</h1>
      <p>Some lorem to <strong>fill</strong></p>
      <div class="featured">
        <p><a href="https://google.com">Visit Us!</a></p>
      </div>
      <p>Closing thoughts</p>
    </div>
    <script src="js/index.js"></script>
  </body>
</html>
```

We have a number of ways to select HTML with the Browser API such as getElementbyID(), but the two simplest are
querySelector() and querySelectorAll() that provides you the ability to work with the browser view based on CSS selectors.

in order to use any DOM or Browser API functions, we have to do document.functionName(). Or, in this

case:

document.querySelector()

document.querySelectorAll().

An example:

```
// Gets the div with id of "content"
const content = document.querySelector(`#content`);
// Gets <h1>Header 1</h1>
const header = document.querySelector(`h1`);
// Gets <strong>fill</strong>
const bold = document.querySelector(`p strong`);
// Gets the div with the class of "featured"
const featured = document.querySelector(`.featured`);
// Gets all the <p> tags and what is inside them
const paragraphs = document.querySelectorAll(`p`);
// Gets all the <div> tags and what is inside them
const divs = document.querySelectorAll(`div`);
```

remember.
1. querySelector() will only return the first matching element and ignore and
2. querySelectorAll() always returns a collection of none, one, or many items

When we use ***querySelectorAll()*** to get a collection of DOM elements, you may think that what we get is a JavaScript array. However, we technically get a NodeList, which cannot do the same things as arrays.
So, sometimes you will need to convert a collection of DOM elements into an array like so if you want to map over it or use other array functionality:

***const headers = Array.from( document.querySelectorAll(`p`) )***
***headers.map( header => console.log( header ) )***

Before we wrap up here though, we need to look at how to dig *inside* the HTML we get.
Here we have a heading with a link inside of it.

```
<article id="post">
  <h2 class="post-title">
    <a href="/url/here">Heading</a>
  </h2>
</article>
```

And here we have selected the heading with the Browser API and JavaScript. Then we go on to select the HTML inside of the heading with .**innerHTML** and the text inside the header with .**innerText**.

```
// Gets <h2 class="post-title">
// <a href="/url/here">Heading</a></h2>
const title = document.querySelector( `.post-title` )

// Logs "<a href="/url/here">Heading</a>"
console.log( title.innerHTML )

// Logs "Heading"
console.log( title.innerText )
```

Try this and check the browser

```
title.innetText = “something else”;
```

We can also get the attributes of any HTML elements.
For instance, here we get the id, class, and href of elements:

```
// Gets <article id="post"></article> and contents
const article = document.querySelector( `article` )
// Logs the article id of "post"
console.log( article.id )
// Get the class attribute
const h2 = document.querySelector( `h2` )
// Logs "post-title"
// Note: Use "className" not "class"!!!
console.log( h2.className )
```

## HOW TO MANIPULATE HTML AND CSS WITH YOUR JAVASCRIPT

For the HTML code:

```
<h2 class="post-title">
  <a href="/url/here">Heading</a>
</h2>
```

If we save elements to variables we can simply override the values of the HTML text, HTML, attributes, and just about everything else.

```
// Get the link
const link = document.querySelector( `.post-title a` )
// Replace the HTML inside the link with bold markup
link.innerHTML = `<strong>Heading</strong>`
// Get the h2 header
const h2 = document.querySelector( `h2` )
// Override the class and add a new one
// Note: Use `className` and not `class`
const h2.className = `post-title hidden`
// Override text inside element or children
const h2.innerText = `NEW HEADING`
```

These codes are in index1. Try index2 and index3.html too.

## ADDING CSS CLASSES

Using **ClassList,** CSS classes can be added or removed. See the code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <style>
        .main { color: red; }
    </style>
</head>
<body>
    <h1 id="item">Black-Red Text</h1>
</body>
</html>
```

```
// Gets the <h2>
const header = document.querySelector( `h2` )
// Results in <h2 class="entry-header"></h2>
header.classList.add(`entry-header`)

// Removes the `entry-header` class added
// in previous example
header.classList.remove(`entry-header`)
```

Also you can toggle a class on or off. If the class is currently on the element, **toggle** will remove it. If the class is not currently on an element, **toggle** will add it.

```
// Gets the <h1>
const header = document.querySelector( `h1`
)
// Results in <h2 class="entry-
header"></h2>
header.className +='main'
// Removes the `entry-header` class added
// in previous example

// header.classList.remove (`main`)
```

For hiding an element You can manually override the class attribute like **section.class = "hidden"**, but using **classList** is the recommended technique to add, remove, or toggle classes. To completely replace the class string, you would use **className**.

a great Browser API function called insertAdjacentHTML() that let us select an existing element on the page and add our element before it, inside it, or after it.

Here are the four insertAdjacentHTML( *option* ) options:

- beforebegin – Insert your HTML before the selected element
- afterbegin – Insert your HTML inside the selected element *before anything else*
- beforeend – Insert your HTML inside the selected element *at the very end*
- afterend – Insert your HTML after the selected element

In most cases we will add our HTML using beforeend as in the example below where we take our article and add it to HTML already on the page.

Starting HTML: 

Then
1) create our HTML,
2) select the div above and
3) add our markup.
Starting HTML:

```
<!DOCTYPE html>
<html lang="en">
<body>
    <div class='app'></div>
</body>
<script src="js/index6.js"></script>
</html>
```

```
// Create a function for our title UI
function title( text ) {
    return `<h1>${text}</h1>`
}
// Select our app div
const container = document.querySelector( `.app` )
// Add title inside app
container.insertAdjacentHTML( `beforeend`, title(`Hello!`) )
```

This would give us the resulting markup on the page:

```
<div class="app">
  <h1>Hello!</h1>
</div>
```

Another example:

Start from

```
<div class="app"></div>
```

This time, using arrow function

```
// Create functions for UI
const header = siteName => `<header><h1>${siteName}</h1></header>`
const content = page => (`<section class="content">
<h2>${page.title}</h2>
<div>${page.content}</div>
</section>`)
const sidebar = widgets => `<aside>${widgets}</aside>`
const footer = siteName => `<footer><p>Copyright ${siteName}</p></footer>`
```

Finally, we can setup some data for the site and add the markup inside our starting ** **.

```
// Get the container div
const container = document.querySelector( `.app` )
// Setup site data
const siteName = `My Site`
const page = { title: `Hello`, content: `<p>Lorem</p>` }
const widget = `<h3><a href="/">About Me</a></h3>`
// Add data to the page
container.insertAdjacentHTML( `beforeend`, header(siteName) )
container.insertAdjacentHTML( `beforeend`, content(page) )
container.insertAdjacentHTML( `beforeend`, sidebar(widget) )
container.insertAdjacentHTML( `beforeend`, footer(siteName) )
```

This would result in the final HTML on the page:

```
<div class="app">
  <header>
    <h1>My Site</h1>
  </header>
  <section class="page">
    <h2>Hello</h2>
    <div><p>Lorem</p></div>
  </section>
  <aside>
    <h3><a href="/">About Me</a></h3>
  </aside>
  <footer>
    <p>Copyright My Site</p>
  </footer>
</div>
```

**Working with forms**

Find the following in (index3-form.html)

Forms are different. We need to get the **value** instead of **innertext**.

```
<form action="">
  <label for="username">Username</label>
  <input type="text" id="username" name="username" value="myname" />
  <input type="submit" />
</form>
```

```
// Gets <input type="text" id="username"
// name="username" value="myName" />
const username = document.querySelector( `#username` )
// Logs "myName"
console.log( username.value )
```

try

```
document.querySelector('form').remove()
```

Same goes for forms, although we use value again for **form** elements like input with value attributes.

```
// Gets <input type="text" id="username"
// name="username" value="myname" />
const username = document.querySelector( `#username` )
// Replaces username <input> value with "newuser"
username.value = "newuser"
```

## HOW TO LISTEN FOR USER INTERACTIONS ON THE PAGE

We can setup what are called “Event Listeners” on any element on the page (or even the page itself) and listen for dozens of different types of events.
Here are just a few of the events we can listen for

- Scrolls
- Hovers
- Mouse movements
- Window (re)size
- URL changes
- Page and item loading
- Anything clicked on
- Touch and drag events
- Key presses
- Form interactions
- Media interactions
- etc

### Adding Event Listeners

**For:**

```
<nav class="main" >
  <ul>
  <li><a href="/">Home</a></li>
  <li><a href="/services">Services</a></li>
  <li><a href="/about">About</a></li>
  <li><a href="/contact">Contact</a></li>
  </ul>
  </nav>
```

Let's listen for anytime someone clicks on the links in the main navigation above and then call a function named sayHi that will alert a hello!

```
// Get a DOM collection and convert to
// a JS array for mapping
const links = Array.from( document.querySelectorAll( 'nav.main a' ))
// Add an event listener to each link
// To call sayHi() when clicked
links.map( link => link.addEventListener( 'click', sayHi ))
// Function called when link is clicked
function sayHi() {     alert( 'Hi!' ); }
```

Many times in JavaScript we want to prevent the default behavior of events so we can create our own custom interactions. For example, let's say in the example above we did not want the user to go to that page, but we wanted to fetch that data ourselves and display it on the page without having to reload the entire page.

### Preventing Default Event Behavior (And Adding Our Own)

To prevent the default behavior we can do the following with our event functions:

```
function sayHi( e ) {
    e.preventDefault()
    alert( `Hi!` )
    }
```

Notice how we simply added an e. We could also use event, or call it whatever we want, but this gives us a function called preventDefault() which stops whatever default behavior was about to occur.

In this case, e.preventDefault() stops the user from going to the link they clicked on. In a form it could prevent the form from submitting. We are then able to create our own custom interaction.

### The Event Object

When a Browser Event occurs and we have hooked into it with an addEventListener() in our JavaScript, we also get access to the event object

```
// Adding an event listener to a link
document.querySelector(`a.hacked`).addEventLister( `click`, sayHi )
// The Browser API will give us the Event Object
function sayHi( e ) {
// Logs out the Event Object
console.log( e )
// This is using the Event Object
e.preventDefault()
}
```

Try and see the results

### Getting the Event Element In Your Event Listener Function

When you have a function hooked up to an event listener, you will often want to get that element and information about it.
setup an event listener on the form submit:

```
// Adding an event listener to a link
document.querySelector(`a`).addEventListener( `click`, sayHi2 )
// The Browser API will give us the Event Object
function sayHi2( e ) {
    e.preventDefault()

// Logs out the Event Object
console.log( e );
console.log(this)
console.log(this.innerText);
// This is using the Event Object
}
```

## EXAMPLES OF DIFFERENT TYPES OF EVENTS

There are many different types of events that you can attach event listeners to in JavaScript.
In general they break down into a few different categories:

1. Mouse events
2. Keyboard events
3. Form events
4. Media events
5. Drag and Drop events
6. Window events

There are many more types of events, but these are the primary ones

### Mouse Event Example

This function and event listener below will log out the X and Y position of the mouse wherever it moves on the screen:

```
document.addEventListener( 'mousemove', logMousePosition);
function logMousePosition(e) {
console.log( e.clientX + ' x ' + e.clientY );
}
```

The event listener is attached directly to the document object and not to a specific element on the page. This ensures the event will execute continuously.

### Keyboard Events Example

In this example below we add an event listener to the **keypress** event on the document object. Then we can log out all the keys that are pressed.

```
document.addEventListener( 'keypress', logKeys );
function logKeys(e) {
            key = e.which;
    console.log(key);
    console.log(String.fromCharCode(key));
}
```

Notice how if we log the key directly we get a number. However, if we use **fromCharCode** then we get the key we would expect. This is because by default, the Browser API captures each character press as a unique number.

**Form Events Example**

In this example below we get the name, email and message from a contact form and log it to console.
Here is an HTML contact form for reference:

```
<form id="contact">
  <p>
    <label for="name">Name:</label>
    <input type="text" id="name" name="myname" />
  </p>
  <p>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" />
  </p>
  <p>
    <label for="message">Message:</label>
  </p>
  <textarea id="message" name="message"></textarea>
  <p>
    <input type="submit" value="Send" />
  </p>
</form>
```

In the following code, we attach an event listener to the form submission and prevent it from being submitted, which would reload the page. Then, we get each of the form field values and log them to the console.

```
const form = document.querySelector(`form#contact`);
form.addEventListener(`submit`, logFormDetails);
function logFormDetails(e) {
    e.preventDefault();
    let name = document.querySelector(`#name`);
    let email = document.querySelector(`#email`);
    let message = document.querySelector(`#message`);

    console.log(`Form Details:`);
    console.log(`Name: ${name.value}`);
    console.log(`Email: ${email.value}`);
    console.log(`Message: ${message.value}`);
}
```

### Media Event Example

In this example we show how you can add control buttons to a video. This would also work with an audio file.

This HTML includes an embedded video file and some buttons for Play, Pause and Skip to Beginning and Skip Forward 10 Seconds. We also have a span around a timestamp that we can update with the current timestamp of the video.

```
    <video controls>
      <source src="video.mp4" type="video/mp4" />
    </video>
    <p>
      <input id="play" type="submit" value="Play" />
      <input id="pause" type="submit" value="Pause" />
      <input id="restart" type="submit" value="Go to the Beginning" />
      <input id="skipForward" type="submit" value="Go Forward 10 Seconds" />
      Timestamp: <span id="timestamp">0:00</span>
    </p>
```

In this JavaScript we select the video and each of the buttons we want to update and attach them to event listeners:

```
const video = document.querySelector("video");
const playBtn = document.getElementById("play");
const pauseBtn = document.getElementById("pause");
const restartBtn = document.getElementById("restart");
const forwardBtn = document.getElementById("skipForward");
let timestamp = document.getElementById("timestamp");
video.addEventListener(`timeupdate`, updateTimestamp);
playBtn.addEventListener(`click`, playVideo);
pauseBtn.addEventListener(`click`, pauseVideo);
restartBtn.addEventListener(`click`, restartVideo);
forwardBtn.addEventListener(`click`, skipForward);
    function playVideo() {
    video.play();
    }
    function pauseVideo() {
    video.pause();
    }
    function restartVideo() {
    video.currentTime = 0;
    }
    function skipForward() {
    video.currentTime = video.currentTime + 10;
    }
    function updateTimestamp() {
    timestamp.innerHTML = parseInt(video.currentTime);
    }
```

Notice that once we select the video element from the page it comes with the methods play() and pause() that let us start and stop the video.

We also get a currentTime property that we can both read from to get the timestamp and write to to change the current location in the video.

### Window Events Example

In this example, we will look at how to use the window.scrollBy method to create a scroll event.

We start with a button and an element 1000px below it to give us room to scroll down.

```
<header>
    <button id="scroll">Scroll to Footer</button>
</header>
<footer style="margin-top: 1000px">Footer</footer>
```

Next we can attach an event listener to it, find the y position of footer, and then call window.scrollBy to move the window down to the footer.

```
const button = document.querySelector(`#scroll`);
const footer = document.querySelector(`footer`);
button.addEventListener(`click`, scrollToFooter);
function scrollToFooter(e) {
    const footerY = footer.getBoundingClientRect().top;
    e.preventDefault();
    window.scrollBy({
        left: 0,
        top: footerY,
        behavior: `smooth`
    });
}
```

Some of the new code we see is the getBoundingClientRect() method that gives us the top, left, right, and bottom position of where an element appears in the window.

We also see the window.scrollBy function that can scroll to a specific x and y position. We also add smooth to the behavior to get a smooth scroll (in supported browsers).