JavaScript:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

Variables:

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Variables>

Math Operator:

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math>

Functions:

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Functions>

Arrays

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Arrays>

Object Basics

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>

VSCode

<https://code.visualstudio.com/docs>

<https://code.visualstudio.com/docs/getstarted/keybindings>

<https://code.visualstudio.com/docs/getstarted/settings>

<https://code.visualstudio.com/docs/editor/extension-gallery>

Chrome Developer Tools

<https://developers.google.com/web/tools/chrome-devtools/>

Precedence

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>

Statement Vs Expression

<https://stackoverflow.com/questions/12703214/javascript-difference-between-a-statement-and-an-expression>

Flow Control and Error Handling

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling>

Loops and Iteration

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration>

Strict Mode

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#Changes_in_strict_mode>

Jet Engine Explained

<https://hackernoon.com/javascript-v8-engine-explained-3f940148d4>

Spider Monkey

<https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals>

Memory Management

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management>

Garbage Collection

<https://v8.dev/blog/free-garbage-collection>

Functions

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions>

Bind

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind>

QuerySelectorAll

<https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll>

Element.getElementsByTagName()

<https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByTagName>

DOM

<https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction>

ARRAYS

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

Summaries:

Summary: Insert, Replace, Remove

There are many ways of creating, inserting, replacing and removing DOM elements - here's a summary of the options you have.

For browser support, check the provided links and also the "Browser Support" module you find later in the course.

#### Create & Insert

You got two main options: Provide an HTML snippet (e.g. via innerHTML) to a valid HTML snippet and let the browser render it OR create a DOM object in JS code and append/ insert it manually. The latter approach has the advantage of giving you direct access to the DOM object (useful for setting its properties or adding event listeners). The downside is that you have to write more code.

**Adding HTML Code:**

1. const root = document.getElementById('root-el'); // selects something like <div id="root-el">
2. root.innerHTML = `
3. <div>
4. <h2>Welcome!</h2>
5. <p>This is all create & rendered automatically!</p>
6. </div>
7. `;

**Important**: Any existing content in root is  completely replaced when using innerHTML. If you want to append/ insert HTML code, you can use insertAdjacentHTML instead: <https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML>

1. const root = document.getElementById('root-el'); // selects something like <div id="root-el">
2. root.insertAdjacentHTML('afterbegin', `
3. <div>
4. <h2>Welcome!</h2>
5. <p>This is all create & rendered automatically!</p>
6. </div>
7. `);

**Creating & Inserting DOM Objects Manually:**

1. const someParagraph = document.createElement('p'); // creates a "p" element (i.e. a <p> element)
2. const root = document.getElementById('root-el'); // selects something like <div id="root-el">
3. root.append(someParagraph);

In this example, we create a paragraph and append it to root - append means that it's inserted at the end of root (i.e. inside of it but AFTER all other child nodes it holds).

**Insertion Methods:**

**append()** => <https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/append>

Browser support is decent but for IE, **appendChild()** could be preferred => <https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild>

**prepend()** => <https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/prepend>

Browser support is decent but for IE, **insertBefore()** could be preferred => <https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore>

**before()**, **after()** => <https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/before> & <https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/after>

Browser support is okay but IE and Safari don't support it. Consider **insertBefore()** (<https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore>) or **insertAdjacentElement()** (<https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentElement>) as substitutes.

Important (no matter how you insert elements): Whenever you insert elements, you **MOVE the element** to that new place if you already inserted it before. It's **NOT copied** (you can copy an element via someElement.cloneNode(true) though).

#### Replace

You can replace elements in the DOM with two main methods:

**replaceWith()** => <https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/replaceWith>

**replaceChild()** => <https://developer.mozilla.org/en-US/docs/Web/API/Node/replaceChild>

replaceWith() is a bit easier to use and has decent browser support - with IE being the exception. To support that as well, consider using replaceChild().

#### Remove

You can remove elements with three main methods:

**someElement.innerHTML = ''** => Clears all HTML content of someElement and hence removes any objects rendered in there.

**someElement.remove()** => Removes a single element (someElement) from the DOM (<https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/remove>). Browser support is good, IE again doesn't like it though. Use removeChild (see below) instead.

**someElement.parentNode.removeChild(someElement)** =>  Removes the provided child element (NOT the element on which you call it). Provides broad browser support but of course requires a bit more code (<https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild>).

#### What about Text Nodes?

You can easily create & insert text nodes in one go:

1. someElement.textContent = 'Hi there!';

This creates and inserts the text node with a content of 'Hi there!'.

**Want to append to existing text?**

Just use:

1. someElement.textContent = someElement.textContent + 'More text!';

"this" - Summary

The this keyword can lead to some headaches in JavaScript - this summary hopefully acts as a remedy.

this refers to different things, depending on where it's used and how (if used in a function) a function is called.

Generally, this refers to the "thing" which called a function (if used inside of a function). That can be the global context, an object or some bound data/ object (e.g. when the browser binds this to the button that triggered a click event).

**1) this in Global Context (i.e. outside of any function)**

1. function something() { ... }
3. console.log(this); // logs global object (window in browser) - ALWAYS (also in strict mode)!

**2) this in a Function (non-Arrow) - Called in the global context**

1. function something() {
2. console.log(this);
3. }
5. something(); // logs global object (window in browser) in non-strict mode, undefined in strict mode

**3) this in an Arrow-Function - Called in the global context**

1. const something = () => {
2. console.log(this);
3. }
5. something(); // logs global object (window in browser) - ALWAYS (also in strict mode)!

**4) this in a Method (non-Arrow) - Called on an object**

1. const person = {
2. name: 'Max',
3. greet: function() { // or use method shorthand: greet() { ... }
4. console.log(this.name);
5. }
6. };
8. person.greet(); // logs 'Max', "this" refers to the person object

**5) this in a Method (Arrow Function) - Called on an object**

1. const person = {
2. name: 'Max',
3. greet: () => {
4. console.log(this.name);
5. }
6. };
8. person.greet(); // logs nothing (or some global name on window object), "this" refers to global (window) object, even in strict mode

this can refer to unexpected things if you call it on some other object, e.g.:

1. const person = {
2. name: 'Max',
3. greet() {
4. console.log(this.name);
5. }
6. };
8. const anotherPerson = { name: 'Manuel' }; // does NOT have a built-in greet method!
10. anotherPerson.sayHi = person.greet; // greet is NOT called here, it's just assigned to a new property/ method on the "anotherPerson" object
12. anotherPerson.sayHi(); // logs 'Manuel' because method is called on "anotherPerson" object => "this" refers to the "thing" which called it

If in doubt, a console.log(this); can always help you find out what this is referring to at the moment!

More on “This” key word

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>

Classes

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

Prototypes - Summary

Prototypes can be a confusing and tricky topic - that's why it's important to really understand them.

A prototype is an object (let's call it "P") that is linked to another object (let's call it "O") - it (the prototype object) kind of acts as a "**fallback object**" to which the other object ("O") can reach out if you try to work with a property or method that's not defined on the object ("O") itself.

**EVERY object in JavaScript by default has such a fallback object** (i.e. a prototype object) - more on that in the next lectures.

It can be especially confusing when we look at how you configure the prototype objects for "to be created" objects based on constructor functions (that is done via the .prototype property of the constructor function object).

Consider this example:

1. function User() {
2. ... // some logic, doesn't matter => configures which properties etc. user objects will have
3. }
4. User.prototype = { age: 30 }; // sets prototype object for "to be created" user objects, NOT for User function object

The User function here also has a prototype object of course (i.e. a connected fallback object) - **but that is NOT the object the prototype property points at**. Instead, you access the connected fallback/ prototype object via the **special \_\_proto\_\_ property** which EVERY object (remember, functions are objects) has.

The prototype property does something different: **It sets the prototype object new objects which you create with this User constructor function will have.**

That means:

1. const userA = new User();
2. userA.\_\_proto\_\_ === User.prototype; // true
3. userA.\_\_proto\_\_ === User.\_\_proto\_\_ // false

Constructor Functions

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Using_a_constructor_function>

Prototypes

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes>