Forms

Basic Form Elements

- `<form></form>` element:
  - The parent element of our forms
- `<input />` element:
  - Single lines of text
  - Radio Buttons
  - Checkboxes
- `<select></select>` & `<option></option>` elements:
  - These make dropdowns
  - Select is the parent element and each option should be contained in option tags
- `<textarea></textarea>` element:
  - Blocks of multi-line text

The <input> Element

- The `type` of an input is an attribute of the tag that tells the browser what kind of input to render
- Another common attribute is the `placeholder` attribute which can be used to add placeholder text in empty input fields

`input type="text"`

```
<form>
  First Name: <input type="text" placeholder="Enter your first name here..."/>
</form>
```

`input type="radio"`

- Radio buttons can have only one value selected
- The `value` is another attribute that we can use, it tells the input what its value is (this can be read or changed in JavaScript)

```
<form>
  <input type="radio" name="time-group" value="am" id="am"/>
  <label for="am">AM</label>
  <input type="radio" name="time-group" value="pm" id="pm"/>
  <label for="pm">PM</label>
</form>
```

`input type="checkbox"`

Checkboxes can have more than one value selected

```
<form>
  <label for="dog">Dog:</label><input type="checkbox" id="dog" value="dog"/>
  <label for="cat">Cat:</label><input type="checkbox" id="cat" value="cat"/>
  <label for="hedgehog">Hedgehog:</label><input type="checkbox" id="hedgehog"
value="hedgehog"/>
  <label for="yak">Yak:</label><input type="checkbox" id="yak" value="yak"/>
</form>
```

Dropdown menus with `select`

```
<form>
  Melon Type:
    <select>
      <option value="watermelon">Watermelon</option>
      <option value="honeydew">Honeydew</option>
      <option value="cantaloupe">Cantaloupe</option>
    </select>
</form>
```

Textareas with `textarea`

```
<form>
  <p>Tell us about yourself:</p>
  <div>
    <textarea></textarea>
  </div>
</form>
```

JavaScript in Browsers

Browsers have their own JavaScript runtime engines

- Google Chrome uses the same engine as Node
- Safari, Firefox, etc. use their own engines

There can be minor differences between implementations

Note: Feature Compatibility Table

If you’re curious about the differences between platforms, check out this JavaScript compatibility table.

You can interact with your browser’s JS runtime through the ***Web Console***

Open your browser’s developer tools → click on ***Console*** tab

- It’s a REPL, so you execute JS expressions and see what they do
- This is also where any error messages, warnings, or console logs will appear

Loading JavaScript

There are two ways to include JavaScript in a web page

You can write *inline* JavaScript with the ***script*** tag

```
<body>
  <h1>JavaScript is fun!</h1>

  <script>
    console.log('Hello, world!');
  </script>
</body>
```

Even better, you can import code from a JavaScript file with the ***script*** tag

```
<body>
  <h1>JavaScript is neat!</h1>

  <script src="./helloWorld.js"></script>
</body>
```

When your browser encounters a ***script*** tag, it loads and executes that code

- All code is executed in the same namespace, as if you've copied and pasted *all* your code into one document
- Be careful if you've defined many globally-scoped variables or functions
  - Further study: look into loading JS files as modules and how to use the `import` / `export` statements

JavaScript Web API

A collection of classes, methods, functions, etc. that are built into your browser

They expose data from the browser so you can do useful, complex things with it

Note: What else is in the Web API?

Some examples from MDN: Web API Documentation:

- WebGL is used to render 2d and 3d images in the browser
- The Notifications API is used to send notifications to your user
- Ambient Light Events can help you execute code based on changes to ambient light

For example, these functions are used to open dialog windows in your browser

```
alert('Message to pop up');

confirm('Hey, user, is this ok?');
// => true or false

prompt('Enter a string');
// => the string that was entered
```

We'll focus on the most commonly-used Web API — the ***document*** API

It's used to make change the contents of a web page by giving us the ability to manipulate the DOM

What's the DOM?

DOM stands for Document Object Model

A tree that stores HTML elements as objects

```
<!doctype html>
<html>
  <head>
    <title>The Title</title>
  </head>
  <body>
    <h1>First Heading</h1>
    <h2 class="urgent">Second Heading</h2>
  </body>
</html>
```

Getting Elements from the DOM

`document.querySelector('selector')`

- Returns *only the first* ***HTML Element*** that matches the selector
- Selectors are always contained in quotes
- Like CSS, use plain tag names, period before a class, and pound before an id

Reference: MDN Web Docs: Document.querySelector()

```html
<p class="lead">
  DOM manipulation is fun.
</p>
<p id="main-content">
  Here's how you do it.
</p>
```

```javascript
document.querySelector('p');
// => <p class="lead">

document.querySelector('#main-content');
// => <p id="main-content">
```

`document.querySelectorAll('selector')`

- Return a *collection* of ***HTML Elements*** that match the selector
- The collection that's returned is called a ***NodeList*** and behaves similarly to an array

Reference: MDN Web Docs: Document.querySelectorAll())

```html
<p class="lead">
  DOM manipulation is fun.
</p>
<p id="main-content">
  Here's how you do it.
</p>
```

```javascript
document.querySelectorAll('p');
// => NodeList [ <p class="lead">, <p
id="main-content"> ]
```

Manipulating DOM Elements

HTMLElement objects have methods and properties for you to get/set data about them

Reference: MDN Web Docs: HTMLElement

Note: How to use the docs

Check out MDN's documentation for the ***HTMLElement*** object and you'll notice that there are more than a few methods missing from the ***Methods*** list. They're not actually missing from the docs!

HTMLElement inherits methods and properties from its parent class, ***Element***. If you want a complete list of methods and properties available on ***HTMLElement*** objects, you should also check out MDN Web Docs: Element.

You can get/set the text content inside an element

```
<a href="/about">About</a>
```

```
const aboutLink = document.querySelector('a');

console.log(aboutLink.textContent);  // => "About"

aboutLink.textContent = 'About Me';
```

Result of setting textContent

```
<a href="/about">About Me</a>
```

You can get/set attributes

```
<img src="cat.jpg">
```

```
const catPhoto = document.querySelector('img');

console.log(catPhoto.getAttribute('src'));  // => 'cat.jpg'

catPhoto.setAttribute('src', 'cat2.png')
```

Result of setAttribute

```
<img src="cat2.png">
```

Get a list of the element's classes

```
<div class="container blog-content"></div>
```

```
const blogContainer = document.querySelector('div');

console.log(blogContainer.classList);
// => DOMTokenList [ "container", "blog-content" ]
```

Add/remove a class

```
blogContainer.classList.remove('blog-content');

blogContainer.classList.add('article');
```

Result

```
<div class="container article"></div>
```

Check out the documentation for more things you can change!

Creating and Removing DOM Elements

HTML Elements and the ***document*** object have built-in methods that we can use to create and remove HTML using JavaScript

Creating and Appending HTML

We can insert HTML into the DOM using ***document.createElement*** and ***element.appendChild***

```
<article>
  <h1>Title</h1>
</article>
```

```
const article = document.querySelector('article')

const newPara = document.createElement('p')

newPara.textContent = 'This is a new paragraph.'

article.appendChild(newPara)
```

Removing HTML

HTML Elements all have the ability to self-destruct using the ***remove*** method

```
<div class='remove'>Remove me!</div>
```

```
const myDiv = document.querySelector('.remove')

myDiv.remove()
```

Event Driven Programming

What is Event Driven Programming?

A way to design programs where code execution is triggered by events like

- User actions, such as clicks and keypresses
- Output from other programs
- Other sensors, such as temperature, movement, pressure
- Custom events that you define

A Basic Event System

Event-driven systems are used in many languages and applications

They all have the same basic parts

event source/event target
The button, sensor, etc. where the event is coming from

event emitter/event dispatcher
Notify the rest of the system that an event has occurred

event listeners and event handlers
Work together to listen for events and handle them by executing code

event loop
In charge of making all of the above work together

Event Handling in the Browser

In the browser, DOM elements are (one of many) sources of events

Common Event Types

MDN Web Docs: Event Reference

- Mouse events
  - `click` — when you click on an element
  - `dblclick` — when you double-click on an element
  - `mouseover` — when the mouse is on top of an element
- Input events
  - `change` — when the value of an ***input*** element changes
- Form events
  - `submit` — when a form is submitted

Adding Event Listeners

`HTMLElement.addEventListener(eventType, callback)`

Add an event listener to ***HTMLElement***

- ***eventType*** — the type of event to listen for
- ***callback*** — the function to call when that event happens

```
const button = document.querySelector('#angry-button');

button.addEventListener('click', () => {
  alert('Stop clicking me!');
});
```

Callback Functions

Callback functions are functions that we don't call right away

Instead, they're passed in as arguments so the program can call them later

Let's break down the syntax in ***button.addEventListener***

```
button.addEventListener(
  'click',

  () => {
    alert('Stop clicking me!');
  }
);
```

- `'click'` is the event type
- 2nd argument is the callback function

Another way to write the above

```
const showAlert = () => {
  alert('Stop clicking me!');
}

button.addEventListener('click', showAlert);
```

Note that we're passing in the function *object*

This won't work:

```
button.addEventListener(
  'click',
  showAlert()  // Don't do this!
);
```

Note: Why the code above doesn't work

It all has to do with objects and their data types. ***HTMLElement.addEventListener*** expects you to pass in a function object — it won't work if you pass in a value with a different data type.

For example, here's a function definition:

```
const getGreeting = () => {
  return 'hello';
}
```

When the definition is executed, JavaScript creates an identifier called ***getGreeting*** and assigns it to a function object. When we access ***getGreeting***, a function object is returned:

```
console.log(getGreeting);
// => function getGreeting()

console.log(typeof getGreeting);
// => "function"
```

This is different from accessing the result of *calling* ***getGreeting***. The result of calling ***getGreeting*** isn't a function object — it's a string:

```
getGreeting();
// => "hello"

console.log(typeof getGreeting());
// => "string"
```

Who Calls the Callback Function

We don't call callback functions — JavaScript does that for us!

- JavaScript will listen for the event
- If the event occurs, JavaScript handles the event for us by
  - Calling the appropriate function
  - Passing in arguments to that function

The ***Event*** Object

When JavaScript calls a function to handle an event, it passes in one argument — the ***Event*** object

The ***Event*** object contains data about the event that has occurred

Event objects have methods and properties that let us do things like

- Cancel the event
- Get the event target (in this case, the ***HTMLElement*** where the event came from)
- Get the position of the mouse when the event occurred

To access the ***Event*** object, our callback needs a parameter:

```
const eventBtn = document.querySelector('#event-btn');

eventBtn.addEventListener('click', (evt) => {
  console.log(evt);
});
```

Cancelling Events

`Event.preventDefault()`

Cancel an event

```
<a href="/secrets">Don't go to this page</a>
```

```
const secretLink = document.querySelector('a');

secretLink.addEventListener('click' (evt) => {
  evt.preventDefault();

  alert('I told you not to go to that page.');
});
```

This is useful when you want to validate data in a form before submitting it

```
<h4>Suggest a word!</h4>
<p>Your word must be > 5 characters long</p>

<form action="/suggest-word" method="POST">
  <input type="text" name="word">
  <input type="submit">
</form>
```

```
const wordForm = document.querySelector('form');

wordForm.addEventListener('submit', (evt) => {
  const wordInput = document.querySelector('input[name="word"]');

  if (wordInput.value.length < 5) {
    evt.preventDefault();
  }
});
```

Event Target

`Event.target`

Return the source of the event as an ***HTMLElement***

```
<button id="event-target">See event target (in console)</button>
```

```
const targetBtn = document.querySelector('#event-target');

targetBtn.addEventListener('click', (evt) => {
  console.log(evt.target);
});
```

Event.target is an ***HTMLElement***!

- It has all properties and methods of ***HTML Elements***
- This is great when you need to extract info from the element that caused the event

For example, we have a bunch of buttons.

We want to use each button to change the text inside of a section on our page.

```
<button class="text-changer" id="apples">
  Change to "apples"!
</button>
<button class="text-changer" id="bananas">
  Change to "bananas"!
</button>
<button class="text-changer" id="strawberries">
  Change to "strawberries"!
</button>
```

We *could* do it this way

```
const section = document.querySelector('section')

const appleBtn = document.querySelector('#apples')
appleBtn.addEventListener('click', () => {
  section.textContent = "apples";
});

const bananaBtn = document.querySelector('#bananas')
bananaBtn.addEventListener('click', () => {
  section.textContent = "bananas;
});

const strawberryBtn = document.querySelector('#strawberries')
strawberryBtn.addEventListener('click', () => {
  section.textContent = "strawberries";
});
```

If we take advantage of ***Event.target***, we can avoid repetition

```
const section = document.querySelector('section')

const btns = document.querySelectorAll('.text-changer')

for (let i = 0; i < btns.length; i++) {
  btns[i].addEventListener('click', e => {
    section.textContent = e.target.id
  })
}
```

Summary

- We can use form elements to aid in user interaction
- JavaScript APIs help us run JavaScript in the browser
- The DOM is an object representation of our HTML
- Use ***querySelector*** and ***querySelectorAll*** to target elements
- We can use JavaScript to manipulate ***HTML Elements***
- The ***Event*** object gives us access to data about the events that occur on our pages

The End