**JavaScript Essential Training:**

**Instructor – Morten Rand-Hendriksen (LinkedIn Learning)**

**\*Tools for working with JavaScript:**

* **Modern browser:** ideally all browsers for testing purposes.
* **Code editor:** the industry standard is VS Code currently
* **Live server environment –** extension on VS Code or similar.
* **The browser console –** all modern browsers have one.

**\*Linting and formatting:**

* **ESLint –** a spell check but for JavaScript. It helps automatically detect coding errors and can-do basic cleanup automatically.
* **Prettier code formatter –** helps automatically clean up formatting.
* **Important note –** both tools require Node.js.

**\*JavaScript in an HTML document:**

Where does the JavaScript live/is the code written?

* Inline (in the HTML file)
* External file reference.

<script> tags render the JavaScript in HTML files.

* Placement of this tag matters in the HTML.
* Technically speaking, we want to place the script tag in the <head> section of the HTML as it’s not part of the actual document.
* The placement of the <script> tag has to do with how to browser renders JavaScript.

When the browser is given an HTML document, it’s read line by line.

* If the browser encounters JavaScript, everything stops.
* This is because the JS may make changes to the document.

For example, the following <script> tag looks for any instance of the code element.

Graphical user interface, text, application, email

Description automatically generated

If we want the browser to find the “code” element, it will first need to render the “code” element which is where the HTML comes into play and why the placement of the <script> tag needs to be at the end of the document in this case.

* The example above is a more traditional way of dealing with JS. Today, there are plenty of tools that help us control what happens.

When using a lot of JS code, it makes more sense to simply add the <script> tag and src=”file.js” to avoid taking up most of the HTML document with JS code.

**\*Modern JavaScript loading:**

Moving a <script> tag to the end of an HTML document is not always the correct fix to render issues.

* Remember that the browser renders line by line until it hits CSS style sheets or JavaScript code then continues rendering.

If we move the <script> tag to the end of the HTML file, it will introduce many other issues because of the placement. JavaScript placement matters!

* In today’s world, we now have new tools in JavaScript that tightly control how and when JavaScript is loaded.

**Tools: “async”** and **“defer”** keywords:

As discussed, the **default browser behavior** **stops rendering** when **JavaScript** is encountered. **JavaScript** is **executed before rendering continues** which is often referred to **as “content blocking”**

Here is a visual example of how this occurs:

A screenshot of a computer

Description automatically generated with low confidence

The **async** keyword changes this behavior significantly.

* Tells the browser to keep parsing the HTML while the JavaScript file is being downloaded. Only stop the parsing when you have the file.
* Once the file is downloaded, execute it, and continue HTML parsing.
* This dramatically shortens the time it takes for the browser to execute everything, and it won’t create a huge render blocking issue.
* This is good when the JS needs to be downloaded quickly without the worry of render blocking.

**Another scenario:**

**What to do if we need to ensure that the browser only executes the JavaScript after the document is complete?**

* For this situation, we have the **“defer”** keyword.

This keyword tells the browser to parse the HTML as usual. If JavaScript is encountered along the way, load it alongside the HTML parsing.

* When the HTML parsing is complete, execute whatever JavaScript you have.
* The browser loads the JavaScript asynchronously when it is encountered, then waits until all HTML is rendered before executing the script.

**Key takeaways:**

* **async/defer should be the standard way of loading JavaScript today.**
* **Only use render blocking when you have a specific reason (placing script tags before or at the end of the HTML file).**

From here on, **JavaScript should always be loaded in the head. Once loaded, use “async” or “defer” to control when that JavaScript is executed on the document.**

**\*JavaScript Modules:**

* Allow us to break pieces out of a JS file and place them in a separate file. Once placed in the new file, the pieces are imported into the original file again.

**Look at the import example code from repo (02\_04 script.js)**

**In backpack.js,** we have defineda const called backpack and exported it as seen below:



This line above tells the browser that the entity called backpack (in this case an object) can be used by any other file if it’s imported.

* The **script.js file shows the import**:



There is one more thing that needs to be done still to ensure the code is working as expected:

* Inside **index.html**, we need to tell the browser that there are **two files** now and that the **two files are modules**.

Text

Description automatically generated

**Modules** may rely on one another so it’s important that they are both loaded before things work properly.

* **Type module** will run to make sure the program has everything it needs before being rendered out.
* **Advantage of using modules:**

Modules enable modularization of code where individual functions, components, data objects, and other parts can be separated into individual files.

**\*Objects: A practical introduction**

With JavaScript, we are working with objects that are based on prototypes which means that each object is a unique instance of an object prototype.

* JavaScript works like how we humans work with objects in the real world.
* An object in JavaScript is pretty much the same as an object in real life only written in code.

Going back to the backpack example in the code. Take a real backpack in real life. There are properties such as size, color, number of pockets, straps, etc.

* Object properties define the specifics of this one object (backpack).

With this in mind, we can have several backpacks that each have unique properties associated.

Objects have features built in that allow us to change their property values.

* In **JavaScript**, these **property changing features** inside of an **object** are called **Methods.**
* **Objects can contain other objects.**
* **Changing one object DOES NOT change the other.**

**For example, my backpack can contain pens, pencils, a laptop, etc.**

**\*JavaScript objects: The code version**

* To define an object/create it, I first need a variable to hold the object (const backpack).
* Once the constant is created, we use an equal’s sign to assign a value to the constant and curly braces (const backback = {}).
* Curly braces define data as an object.

**Once created, I need to populate my object with some data:**

* **Data** is **populated** using **properties**. For example:

const backpack = {

name: “Everyday Backpack”,

};

**Objects** can have **as many properties as you like** **separated by commas**. They can also **nest sub-objects with their own properties.**

For example:

const backpack = {

name: “Everyday Backpack”,

type: “leather”,

strapLength: {

left: 26,

right: 26,

};

};

**Objects** can also have **methods** to **change the properties of the object:**

lidOpen: false,

toggleLid: function (lidStatus){

this.lidOpen = lidStatus;

};

};

Remember that the **“this” keyword** refers to the **current object (this object right here).**

**Here is another quick method syntax example:**

newStrapLength: function(lengthLeft, lengthRight){

this.strapLength.left = lengthLeft;

this.strapLength.right = lengthRight;

};

**\*Object containers**

Const backpack = {

name: “Everyday Backpack”,

volume: 30,

color: “grey”,

};

The example above is considered a **container**.

* **Objects are typically Constants.**

We can change the properties of the object inside the container. We can’t remove or replace the object from the container.

For example, setting backpack = 50 will not work as it is already defined as a constant with other values. However, changing properties inside the container such as updating the value of volume from 30 to 50 will work **(backpack.volume = 50 in the console window).**

**\*Accessing object properties:**

* To access properties inside of an object’s container, simply use dot notation.

**For example, backpack.pocketNum or backpack.strapLength.left**

There are some cases where you need more control either because you want to use a variable as the property name or because the property name is non-standard.

* In this case, there is also bracket notation.

Example: **To use bracket notation, I need to wrap the property name** (pocketNum) **in quotation marks because it is a string as well as put it in square brackets.**

**Code:**

var query = “pocketNum”;

console.log(“The pocketNum value: ”, backpack[query]);

* This step is necessary to pass a variable into the property name or you need to access a property that is somehow breaking convention, then use bracket notation.

One way the variable could be breaking convention is if the variable has a hyphen in it.

For example, **console.log(“The pocketNum value: ”, backpack[“property-hyphen”]);**

**\*Object methods:**

* In addition to **properties**, **objects** can **contain their own functions** on the **properties** of the **object**.
* When a **function** **is inside of an object**, it’s called a **method**.

**Methods** = **functions sitting inside of an object**.

Example: Create an object with a method inside.

const paperPlate = {

disposable: true,

color: “white”

throwAway: function(disposableStatus){

this.disposable = disposableStatus;

},

};

**\*Classes: Object blueprints**

Classes are new to JS but have been in other languages for some time now.

* There are two ways to declare a class.

By **class declaration:**

Graphical user interface, text, application

Description automatically generated

Or by setting up a **class expression**:

A picture containing text

Description automatically generated

Where you create a constant with a name and set it equal to class with a set of curly braces.

Inside of the class, we use a **constructor method** to **construct the object** created from that class.

* The method first defines the parameters from each of the properties which is done inside of the constructor parenthesis (list of all properties such as name, volume, color, etc.).
* Next is the curly braces where each property is defined and sets their values to the parameters that are passed in from the class.

**How to use a class to create a new object:**

const everydayPack = new Backpack (

“Everyday Backpack”,

30,

“grey”,

);

The **properties** above should **match each one that is in the constructor**. For example, **“Everyday Backpack”** **matches** with the **name property** in the **constructor**. Same goes for **30** being the **volume** and **grey** being the **color**.

* All we would need to do now to add a new unique backpack would be to create another const and variable (hiking backpack for example) = new Backpack ( ) and add the new properties based on what we added to the constructor.

One **important** **note** here is that we can **only use a class after it has been declared.**

* JavaScript reads from the top down.

**\*Template Literals:**

* Back ticks tell the browser that anything inside here is a template literal which means that we can mix HTML and Strings with JavaScript expressions to create templates.

For example, any JavaScript expression can be added to HTML like the following:

const content = `

<main>

<article>

<h1>${backpack.name }</h1

`

**Using JavaScript to inject new content into an HTML document.**

* When the **browser renders a document**, it **creates** a **document object model (DOM).**
* With the **creation of a DOM**, we can then **access that DOM by using JavaScript.**

**\*How to manipulate this using a Template Literal:**

A simple example of this is to use .innerHTML:

For example, running **document.body.innerHTML** would return everything in the body of the HTML file.

* I can override this by running **document.body.innerHTML = “some text”**

Running this command would return **“some text”** instead of **everything in the HTML body**.

**\*Template Literals** – inside a template literal, anytime you want to call in something generated by JavaScript we add a placeholder such as the following example:

**${any JavaScript expression}**

Example from previous object code:

**${everydayPack.name}**

* Even if the name in the everydayPack were to change, the browser would still update the most recent value now that we are injecting the name value into the HTML.

Template Literals are great. However, they are still very new to the JavaScript world and you may not come across many files that use them.

* Instead, you will likely see **standard strings** **used**.

As discussed above, template literals use a dollar sign and curly braces ${ }

* To get the content we want using standard strings, we need to break the string into pieces and use string concatenation.

For example, **const content = “<h1>” + everydayPack.name + “</h1>”**

**\*Important note:** Keep in mind that this is not the way to do things anymore. We have template literals for this reason.

**\*The Document Object Model (DOM):**

When you present a web browser with an HTML document, the browser creates an object model for the document (DOM).

* This DOM describes the hierarchical tree structure for that document.
* Each element relates to the other.

When you want to work with the DOM, locate the elements to use by Document.querySelectorAll( ) and querySelector( ).

Run the following in the console window **document.querySelector( “main” )**

If I want a specific list item for example, I can run the following command in the console:

**document.querySelector(“main li:last-child”)**

Look at the example below to see another way querySelector can be used:

Text

Description automatically generated



This is an example of **how to target things with querySelector** which uses **CSS queries**.

* If you know how to target things using CSS queries, then you know how to target things using JavaScript.

**\*Accessing elements using older methods**

* **Element.getElementsByClassName( )** and **Document.getElementById( )**
* These two methods above used to be the only tools we had for accessing elements using classes and ids.

**Both methods still work.** However, **querySelector( )** **and querySelectorAll( )** were introduced to simplify this process.

Graphical user interface, text, application

Description automatically generated

**\*Modifying element classes:**

**Important note:** remember to look at MDN web docs for more info on each of these methods.

**Element.classList:**

**Goal:** target an element that has multiple classes.

In our example, calling the first class **document.querySelector(“main li:first-child”).classList** gives us a DOM token list which is an array listing out each of the classes in turn.

* If I want to **add a new class**, do the following:

**document.querySelector(“main li:first-child”).classList.add(“new-class”)**

There are other things we can do using dot notation as well such as **.remove**, **.toggle** (used to add a class that is not there/remove it if it is there), and **.replace** to replace one class with another (first item is the class I want to replace separated by a comma and the second item is the class I want to replace it with).

**\*Attributes:** choose any attribute with the **.attributes**

Graphical user interface, text

Description automatically generated

If I want to get specific info from an attribute, run the following **.getAttribute(“src”) command:**

Graphical user interface, text

Description automatically generated

If I want to **add data (string data)** to an **attribute**, run the following **.setAttribute command:**

Graphical user interface, text, application, email

Description automatically generated

**\*Assignment vs. Comparison:**

* color = red is an assignment.
* a = = b is a comparison.
* a = = = b looks for absolute equality meaning the value on the left must equal the value on the right in type and value.

**\*Arrays in code:**

* **Accessing a certain value within an array:** nameOfArray[1]
* **How to add a new value to the end of an array:**

**const collection = [“Piggy”, item, 5, true];**

**collection[2] = “camera”** (replaces 5 with string “camera”)

**collection[collection.length] = “new item”;** (Adds the “new item” string to the end of the current array)

I can also **assign a new item to a slot in the array that doesn’t exist yet further down the chain.**

For example: **collection[9] = “at the end”;**

**\*Array Methods:** For more examples, look to **MDM web docs** under **Static Methods** and **Instance Methods.**

**Link:** [**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

**How to output all values of an array as a string using the join method:**

let backpackContents = [“piggy”, “headlamp”, “pen”];

console.log(backpackContents.join(“ | ”));

\*Output: **piggy | headlamp | pen**

**Another common task is to add one or more items to the end of an array.**

* Use **.push( )** for this.

For example: **backpackContents.push(“pencil”, 5);**

To add items to the front of an array, use **.unshift( )**

**backpackContents.unshift(“pencil”, 5);**

**Why is this method called unshift?**

* **.shift( )** takes the first item off of an array.
* In this case, **.unshift( ) adds the item back to the front of an array.**

If I want to take an item off the back of an array, use the **.pop( ) method.**

**How to apply functions to each item in an array:**

* Example**: Add an li around each of the items and then output them to the console.**

**.forEach( ) –** loop that grabs each item and does something to each item.

* Keep in mind that **.forEach( )** does not do anything to the array itself. Instead, it simply loops over the array to perform some action outside the array.

Here is the output for the example mentioned above. The code for this can be found in the **exercise files (Section 07\_03).**

Graphical user interface, text, application

Description automatically generated

General Array Examples from MDM Webdocs:

**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array**

**\*Functions and Methods:**

* The term “**function”** is used to refer to a **function that sits on itself.**
* A **“method”** is a **function that sits inside an object and acts on that object.**

**These are essentially the same thing but appear in different objects.**

**\*Functions can be created in a few different ways:**

* Keep in mind that **how we create a function** has an **impact** on **how it can be used**.
* The **classic function** is a **function declaration as seen below.**

**\*Function Declaration:**

Text

Description automatically generated with medium confidence

* If we use a **function declaration** like this, the function is **hoisted to the global scope** meaning that it **becomes available everywhere**.
* **Function declarations are parsed in the order they appear in the code.**

For example, if a **function “b” relies on another function “a”**, then **function “a”** must be **declared before.**

**\*Function Expression:**

A picture containing text

Description automatically generated

* This type of function is one that doesn’t have its own name.
* **Function Expressions** are more commonly used as they are block/scope level. In this case, putting a function inside of a const means that it cannot be redeclared.
* **Function Expressions are not hoisted** meaning they exist only in the scope they were created.

**Function Expressions** are a more advanced version of **functions** (in a way).

* To call/use either a **Function Declaration or Expression,** simply **name is and add parenthesis at the end.**

**If there are parameters, remember to pass them in as well.**

For example: **doMoreMath(5, 6);**

If I were to write **doMoreMath;** it will **return the whole function in the console window** if a **console.log(doMoreMath);** is written.

One more type of function type worth mentioning is the **Immediately Invoked Function Expression (IIFE).**

**Graphical user interface, text

Description automatically generated**

**Take note of the syntax here for this type of function.** For example, **the “function” name** alone with no parameters and **extra parens** at the **end of the function**.

* As soon as the browser encounters this function, it will run immediately.