

JavaScript

Basic JavaScript syntax

Variables

```
// An immutable value available from the line
// where the constant is defined
const constValue = "hey";
constValue = "you"; // ⚠️ ERROR

// A mutable value available from the line
// where the variable is defined
let variableValue = 0;
variableValue = 1;

// A mutable value, old syntax. Hoists to the beginning
// of the function scope or to the global scope,
// whatever is closer. Not recommended for use.
var anotherVariableValue = false;
```

print statements

```
console.log("Hello, world");  
  
const name = "Alex";  
const age = "20";  
  
console.log("My name is ", name);  
console.log("I am " + age + " years old");
```

Variables in strings

You can display the contents of a variable in a string using the `${variable}` syntax

Note: the string is surrounded by backticks when using this syntax

```
const name = "Alex";  
const age = 15;  
  
console.log(`My name is ${name}. I am ${age + 5} years old.`);  
  
console.log(`Right now it is ${new Date.toString()}`);
```

Data Types

```
let value;
```

```
Number;
```

```
// A float or integer number
```

```
value = 0.1;
```

```
String; // Strings are an Array of Characters.
```

```
value = "some-string";
```

```
console.log(value[2]); // prints 'm'
```

```
null; // In JavaScript, null is a value
```

```
value = null;
```

```
undefined; // This means a variable hasn't been assigned a value
```

```
undefVal;
```

```
console.log(undefVal); // is undefined
```

```
console.log(typeof undefVal); // is undefined
```

```
Boolean; // A boolean value, either true or false
```

```
value = true;
```

Equality

```
# Python
if 1 == "1": # False
    print('hello')
```

JavaScript is a little different when it comes to comparing things

→ `==` means loose equality

→ `===` means strict equality

```
const looseEqual = "2" == 2; // true
const strictEqual = "2" === 2; // false
```

Ternary operator and short IF statements

```
if (num % 2 === 0) {  
  console.log("Number is even");  
} else {  
  console.log("Number is odd");  
}
```

```
// formatting when statements are longer  
num % 2 === 0  
  ? console.log("This number is even")  
  : console.log("This number is odd");
```

```
// Alternatively, you can minimise the scope to only the things that actually change  
console.log(num % 2 === 0 ? "This number is even" : "This number is odd");  
console.log("This number is " + (num % 2 === 0 ? "even" : "odd"));
```

```
// One liner  
if (condition) {  
  runSomething();  
}  
condition && runSomething();
```


Objects

An object, which is basically a dictionary of dynamically typed values identified by their keys

```
value = { key: "hey", anotherKey: 10 };  
value = {  
  key: "hello there1",  
  names: ["Alex", "Sam", "Jason", "Henry"],  
  anotherObject: {  
    1: "a",  
    2: "b",  
    3: "c",  
  },  
};
```

See: JavaScript Object Notation (JSON)

Objects (cont'd)

```
// Object keys can be accessed in two ways
const obj = {
  name: "Alex",
};

// these are equivalent
let name1 = obj.name;
let name2 = obj["name"];

// the same can be done when assigning values to an object
obj.name = "Sam";
obj["name"] = "Sam";

// however, if you want a key with a name tied to a variable, you must use the second method
const keyName = "age";
obj.keyName = 20; // ❌ → { keyName: 20 }
obj[keyName] = 20; // ✅ → { age: 20 }, note the lack of quotes,
```

Functions

Functions in JavaScript can be assigned to variables and referenced.

```
// Normal function declaration
function addTwo(a, b) {
  console.log(a + b);
}

// function assigned to variable
const foobar = function (foo, bar) {
  console.log(foo + bar);
};

// anonymous function . Useful when functions are passed as parameters,
// esp. single use functions
const anduril = () => {
  console.log("Flame of the West");
};

// calling a function
someFunction("a", "b", () => console.log("c"));
```

Destructuring

Destructuring is the process of breaking down an array or object.

```
let a, b, rest;  
[a, b] = [10, 20];  
  
console.log(a);  
// Expected output: 10  
  
console.log(b);  
// Expected output: 20  
  
[a, b, ...rest] = [10, 20, 30, 40, 50];  
  
console.log(rest);  
// Expected output: Array [30, 40, 50]
```

From: [MDN | Destructuring Assignment](#)

Destructuring (cont'd)

You can use destructuring to **shallow** copy an array

Note: `arr2 = arr1` does not copy `arr1`. `arr2` just points to `arr1` in memory.

```
let arr1 = [10, 20, 30];  
let arr2 = [...arr1]; // => [10, 20, 30]
```

Destructuring can also be used to swap values, or assign to multiple variables

```
// swap  
let a;  
let b;  
  
[a, b] = [b, a];  
  
console.log(a); // => 1  
console.log(b); // => 2
```

Advanced Functions

| a closer look at `map`, `filter`, and `reduce`

.map()

The `.map()` function is used to apply a function to each element of an array.

Traditional

```
const numbers = [1, 2, 3, 4, 5];
const doubledNums = [];

for (const num of numbers) {
  doubledNums.push(num * 2);
}
```

`.map()`

```
const numbers = [1, 2, 3, 4, 5];

numbers.map((num) => {
  return num * 2;
});
```

.filter()

Takes an array and filters out all elements that do not satisfy a certain condition. You're left with a potentially smaller array.

Some code to filter even numbers:

Traditional

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

evenNums = [];
for (const num of numbers) {
  if (num % 2 === 0) {
    evenNums.push(num);
  }
}
```

.filter()

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

numbers.filter((num) => {
  return num % 2 === 0;
});
```


.reduce()

`.reduce()` runs a reducer function on each array element and returns a single output.

The best example is a sum function:

Traditional

```
const students = [
  { name: "Alex", mark: 55 },
  { name: "Ben", mark: 43 },
  { name: "Chloe", mark: 34 },
];

let sum = 0;
for (const student of students) {
  sum += student.mark;
}
```

`.reduce()`

```
const students = [
  { name: "Alex", mark: 55 },
  { name: "Ben", mark: 43 },
  { name: "Chloe", mark: 34 },
];

const add = (prev, curr) => prev + curr.mark;
const sum = students.reduce(add, 0);

// Alternatively
const sum = students.reduce((prev, curr) => {
  prev + curr.mark;
}, 0);
```



For writing backend code and creating frontend apps locally, NodeJS is widely used.

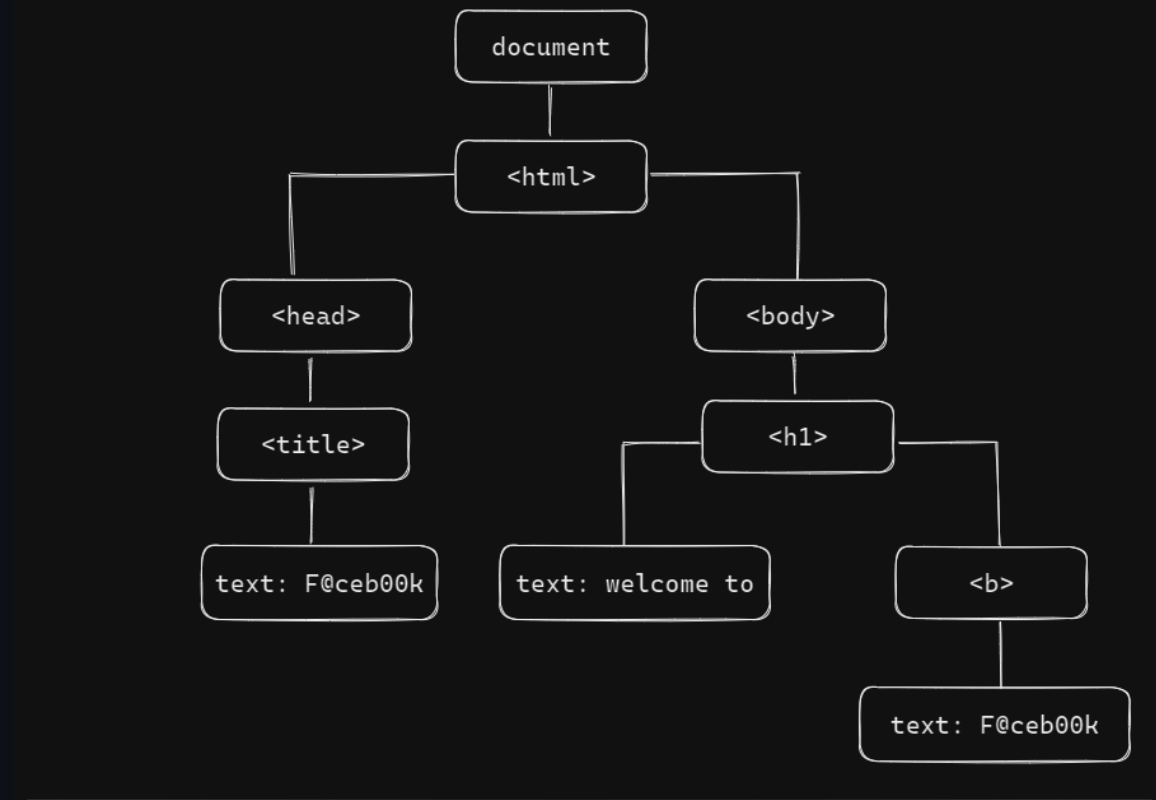
WebJS

used to manipulate the DOM in a web browser

The DOM

Stands for Document Object Model. It's an interface that allows JavaScript to interact with HTML through the browser.

```
<!DOCTYPE html>
<html>
  <head>
    <title>F@ceb00k</title>
  </head>
  <body>
    <h1>Welcome to <b>F@ceb00k</b></h1>
  </body>
</html>
```



The DOM is a tree!

DOM data types

`document` → the root of the entire DOM tree

Elements → a node in the DOM tree.

`NodeList` → an array of elements

Different Element types include:

```
HTMLInputElement, HTMLSpanElement, HTMLDivElement,  
HTMLScriptElement, HTMLHeadingElement, HTMLImageElement, ...
```

You easily can guess which HTML tags the above refer to.

Reading from the DOM

```
document.getElementById(id); // returns a single NodeElement
```

```
document.getElementsByTagName(name); // returns an array [...]
```

```
document.getElementsByClassName(className); // returns an array [...]
```

```
document.querySelector(query); // returns an array [...]
```

Writing to the DOM

```
// create a new div
let element = document.createElement("div");
/* (1)
  <document>
    <div></div>
  </document>
*/
let textNode = document.createTextNode("Hello there!");
/* (2)
  <document>
    <div></div>
    Hello there!
  </document>
*/
// adding elements
element.appendChild(textNode);
/* (3)
  <document>
    <div>
      Hello there!
    </div>
  </document>
*/
element.removeChild(textNode); // removes the text node, same as (1)
```

Writing to the DOM (cont'd)

```
...  
button.setAttribute("disabled", true);  
  
// Changing element. classList  
element.classList.add("class");  
element.classList.remove("class");  
element.classList.toggle("class");  
element.classList.contains ("class"); // returns true if class exists on element
```


Styling in JS

Styles can be changed via the `styles` property.

```
// Style Changes:  
element.style.left = "50px"; // same as in CSS → left: 50px;  
element.style.backgroundColor = "red"; // same as → background-color: red;
```

Note: all `kebab-case` properties from HTML and CSS are all `camelCase` in JavaScript.

e.g., `background-color` → `backgroundColor`

e.g., `flex-direction` → `flexDirection`

Events

An *event* is when "something" happens. This "something can be" a loading state, a key press or clicking. We use events to trigger certain actions.

Common events

Common events include

↳ `click`

↳ `dblclick`

↳ `mouseup`

↳ `mousedown`

↳ `mouseleave`

↳ `keyup`

↳ `keydown`

↳ `keypress`

↳ `focus`

↳ `hover`

↳ `blur`

↳ `change`

Setting events

There are 3 ways to set events

In HTML

```
<button onclick="myFunction()">Click me!</button>
```

In JS, tied to the DOM

```
let element = document.getElementById("btn");
element.onclick = () => {
  console.log("Clicked!");
};
```

Using event listeners

```
let element = document.getElementById("btn");
let handler = () => {
  console.log("Clicked!");
};
element.addEventListener("click", handler);
element.removeEventListener("click", handler);
```

Event handling

When a user interacts with the page, an `event` is triggered. A piece of JS code is then run.

Every event trigger contains an event object. This can be read as a parameter:

```
document.addEventListener("mousemove", (event) => {  
  // prints the position of the user's mouse  
  console.log(event.clientX);  
  console.log(event.clientY);  
});
```

Note: this *event* object is specifically a **MouseEvent**. Events are context specific.

e.g., a `mousemove` event will have property `event.clientX` (but not `event.key`)

e.g., a `keydown` event will have property `event.key` (but not `event.clientX`)

Event Bubbling

An event triggered within a child element can often trigger in the parent element as well.

```
<div id="btndiv">
  <button id="btn">Click Me!</button>
</div>
```

```
document.getElementById("btndiv").addEventListener("click", () => {
  console.log("OUTSIDE");
});

// Clickin the btn will print "INSIDE" and "OUTSIDE"
// because you've technically clicked the parent btnDiv as well
document.getElementById("btn").addEventListener("click", () => {
  console.log("INSIDE");
});
```

The Event Loop

The JavaScript Engine executes the following steps:

1. Execute the JS Script from top to bottom, registering any event handlers
2. Wait for the next JS event and loop
3. If an event comes in and there exists a handler for it, execute the handler **to completion**
4. Repeat step 2 forever

Note: The BROWSER is in charge of:

- counting down timers
- network stuff
- adding new events to the event queue

Asynchronous JavaScript

Not all things on the web is done in synchronously. Sometimes we wait for multiple responses from the servers.

We need to use JavaScript asynchronously when it comes to web development.

We want the website to be operation while we're processing a longer task. For example, we want to still be able to navigate the site after we submit a form, even if the submission is ongoing to the backend.

Promises

In JavaScript, a `Promise` object signals an asynchronous execution of code. In other words, it's a promise that some code will be run at some time.

Promises (cont'd)

- A good scenario to imagine is at a party. You tell your friend to go and buy 5 packs of beers.
- That is a `Promise`. The friend will go out to buy the beers while you do something else, like procuring some plastic cups.
- When your friend returns, they will either have the 5 packs of beers or they won't.
- Depending on the result, different actions will be taken. (Pour the beers) or (Pour soft drinks)

Promises make up the backbone of JS web development.

Promises (cont'd)

Promises have 4 states:

Pending → not yet complete

Fulfilled → successfully evaluated

Rejected → evaluation failed

Settled → evaluation complete (regardless of success or failure)

Promises (cont'd)

🤔: Imagine I have in my file directory, 3 Chapters of a book:

```
const fs = require("fs");
fs.readFile("chapter1.txt", "utf-8", (err, data) => {
  err ? console.log("Error: ", err) : console.log("Chapter 1: ", data);
});

fs.readFile("chapter2.txt", "utf-8", (err, data) => {
  err ? console.log("Error: ", err) : console.log("Chapter 2: ", data);
});

fs.readFile("chapter3.txt", "utf-8", (err, data) => {
  err ? console.log("Error: ", err) : console.log("Chapter 3: ", data);
});
```


What is the output?

Promises (cont'd)

So how can we ensure the output is synchronous when the processes themselves aren't?

`.then()` and `.catch()`

`.then()` runs if and only if the previous promise is fulfilled (i.e., success )

`.catch()` is run when a nearby promise is rejected (i.e., failure )

Promises (cont'd)

Hence, we can chain these promises one after the other.

```
const fs = require("fs").promises;
const files1 = fs.readFile("chapter1.txt", "utf-8"); // returns a promise to read chapter 1
files1
  .then((data) => {
    // only once chapter 1 is successfully read:
    console.log("Chapter 1: ", data);
    return fs.readFile("chapter2.txt", "utf-8");
  })
  .then((data) => {
    console.log("Chapter 2: ", data);
    return fs.readFile("chapter3.txt", "utf-8");
  })
  .then((data) => {
    console.log("Chapter 3: ", data);
  })
  .catch((err) => {
    console.log("Error: ", err);
  });
```

Promise Orchestration

The `Promise` class has some useful functions to help manage asynchronous tasks. `resolve` will be used in place of `fulfill` to align with JS syntax.

`Promise.all([...])` returns a `Promise` which resolves (✓) iff all given promises are resolved (✓)

`Promise.allSettled([...])` returns a `Promise` which resolves (✓) if all given promises are settled (✓ or ✗)

`Promise.any([...])`

- resolves (✓) when at least one of the given promises is resolved
- rejects (✗) when all given promises are rejected

Promise Orchestration (cont'd)

```
Promise.race([...])
```

→ resolves (✅) when any of the given promises is first to settle (✅ or ❌)

```
Promise.reject(val)
```

immediately reject a promise

```
Promise.resolve(val)
```

immediately resolve a promise

Fetch API

The Fetch API is native to JS and is used to to download and upload data to backend servers.

The Fetch API uses Promises.

fetch()

```
fetch(`http://localhost:${BACKEND_PORT}/${route}`, {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(bodyData);
})
.then((response) => {
  return response.json();
})
.then((data) => {
  data.error
    ? throw new Error(data.error);
    : return data;
})
.catch((error) => {
  alert(error);
});
```

`fetch()` (cont'd)

- resolves (✓) if a response is received, even when the HTTP status is not 200
- if there is a network error, the `fetch()` promise immediately rejects

When writing `fetch()` requests, try to make a reusable function instead of rewriting one each time.

fetch() example

```
export function fetchAPIRequest(route, httpMethod, bodyData) {
  const options = {
    method: httpMethod,
    headers: {
      "Content-Type": "application/json",
    },
  };

  httpMethod !== "GET" && options.body = JSON.stringify(bodyData);

  localStorage.getItem("userToken") &&
    options.headers.Authorization = `Bearer ${localStorage.getItem(
      "userToken"
    )}`;

  return fetch(`http://localhost:${BACKEND_PORT}/${route}`, options)
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      data.error
        ? throw new Error(data.error);
        : return data;
    })
    .catch((error) => {
      error.message === "Failed to fetch"
        ? alert("Unable to fetch from server.", false);
        : alert(error)
    });
}
```

CRUD / HTTP Request types







POST Used to **CREATE** or write **new** data to the backend. Requires a **body** property

GET **READS** data from the backend. Does not require a **body** property


PUT Used to **UPDATE existing** data in the backend. Requires a **body** property

DELETE **DELETES** data entirely in the backend. Requires a **body** property

Common HTTP Status Codes

- 200  OK
- 400  Bad Request - your fetch request is missing data or has incorrect properties. Usually this is a misspelt body object key.
- 403  Forbidden - You're not allowed to access this resource. Usually occurs when you're logged out.
- 404  Not Found - The resource you're trying to access doesn't exist. Usually an incorrect url which the backend doesn't have a route for.
- 500  Internal Server Error - the backend is broken somewhere. This error is very generic
- 503  Service Unavailable - backend is (probably) turned off.

Homework

 Watch the following videos (preferably on 2x speed). These are from the COMP6080 course.

- DOM
- Events
- Forms
- Promises

 **mdn web docs**

Note: if in doubt, search it up. A good place to start is the [MDN Web Docs](#). Search anything Web dev related on google and read up on the MDN docs.