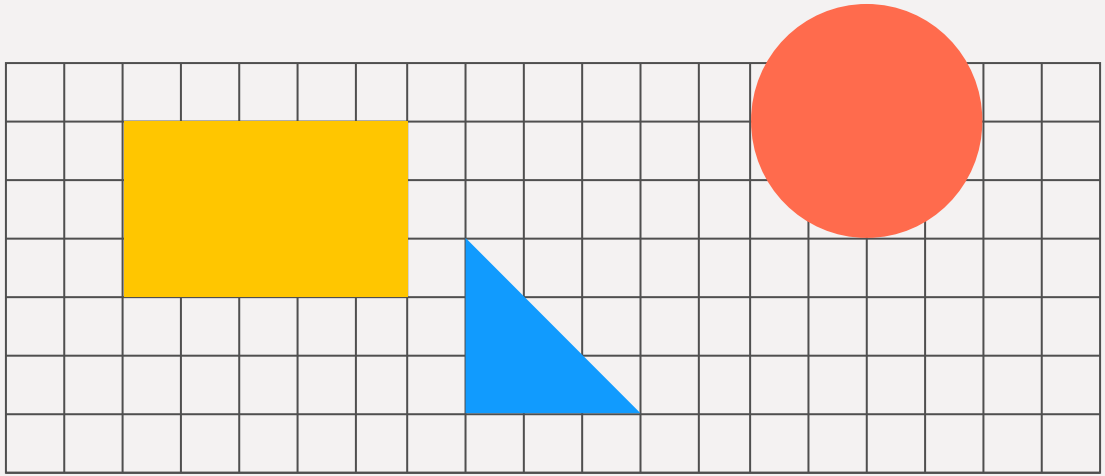
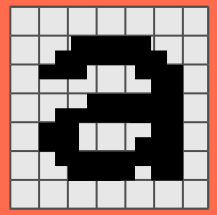
Make ‘em Interactive!

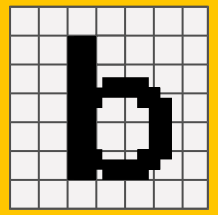
Ali Abrishami



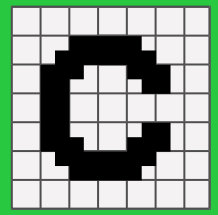
In This Lecture You Will...



Learn What JavaScript is



Understand it's features and limitations



Learn how async programming works in JS

REMINDER: The Three Layers of a Website

Structure

What the **content** is

Style

How it **looks**

Behaviour

How it **reacts**

What We Can Do So Far…

- We learned how to specify the **structure** for our web pages.
- We learned how to **divide** our **contents** into meaningful **sections**.
- We learned how to **style** our web pages.
- We learned the **abilities** and **limits** of **CSS**.
- We still CAN’T make our web pages **interactive**…

What is JavaScript?

Introduction

- JavaScript (JS) is an actual programming language
- It is able to run in your browser AND your machine (via Node.JS, but more on that later!).
- Brendan Eich created the initial prototype of JavaScript.
- He created it in just **10** days in May 1995 while working at Netscape.

History of JavaScript

- 1995: Brendan Eich creates the very first version of JavaScript in just 10 days. It was called Mocha.
- 1996: Mocha changes to LiveScript and then to JavaScript, in order to attract Java developers. However, JavaScript has almost nothing to do with Java.
- 1996: Microsoft launches Internet Explorer, copying JavaScript from Netscape and calling it JScript.
- 1997: With a need to standardize the language, ECMA releases ECMAScript 1 (ES1), the first official standard for JavaScript (ECMAScript is the standard, JavaScript the language in practice)

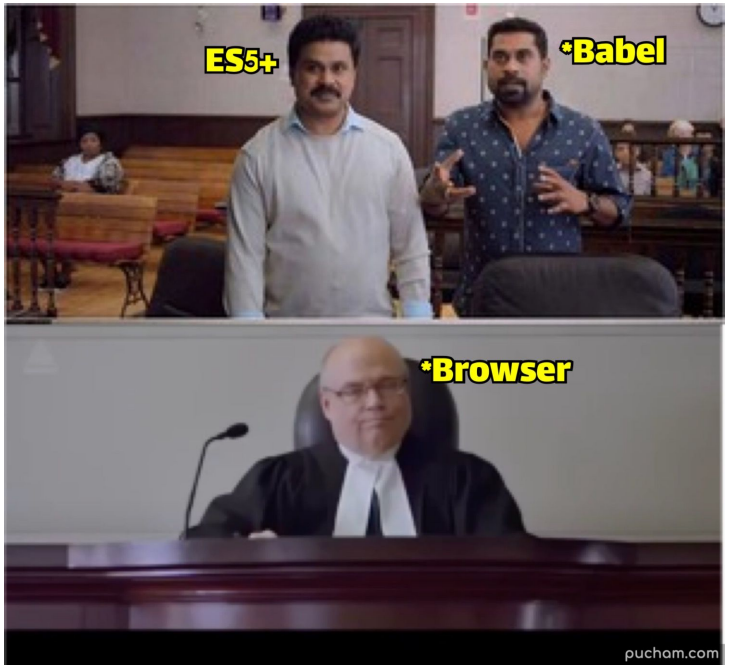
A Modern JavaScript

- 2009: ES5 (ECMAScript 5) is released with lots of great new features.
- 2015: ES6/ES2015 (ECMAScript 2015) was released (the biggest update to the language ever).
- 2015: ECMAScript changes to an annual release cycle in order to ship less features per update.
- 2016 - Present: Release of ES2016 / ES2017 / ES2018 / ES2019 / ES2020 / ES2021 / …

Source: Medium

JS Transpile

- Converts new JS to older syntax
- Keeps code working on **old browsers**
- Uses tools like Babel or SWC
- Lets devs use future JS features
- Adds a small build step



Source: X

Class Activity Time!

These are some of the features of JavaScript. What does each one of them mean?

High Level

Garbage Collected

Interpreted/JIT Compiled

Multi-paradigm

Prototype-based OO

First-class functions

Dynamic

Single threaded

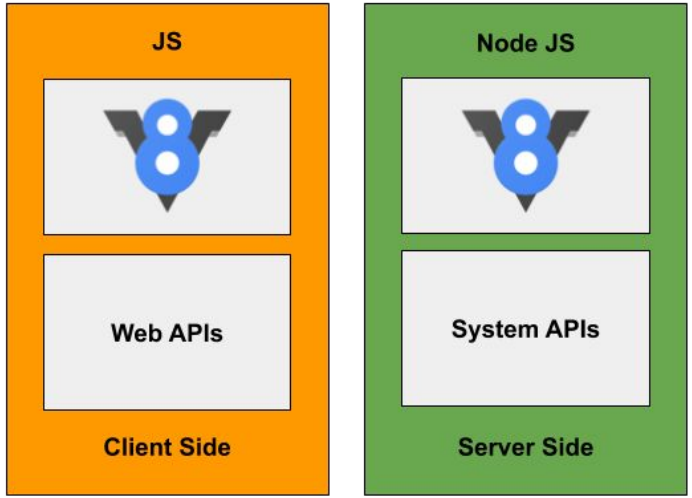
Non-blocking event loop

How Does It Run?

- Runs **line-by-line** inside the browser via an engine or on your machine via Node.js.
- Chrome uses **V8**, Firefox uses **SpiderMonkey**, etc.
- Code is parsed, optimized, and compiled to **machine code**.
- Handles async tasks (timers, promises, user events) without blocking.
- Browser or Node.js provides APIs (DOM, HTTP, File system, etc.). We'll talk about these APIs further in the course!.

JavaScript APIs

- JavaScript isn't just about data and functions.
- It runs inside an **environment** (a browser, Node.js, etc.)
- These environments provide **APIs** (Application Programming Interfaces).
- APIs let JavaScript interact with the outside world (show messages, fetch data, manipulate the page).
- Without these APIs, JavaScript alone couldn't do much beyond calculations.



Source: wanganator414.github.io

The Console API

- A built-in tool provided by browsers.
- Used mainly for **debugging** and **inspecting** values.
- Accessible through the global console object.
- Provides many methods (log, warn, error, table, etc.).
- The **most common** and first one we use: console.log().

Projects/University/web-ta/sampleprojects/js/console.js

```
console.log("Hello, world!");

console.info("THIS IS AN INFO MESSAGE.");

console.warn("YOU WERE WARNED WHAT DEFEAT WOULD BRING!!!");

console.error("AH DOODOO, HERE WE GO AGAIN!");
```

What does each line do?

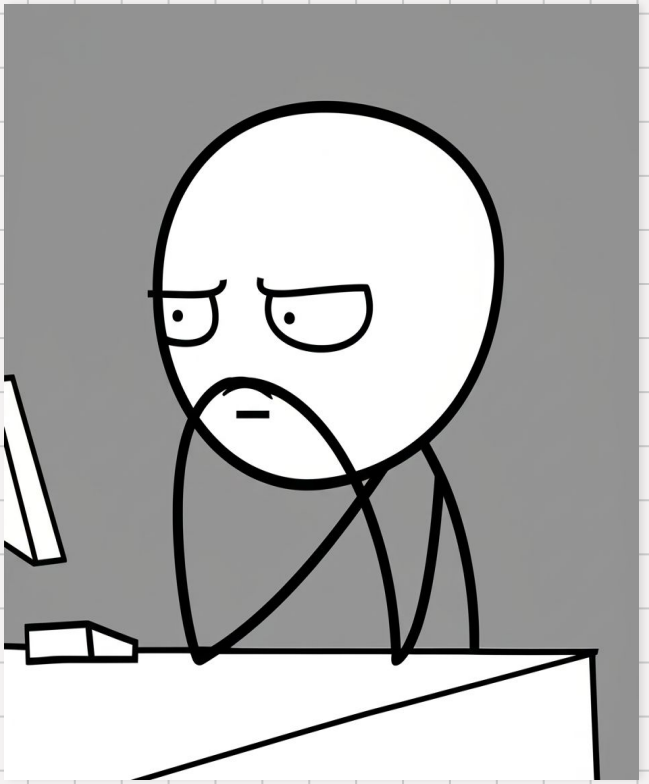
But Where Do I Code?

- Browser Console:
  1. Open “DevTools”
  2. Go to “Console” tab
  3. Type JS codes directly.

- Inside HTML: Write your JS code between `<script> ... </script>`.

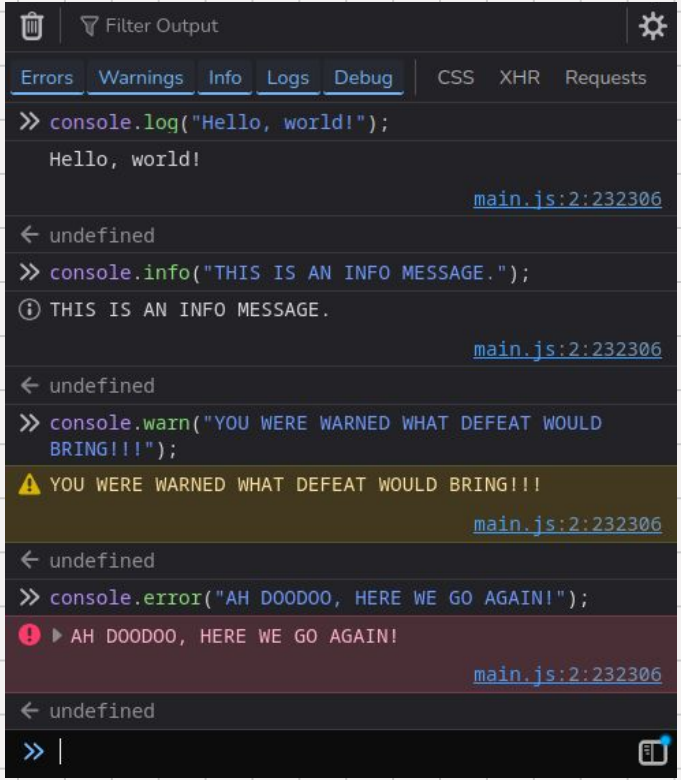
- External JS File:
  1. Create a .js file
  2. Link your .js file with `<script src="...">` (like what you did with your CSS)

- Online Playgrounds: e.g. CodePen, JSFiddle, StackBlitz.



Testing That Console Code

- For the sake of simplicity, we'll run our JS codes inside the developer tools console (for now).
- It runs our JS code line by line (like Python shell).
- This method is great when you're learning the basics of JavaScript.
- You can see that our codes works properly :).
- You can see that there is an obvious difference between different types of messages. Since console methods return nothing, the return values are `undefined`.



Data In JavaScript

- Data has two main types in JavaScript: Everything is either a **primitive**, or an **object**.
- There are **7 primitive data** types in JavaScript: `number`, `string`, `boolean`, `undefined`, `null`, `Symbol`, and `BigInt` (more on this in the next slide!).
- Variables can hold **any type** and change at runtime (dynamic typing).
- Arrays, functions, and even dates are specialized objects.
- Conversion of types can happen either automatically (**coercion**) or explicitly (**casting**).

The Primitives

- `number`: integers & floating-point values
- `string`: sequence of characters in quotes
- `boolean`: only `true` or `false`
- `undefined`: variable declared but not assigned
- `null` → intentional absence of a value
- `Symbol` → unique, immutable identifier
  - Represents a unique and immutable identifier (often used as keys for object properties).
- `BigInt` → integers larger than `Number.MAX_SAFE_INTEGER`

More Complex Objects

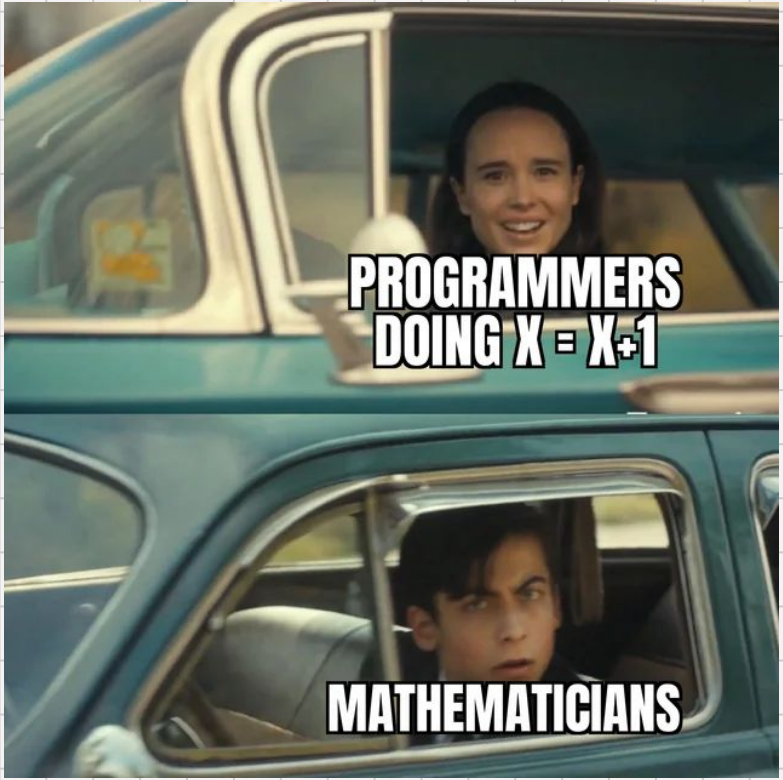
- Objects group related data and behavior into **key-value pairs**.
- Objects can nest other objects, arrays, or functions.
- Arrays hold ordered collections, accessed with numeric indices.
- Arrays provide **built-in methods** (like Java).
- Functions are **first-class objects** (can be passed, stored, and returned).
- Functions can act as object methods or standalone code blocks.

Arithmetic Operators

- +: addition
- -: subtraction
- *: multiplication
- /: division
- %: remainder (modulo)
- **: exponentiation

Assignment Operators

- `=` assign value
- `+=` add & assign
- `-=` subtract & assign
- `*=` multiply & assign
- `/=` divide & assign
- `%=` remainder & assign

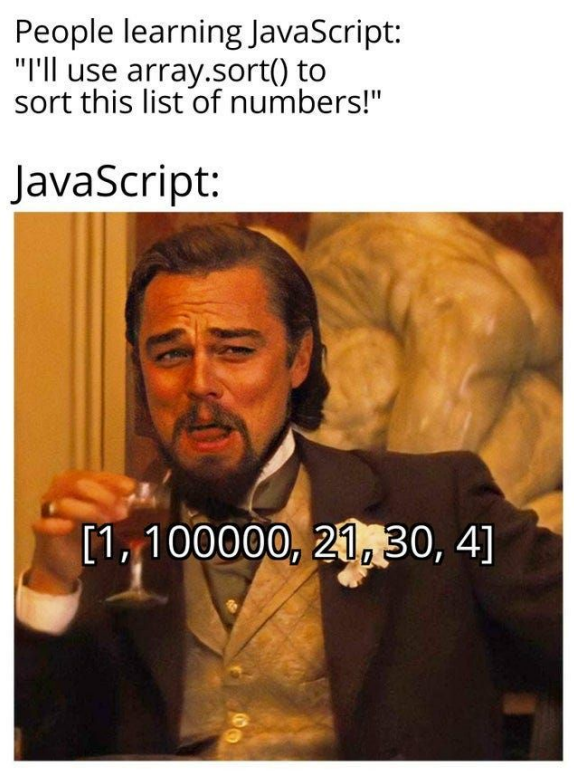


Casting

- Casting is converting one type of value into another. JavaScript has **explicit** casting:
  - `Number("42") → 42`
  - `String(99) → "99"`
  - `Boolean(1) → true`
- Also has implicit casting (**type coercion**):
  - `"5" + 1 → "51"`
  - `2 * "3" → 6`
  - What is the result of `“5” - 1` ?

Casting Pitfalls

- `"5" + 1 → "51"` (string concatenation instead of math).
- `true + true → 2` (booleans become numbers).
- `null + 1 → 1` but `undefined + 1 → NaN`.
- `[] == false → true`, also `[] == 0 → true`
- `==` can cause unexpected coercion, so prefer `===`.
- Look at the meme in the right; can you tell what's going on when that happens?



Type Coercion

- Automatically converting one data type to another when operations involve values of **different types**.
- It's one of the key features of JavaScript's **dynamic typing system** — variables don't have fixed types, and the engine tries to make sense of operations even if the operands aren't of the same type.:
  - ➕ String + Anything → String
    ```
    '10' + 5; // "105"
    ```
  - ➖, ✖️, ➗ → Numeric Context
    ```
    '10' - 5;   // 5
    '10' * 2;   // 20
    ```
  - Comparison Operators:
    ```
    '5' == 5;    // true  (string → number)
    false == 0;  // true  (boolean → number)
    null == undefined; // true (special rule)
    ```

Variables in JavaScript

- Variables are containers for storing data.
- Declared using `let`, `const`, or the older `var`.
- `let`: **changeable values** (mutable).
- `const`: **fixed values** (cannot be reassigned).
- `var`: *outdated*, function-scoped (avoid in modern code).
- Variables let us reuse and manipulate data in programs.

`let` vs. `const`

- with `let`, your variables can be reassigned to new values.
- with `const`, however, your variables cannot be reassigned once set.
- Both are **block-scoped** (only exist inside `{ ... }`).

Projects/University/web-ta/sampleprojects/js/letconst.js

```
let age = 21;

age = 22;   // ✅ works

const pi = 3.14;

pi = 3.1415;   // ❌ error
```

BEST PRACTICE Use `const` by default, switch to `let` when reassignment is needed.

Blocks & Scopes

- A block is code inside { ... }.
- Variables declared with let or const exist only inside that block.
- Outside the block, they are **not** accessible.
- The code in the right shows a block.

Projects/University/web-ta/sampleprojects/js/scopes.js

```
{
  let x = 10;
  console.log(x); // ✅ works
}
console.log(x);   // ❌ error: x is not defined
```

blocks aren't just syntax, they're a tool to organize and protect variables.

Global Scope

- A variable declared **outside** any block is global.
- Global variables can be accessed from anywhere in the code.
- They live for as long as the program runs.
- In browsers, global variables become properties of the `window` object (more on that later).

Projects/University/web-ta/sampleprojects/js/global.js

```
let message = "Hello";   // 🌍 global

function greet() {
  console.log(message); // ✅ accessible here
}

greet();
console.log(message);    // ✅ accessible here too
```

Too many globals can cause naming conflicts and bugs. They make code harder to maintain and debug.

`var` Problems

- Function-scoped, not block-scoped: can leak outside `{}` blocks.
- Variable hoisting: declarations move to the top, causing unexpected undefined.
- Allows redeclaration: same variable can be defined multiple times without error.
- Global scope pollution: `var` in global context attaches to window.
- Inconsistent behavior: makes debugging and reasoning about code harder.

Closure

- A function that remembers variables from its outer scope
- Keeps access even after the outer function ends
- Used for encapsulation, state, and callbacks
- Common in JS and other functional languages
- **BEWARE OF PITFALL:** Unintended variable sharing between closures in loops.

sample.js

```
function counter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}

const c = counter();
console.log(c()); // 1
console.log(c()); // 2
```

Control Flow

- Control flow decides which code runs and when.
- By default, JS runs top to bottom.
- Conditionals let us branch (e.g. `if`, `else`).
- Loops let us repeat code (`for`, `while`).
- Together, they give logic and structure to programs.

Projects/University/web-ta/sampleprojects/js/cf.js

```
let age = 18;

if (age ≥ 18) {
  console.log("You can vote!");
} else {
  console.log("Too young to vote.");
}

for (let i = 1; i ≤ 3; i++) {
  console.log("Count:", i);
}
```

Conditionals

- Allow code to make decisions.
- Run certain blocks only if a condition is true.
- Main forms:
  - `if`: run code if condition is true.
  - `if / else`: choose between two paths.
  - `else if`: chain multiple conditions.
  - `switch`: handle many specific cases.

Comparison Operators

- Equal (==) compares values (allows type coercion).
  - Note: Primitives are compared by value, not by reference.
- Strict Equal (===) compares values and types.
- Not Equal (!=), Strict Not Equal (!==).
- Greater / Less Than: >, <.
- Greater / Less Than or Equal: >=, <=.
- Logical operators are just like C: && (AND), || (OR), ! (NOT). You can also combine or invert boolean values.

Loops

- Loops let us repeat code without writing it many times.
- Run a block while a condition is true.
- Main types of loops:
  - `for`: repeat a set number of times.
  - `while`: repeat as long as condition is true.
  - `do...while`: run at least once, then check condition.
  - `for...of`: loop over items in an array.
  - `for...in`: loop over keys in an object.

Functions

- A function is a **reusable** block of code.
- Helps avoid repetition and keeps code **organized**.
- Can take parameters (inputs).
- Can return a value (output).
- Defined once, used many times.

Method I: Function Declaration

- Classic way to define a function.
- Starts with the `function` keyword.
- Can be called before or after it's defined (hoisted).

Projects/University/web-ta/sampleprojects/js/fn.js

```
function add(a, b) {
  return a + b;
}

console.log(add(2, 3)); // 5
```

Remember, this can be hoisted!

Method II: ES6 Arrow Functions

- Introduced in **ES6** (modern JS).
- Shorter syntax; often used for quick functions.
- Not hoisted (must be defined before use).

Projects/University/web-ta/sampleprojects/js/fn.js

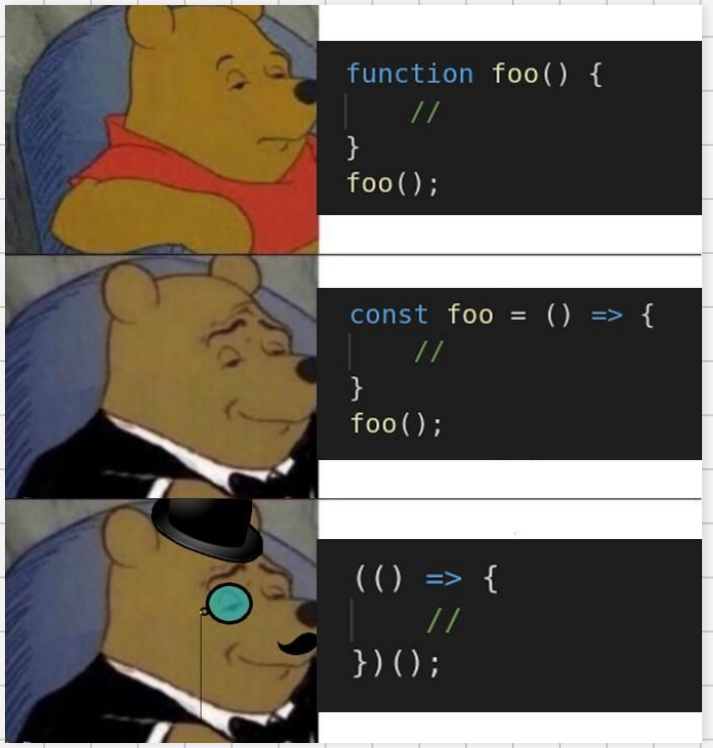
```
const multiply = (a, b) ⇒ {
  return a * b;
};

console.log(multiply(4, 5)); // 20
```

*Remember, this **cannot** be hoisted!*

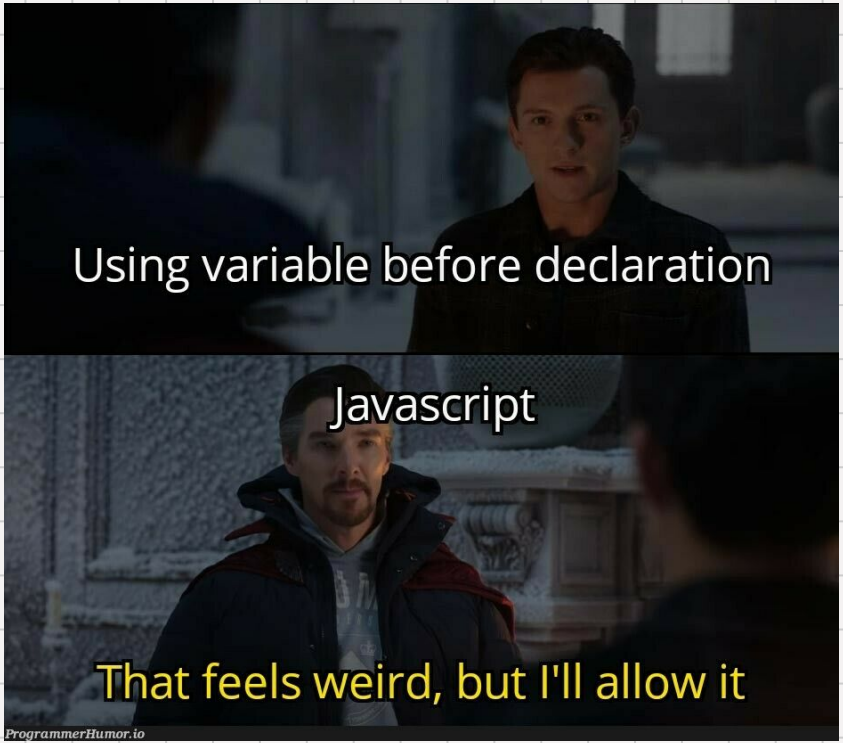
Method I vs. Method II

- Function: declared with function keyword.
- Arrow: **shorter** => syntax.
- Functions are **hoisted** (usable before definition).
- Arrows are **not** hoisted.
- Functions have their own `this`; arrows **inherit** `this` (more on that later!).



What is hoisting?

- Moving declarations before execution
- Common in languages like JavaScript
- Means code runs as if declared at top
- Variables/functions exist before use
- Can cause unexpected behavior



Hoisting in JavaScript

- JS moves declarations to the top of scope
- Works for `var`, `function`, and `class` (partially)
- Variables are hoisted but not initialized
- Access before init gives `undefined` or error
- Makes code order confusing if misused

Classes

- Introduced in ES6 as a cleaner way to create objects.
- Use the `class` keyword.
- Classes define a blueprint for creating objects.

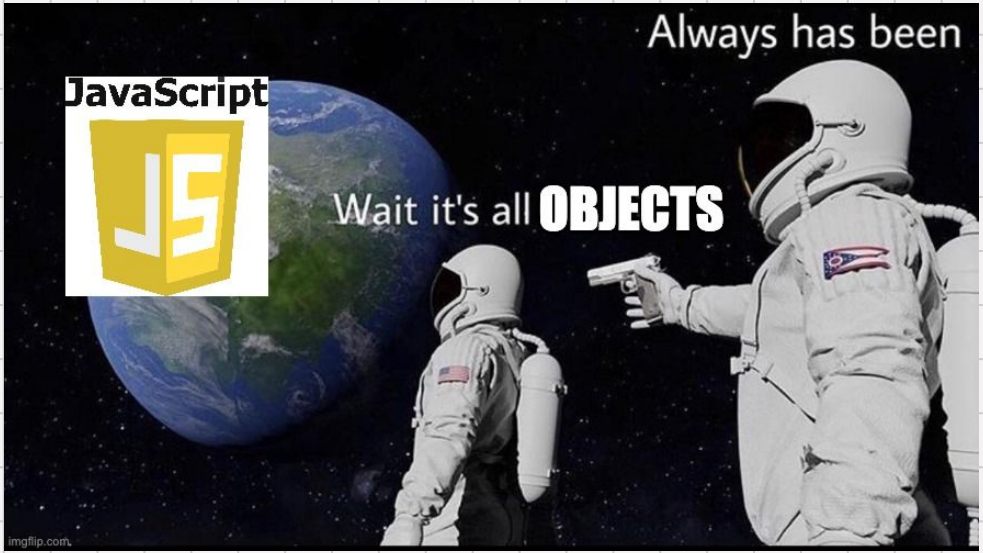
Projects/University/web-ta/sampleprojects/js/classes.js

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log("Hello, " + this.name);
  }
}

const kermit = new User("Kermit", 22);
kermit.greet();
```

Objects

- Objects group related **data** and **behavior**.
- Made of key-value pairs.
- Values can be data (properties) or functions (methods).
- JSON (JavaScript Object Notation) is a common way to store & share object data.



Technically speaking, in JavaScript, almost everything is an object.

Object Literals

- Define objects with `{ key: value }` pairs
- Store related data & behavior together
- **Keys** are strings (or symbols), **values** any type
- Support methods and nested objects
- Basis for JSON structure in JS

Projects/University/web-ta/sampleprojects/js/ol.js

```
const user = {
  name: "Barry Allen",
  age: 24,
  isTheFlash: true,
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }
};

user.greet();
```

Destructing

- Extract values from arrays or objects easily
- Uses {} for objects, [ ] for arrays
- Can rename variables during destructuring
- Supports default values if property is missing
- Makes code clearer

Projects/University/web-ta/sampleprojects/js/destructing.js

```
const person = {
  first: "Alice",
  last: "Doe",
  age: 30 };
const { first, age } = person;

const colors = ["red", "green", "blue"];
const [primary, secondary] = colors;

console.log(first, age);          // Alice 30
console.log(primary, secondary); // red green
```

The `this` Keyword

- Refers to the object that is calling the function.
- In objects, `this` points to the object itself.
- In classes, `this` points to the created instance.
- In arrow functions, `this` is inherited from the outer scope.
- The value of this depends on how a function is called, not where it’s defined.

The `this` Keyword (cont.)

- Refers to the object that calls the function
- Value depends on call context, not declaration
- In global scope: `window` (browser) or `undefined` (strict mode)
- In methods: the object before the dot
- In arrow functions: inherits `this` from outer scope (lexical `this`)
- Can be bound manually using `call`, `apply`, or `bind`

sample.js

```
const obj = {
  name: "Robert",
  say() {
    console.log(this.name);
  }
};
obj.say(); // "Robert"
const f = obj.say;
f(); // undefined (or global)
```

Arrays

- Arrays store **ordered** collections of values.
- Elements are accessed by **index** (starting at 0).
- Can hold mixed types (`numbers`, `strings`, `objects`, etc.).
- Have **built-in** methods for adding, removing, and transforming data.
- Common methods: `push`, `pop`, `map`, `filter`, `forEach`.

Projects/University/web-ta/sampleprojects/js/array.js

```
const numbers = [1, 2, 3];
numbers.push(4);
console.log(numbers[0]);
```

What's the output of this code?

Array Methods

- `push` / `pop`: add/remove from the **end**.
- `shift` / `unshift`: remove/add at the **start**.
- `filter`: keep only elements that match a condition.
- `forEach`: run a function for each element.
- `find` / `findIndex` → get the first element (or index) that matches.
- `for...of`: loop through iterable values (arrays, strings, etc.) directly.

`map` and `reduce` Methods

- `map`
  - Transforms each element into a new array
  - Does not change the original array
  - Often used for formatting or calculations
- `reduce`
  - Combines all elements into a single value
  - Uses an accumulator + current value
  - Useful for sums, averages, objects, etc.

An Example...

Projects/University/web-ta/sampleprojects/js/mapred.js

```
const students = [{ name: "Alice", score: 85 }, { name: "Bob", score: 72 }];

const scores = students.map(s => s.score);
console.log("Scores:", scores);

const total = scores.reduce((acc, s) => acc + s, 0);
const average = total / scores.length;

console.log("Total:", total);
console.log("Average:", average);

const results = students.map(s => ({
  ...s,
  passed: s.score >= 70
}));

console.log("Results:", results);
```

Class Activity Time!

Predict the output of this code.

Trace the state of elements in `nums` in each step and explicitly tell what happens to them.

Projects/University/web-ta/sampleprojects/js/arraymethods.js

```js
const nums = [1, 2, 3, 4];

nums.push(5);
nums.pop();

nums.unshift(0);
nums.shift();

const doubled = nums.map(n => n * 2);

const evens = nums.filter(n => n % 2 === 0);

nums.forEach(n => console.log(n));

console.log(nums.find(n => n > 2));
console.log(nums.findIndex(n => n > 2));
```

Error Handling

- Errors happen when code breaks at runtime.
- Without handling, they stop the whole program.
- Use `try...catch` to safely run risky code.
- `throw` lets you create your own errors.
- `finally` runs code whether an error happened or not.

Projects/University/web-ta/sampleprojects/js/errorhandling.js

```
try {
  JSON.parse("{ bad json }");
} catch (err) {
  console.log("Oops:", err.message);
} finally {
  console.log("Done!");
}
```

BEST PRACTICE Catch only errors you can actually handle.

Template Literals

- Introduced in ES6 (use **backticks** `).
- Allow multi-line strings without `\n`.
- Support string interpolation with `${expression}`.

Projects/University/web-ta/sampleprojects/js/templateliterals.js

```
const product = "Laptop";
const price = 1200;
const quantity = 2;

const message = `
You ordered: ${quantity} × ${product}
Total price: $${price * quantity}
Thank you for shopping with us!
`;

console.log(message);
```

JavaScript Object Notation

- JSON (JavaScript Object Notation) is a text format for **storing** and **exchanging** data.
- Looks like JavaScript objects, but stricter (keys & strings in quotes).
- JS provides two key methods:
  - `JSON.stringify(obj)`: converts object to JSON string.
  - `JSON.parse(jsonStr)`: converts JSON string to object.

Projects/University/web-ta/sampleprojects/js/json.js

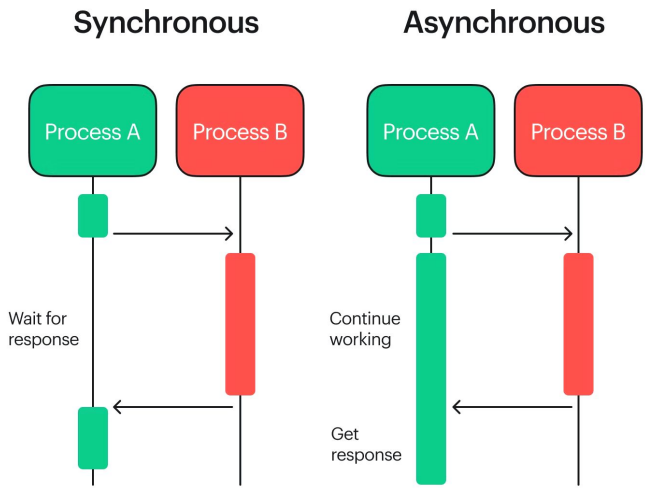
```
const order = {
  id: 101,
  customer: {
    name: "Alex",
    email: "alex@example.com"
  },
  items: [
    { product: "Laptop", qty: 1, price: 1200 },
    { product: "Mouse", qty: 2, price: 25 }
  ],
  paid: true
};

const json = JSON.stringify(order, null, 2);
console.log(json);

const obj = JSON.parse(json);
console.log(obj.customer.email);
```

Asynchronous JavaScript

- By default, JS is **single-threaded** → runs one task at a time.
- Some operations take time (e.g. fetching data, waiting for timers).
- Instead of blocking everything, JS uses **asynchronous code**.
- Async code lets other tasks run while waiting.



Source: Ramotion

Method I: Callback

- A `Promise` represents a value that may arrive now, later, or never.
- Has 3 states: pending → fulfilled → rejected.
- Use `.then()` for success, `.catch()` for errors.
- Avoids deep nesting from callbacks.

Projects/University/web-ta/sampleprojects/js/callback.js

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received!");
  }, 1000);
}

fetchData(result => {
  console.log(result); // ✅ "Data received!" (after 1 sec)
});
```

BEST PRACTICE Break logic into smaller functions instead of nesting too deep.

Method II: Promises

- A `Promise` represents a value that may arrive now, later, or never.
- Has 3 states: pending → fulfilled → rejected.
- Use `.then()` for success, `.catch()` for errors.
- Avoids deep nesting from callbacks.

Projects/University/web-ta/sampleprojects/js/promises.js

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data received!");
    // reject("Error occurred!");
  }, 1000);
});

fetchData
  .then(result => console.log(result))    // ✅ "Data received!"
  .catch(error => console.log(error));
```

BEST PRACTICE Always add `.catch()` to handle errors; unhandled rejections can crash apps.

Method III: async/await

- Built on top of Promises (ES8+).
- Lets you write async code that **looks like** sync code.
- Use `async` before a function to return a Promise.
- Use `await` to pause until a Promise resolves.
- Must be inside an `async` function.

Projects/University/web-ta/sampleprojects/js/asyncawait.js

```
async function getData() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (err) {
    console.error("Error:", err);
  }
}
```

BEST PRACTICE Always wrap `await` calls in `try...catch` to handle errors gracefully.

Comparison of Methods

- Callbacks: Pass a function to run later.
- Promises: Represent a future value with states.
- `async`/`await`: Built on Promises, but cleaner syntax.
- Callbacks: can cause nested “callback hell.”
- Promises: chain with `.then()` / `.catch()`.
- `async`/`await`: looks synchronous, easier to read.
- Error handling: callbacks → manual, promises → `.catch()`, `async` → `try...catch`.

Async Problems & Pitfalls

- Callback hell: deeply nested, hard-to-read code.
- Forgetting `await`: unexpected Promises instead of values.
- Uncaught rejections: crashes if `.catch()` or `try...catch` missing.
- Race conditions: multiple `async` tasks finish in the wrong order.
- Blocking awaits in loops: slow performance vs running in parallel.
- Mixing sync & async code: bugs from assumptions about timing.

Class Activity Time!

Predict the output

What's the order of tasks being done?

How did you came to that conclusion?

Projects/University/web-ta/sampleprojects/js/asyncchallenge.js

```
function task(name, time) {
  return new Promise(resolve ⇒ {
    setTimeout(() ⇒ {
      console.log(name, "done");
      resolve(name);
    }, time);
  });
}

async function run() {
  const a = task("A", 1000);
  const b = task("B", 500);
  const c = await task("C", 300);

  console.log("Waiting ... ");
  const results = await Promise.all([a, b, c]);
  console.log("All finished:", results);
}

run();
```

Class Activity Time!

Predict the output

What's the order of tasks being done?

How did you came to that conclusion?

Projects/University/web-ta/sampleprojects/js/racechallenge.js

```
let winner = null;

function race(name, time) {
  return new Promise(resolve => {
    setTimeout(() => {
      if (!winner) winner = name;
      resolve(name);
    }, time);
  });
}

async function run() {
  const results = await Promise.all([
    race("A", Math.random() * 500),
    race("B", Math.random() * 500),
    race("C", Math.random() * 500)
  ]);
  console.log("Winner:", winner);
  console.log("All finished:", results);
}

run();
```

Resources

- https://javascript.info
- https://www.geeksforgeeks.org