Introduction to JavaScript

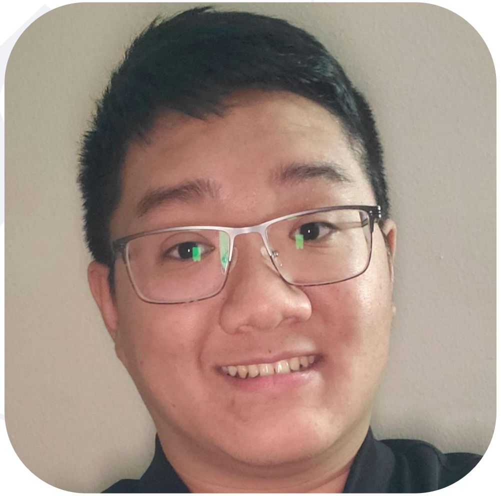
CIS 1962 (Fall 2025)
September 8th, 2025

Lesson Plan

Class Introduction

Penn Engineering

Staff



Instructor:
Voravich Silapachairueng
(he/him)

- Alumni of the MCIT program at UPenn, currently pursuing a career in teaching and software engineering.
- **Office hours**: Monday, 3 - 5 PM at Levine 601 (Bump Space)
- **Email**: voravich@seas.upenn.edu
- Fun Fact: I've been learning game development and pixel art as a hobby over the summer!

Staff

- Junior studying NETS
- **Office hours**:
  Monday, 7 - 9PM
  Levine 5th floor bump space
  (next to 501)
- **Email**: esinx@seas.upenn.edu
- Fun Fact: I served in the Korean Army as an intelligence specialist, and I built a React app to manage and distribute physical keys while I was there!

TA:
Eunsoo Shin
(he/him)

Prerequisites

CIS 1200, or equivalent programming experience

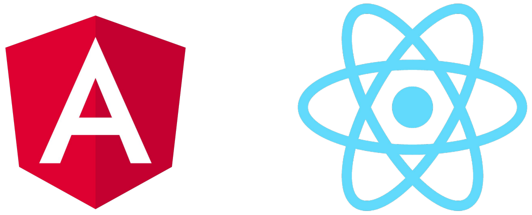
The class pace will be brisk for basic programming topics such as variables, functions, and control flow, with the assumption that you already have experience using them in other programming languages (like Java and Python).

This class will focus on the JavaScript implementation of those topics, and how you apply them to a web development context.

What will this class be about?

JavaScript Fundamentals:

The core concepts of JavaScript that make it unique



Modern JS Frameworks and Tools

Exploring various libraries and frameworks that power modern web applications

Foundations of Web Development:

Learning how JavaScript interacts with HTML and CSS to create dynamic web pages

Penn Engineering

Class Schedule and Topics

Module 1: Core JavaScript Syntax and Concepts

- What makes JavaScript unique?
- How do we write code to do specific tasks in JavaScript?
- How do we deal with asynchronous tasks in JavaScript?

Lecture	Date	Topic
Lecture 1	**9/8**	Introduction to JavaScript
Lecture 2	**9/15**	Collections and Control Flow
Lecture 3	**9/22**	Callbacks and Asynchronous JavaScript

Class Schedule and Topics

Module 2: Web Development and Advanced JS Topics

- How does JavaScript interact with HTML and CSS?
- How should project code be structured for web applications?
- What tools does JavaScript have for full-stack development?

Lecture 4	9/29	Lecture 5	10/6	Lecture 6	10/20
HTML, CSS, and the DOM		Project Management, NPM, OOP		Testing, Linting, and Introduction to Full Stack Development	

Class Schedule and Topics

Module 3: Exploration of Contemporary JavaScript Topics

- Topics for these 6 lectures will be decided by student interests (via a survey in early-October)
- What would you like to learn from JavaScript? What frameworks or libraries are you interested in exploring together?

Lectures 7-12	**10/27-12/1**
???	

Class Policies

Grade Breakdown:

Attendance	5%
Homework	60%
Final Project	35%

- Attendance taken through online polling (Slido)
  - Attendance is not mandatory, but lectures will not be recorded
- Homeworks are graded based on **correctness** (passing tests), and **style** (running through eslint) based on a posted style guide
  - 24 hours late = -10%
  - 25-72 hours late = -20%
- **Extensions** will be handled through email/Ed, feel free to reach out if you have extenuating circumstances or accommodations!

AI Policy

- Goals behind AI Policy:
  - Encourage **responsible** use of LLM tools in the classroom
  - Augmentation, rather than automation
  - Support full transparency and attribution of work to AI
  - Teach skills of working with AI tools for the workplace
- Each homework will be supplemented with one of two **AI enrichment assignments**, changing depending on whether or not you used AI during the assignment

AI Policy

- If you **did use AI**, you will document your usage:
  - Include the context, prompts, and conversations you had with the LLM (as a screenshot, or preferably a link)
  - Write a short response on why you used AI, what you learned from using it, and evaluate the quality of the response
- If you **did NOT use AI**, you will evaluate an AI response to your code:
  - Ask an AI about how you might improve your code, through improving efficiency, improving syntax, or learning other ways certain actions can be performed (better algorithms, using libraries, etc.)
  - Evaluate the quality of the response, perform some research to cross-reference whether AI is hallucinating or providing a solid response to your query

- Examples for both of these responses will be on the class website!

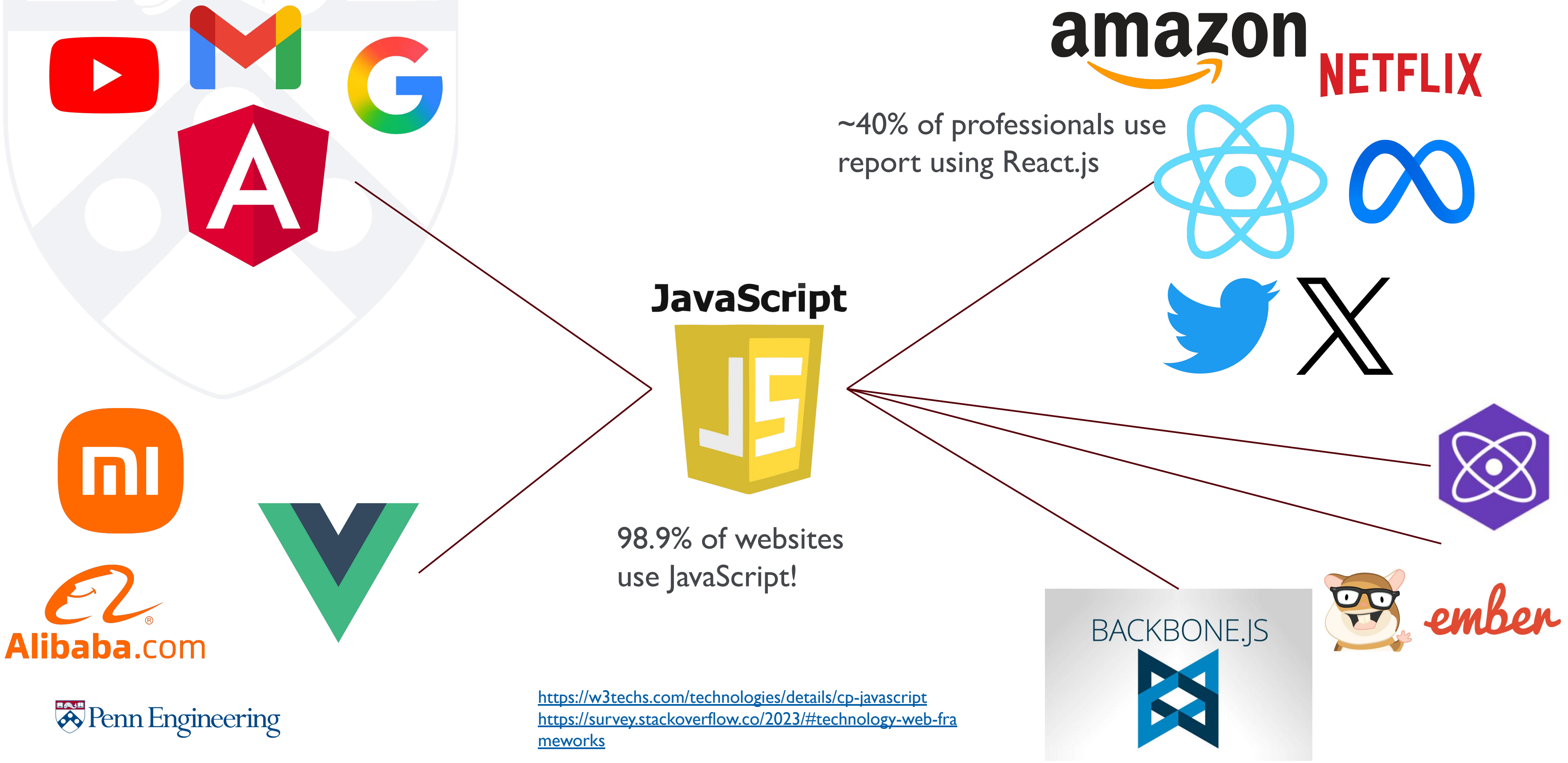
Any questions?

Introduction to JavaScript

What is JavaScript?

- JavaScript is a **high-level**, **interpreted** programming language
- It is used for dynamic client-side scripting on web pages
  - Controlling the content of web pages
  - Retrieving from servers
  - Storing to databases
  - Building and deploying web interfaces
  - … among other things!

What websites do you visit often?



~40% of professionals use report using React.js

98.9% of websites use JavaScript!

https://w3techs.com/technologies/details/cp-javascript
https://survey.stackoverflow.co/2023/#technology-web-frameworks

Key Features of JavaScript

- Runs natively in all browsers
- Single-Threaded Event Loop

```javascript
console.log('Hi!');

setTimeout(() => {
    console.log('Execute
immediately.');
}, 0);

console.log('Bye!');
```

output:

```
Hi!
Bye!
Execute immediately
```

Key Features of JavaScript

- Loosely-Typed, Aggressive Type Coercion

```
console.log("5" * "2") // 10
console.log(false + null) // 0

var empty;

if (empty) {
     console.log(“falsy!”);
}
```

- Event-driven programming

```
btn.addEventListener('click',
function() {
  btn.style.backgroundColor = 'red';
});
```

A Brief History of JavaScript

1995: Brendan Eich, part of Netscape, creates "Mocha", later renamed to "LiveScript" in only 10 days

December 1995: The language is renamed to "JavaScript" to capitalize on Java's popularity

A Brief History of JavaScript

1996-1997: JavaScript is submitted to ECMA International for approval, leading to its standardization as ECMAScript (ES1), revolutionizing web development as a cross-browser compatible language

https://www.ecma-international.org/wp-content/uploads/ECMA-262_1st_edition_june_1997.pdf

A Brief History of JavaScript

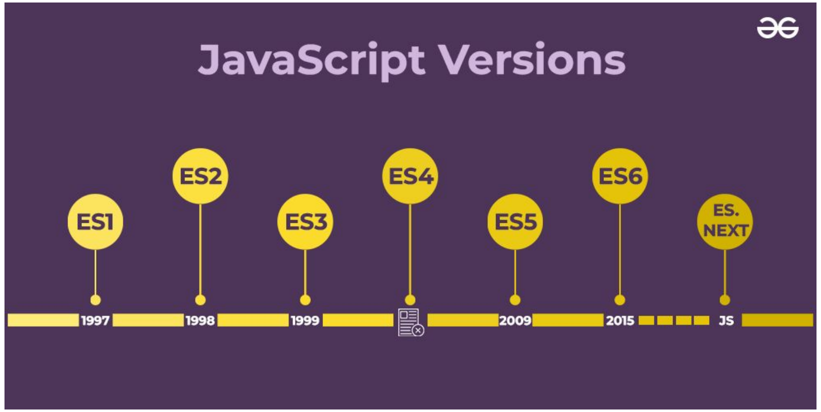
2005: The rise of AJAX (Asynchronous JavaScript and XML) coincides with the rise of Web 2.0, allowing for more dynamic web pages that have features such as live loading and dynamic content updates

- Gmail
- Facebook
- MySpace

A Brief History of JavaScript

2009: Node.js revolutionizes JavaScript, allowing for both client-side and server-side programming. ES5 is also released, allowing JSON support.

2015: ES6 releases with modern features like classes, modules, and promises.



2023: ESNext/ES2023

JavaScript: An Interpreted Language

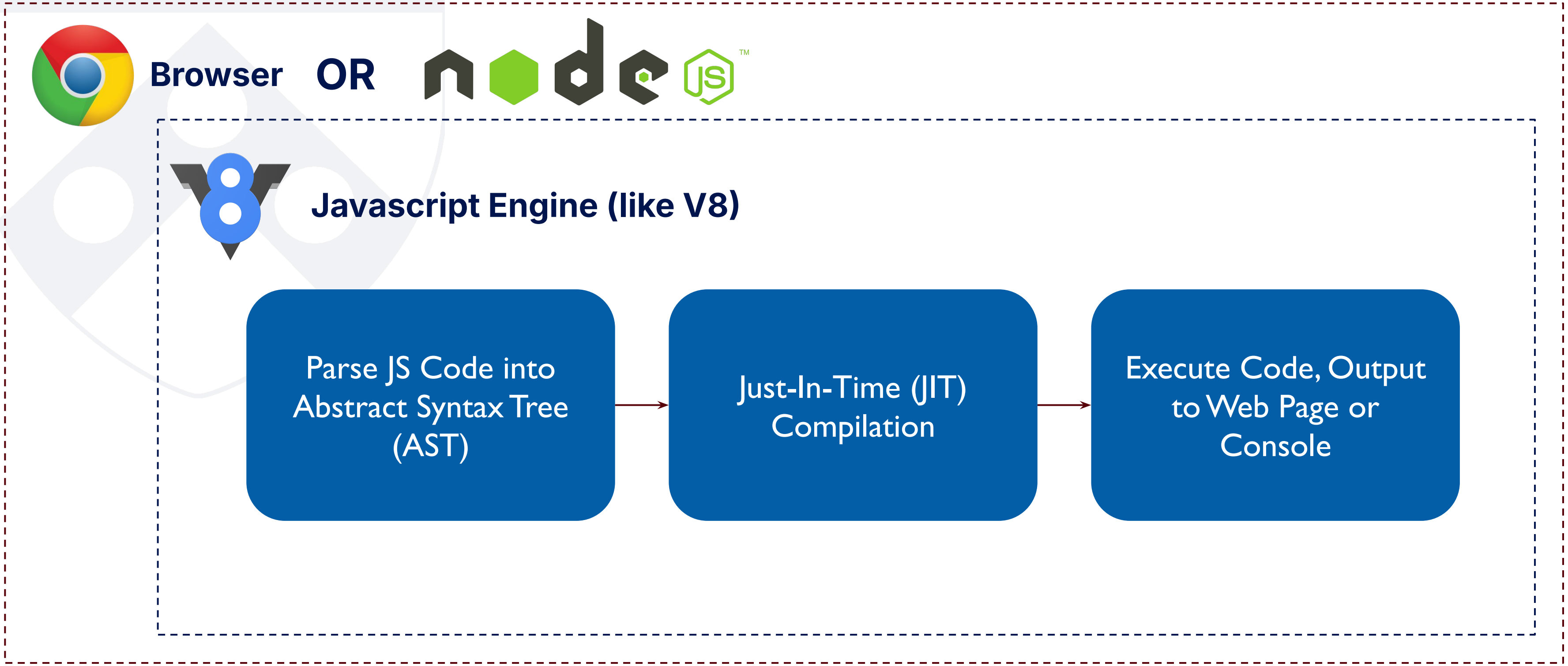
How do we run a .js file?

It must be parsed and interpreted by a **JavaScript engine.**

Unlike purely interpreted languages (like Ruby or Python), JavaScript makes use of a **Just-In-Time (JIT) Compiler** to both compile and interpret source code.

This is done inside a browser environment or Node.js.

Interpreting JavaScript


Browser
OR
node JS
Javascript Engine (like V8)
Parse JS Code into Abstract Syntax Tree (AST)
Just-In-Time (JIT) Compilation
Execute Code, Output to Web Page or Console

Hello World & Data Types

How do we print out items in JavaScript and work with various types of data?

Penn Engineering

hello_world.js

JavaScript files use the extension **.js**

You can print to the console using **console.log()**

Ending statements in semicolons (;) is **optional**, but good style.

Comments can be declared with **//**

```
// hello_world.js
console.log(“hello world!”);
```

Run the line of code within a browser, or if using Node for the file:

```
node hello_world.js
```

Data Types

```
let n; // undefined
n = null; // null
n = 10; // number
n = “ailurus fulgens”; // string
n = true // boolean
n = {
    name: “UPenn”,
    state: “Pennsylvania”,
    ivy_league: true,
}; // object
n = [ 1, 2, “hello”, false]; // array
```

Type Coercion

JavaScript is **loosely typed** and **flexible**.

The language will attempt to convert types into sensible ones with they are mixed.

`console.log(“5” + 1)`	“51”
`console.log(“5” - 1)`	4
`console.log(0 + true)`	1
`console.log(0 + false)`	0
`console.log(“a” + 2 - 4 + true)`	NaN

Booleans: Truthy and Falsy

JavaScript popularized the terms "**truthy**" and "**falsy**" due to its very aggressive type coercion.

Values that are **falsy**: 0, "", null, undefined, NaN, false

All other values are **truthy**.

```
let password = “”

if (!password) {
    console.log(“password empty!”)
}
```

```
if (nonexistant) {
    console.log(“this variable
doesn’t exist yet!”)
}
```

Numbers

Numbers in JavaScript are stored as double precision floating point numbers, maxing at $2^{53} - 1$.

Numbers beyond $2^{53} - 1$ require the **BigInt** type. **(append n)**

```
let big = 1234567890123456789012345678901234567890n;
```

Numbers have precision up to 15-17 decimal places. Floating point arithmetic can result in errors due to binary representations.

```
console.log(0.1 + 0.2 === 0.3) // false!
```

Strings

Strings can be created with single quotes (‘’), double quotes (“”), or backticks (``, often called **template literals**)

```
let string1 = ‘hello world’
let string2 = “hello world”
let string3 = `hello world`
```

Strings are **immutable**; you cannot change individual characters within a string after declaration.

```
let string = “hello world”
string[0] = “y”
console.log(string) // Still hello world
```

Strings: Template Literals

Introduced in ES6, template literals allow for **multi-line strings** and **embedding expressions** (like variable values) into strings.

```
let string = `change da world

my final message.

Goodbye`
```

```
let name = “Voravich”
console.log(`Hello, ${name}!`);
```

```
let x = 9, y = 8;
console.log(`Sum: ${x + y}`);
```

```
let isMember = false;
console.log(`Access: ${isMember ?
"Granted" : "Denied"}`);
```

5-Minute Break!

Operators

What operators are available in JavaScript and how do we use them?

Operators: Arithmetic

Operator	Name	Example
+	Addition	`let x = 4 + 7 // x = 11`
-	Subtraction	`let x = 11 - 4 // x = 7`
*	Multiplication	`let x = 4 * 7 // x = 28`
/	Division	`let x = 28 / 7 // x = 4`
%	Modulus	`let x = 30 % 7 // x = 2`
++	Increment	`let x = 2; x++; // x = 3`
--	Decrement	`let x = 2; x--; // x = 1`

Operators: Assignment

Operator	Example
`=`	`let x = 5`
`+=`	`let x = 5; x += 5 // x = 10`
`-=`	`let x = 10; x -= 5 // x = 5`
`*=`	`let x = 5; x *= 6 // x = 30`
`/=`	`let x = 30; x /= 5 // x = 6`
`%=`	`let x = 31; x %= 5 // x = 1`

Operators: Logical

Operator	Name	Example

!	NOT	`!true // false`
		`!false // true`
		`!(true || false) // false`

&&	AND	`true && true // true`
		`true && false // false`
		`false && false // false`

||	OR	`true || true // true`
		`true || false // true`
		`false || false // false`

Operators: Comparison

Operator	Name	Example
`==`	Loose Equality	`2 == “2” // true`
`!=`	Loose Inequality	`2 != “2” // false`
`===`	Strict Equality	`2 === “2” // false`
`!==`	Strict Inequality	`2 !== “2” // true`
`>`	Greater Than	`6 > 3 // true`
`<`	Less Than	`5 < 10 // true`
`>=`	Greater Than or Equal To	`50 >= 50 // true`
`<=`	Less Than or Equal To	`40 <= 100 // true`

Loose Vs. Strict Equality/Inequality

The behavior of **loose equality/inequality** (==, !=) is another consequence of JavaScript type coercion!

== or != between a number and a string will attempt to coerce both types to match each other, giving unintended behavior.

Use **strict equality/inequality**(=== , !==) in cases where it matters where numbers and strings are used together and should not be strictly equal.

Loose Vs. Strict Equality/Inequality

```
let input = “”

if (input == 0) {
    console.log(“input is 0”);
} else {
    console.log(“input is NOT 0”);
}
```

output

```
input is 0
```

“” is coerced to be 0!

```
let input = “”

if (input === 0) {
    console.log(“input is 0”);
} else {
    console.log(“input is NOT 0“);
}
```

output

```
input is NOT 0
```

Strict Equality fixes this behavior.

Variables

How do you declare a variable for use in JavaScript?

Variables

Variables in JavaScript are used to store data values that can be referenced and manipulated in a program.

JavaScript variables does not use explicit type declarations - variables are **dynamically typed**.

```
let value;
value = 157;
console.log(typeof value); // number

value = “hello world”;
console.log(typeof value); // string
```

Penn Engineering

Variables in Javascript

Variables in Javascript can be declared in 4 ways:

undeclared

```
x = 1
```

var

```
var x = 1
```

let

```
let x = 1
```

const

```
const x = 1
```

Undeclared Variables

Variables not declared using var, let, or const are undeclared.

They, by default, become **global variables**, which may clash with other declared variables.

```
function test() {
    x = 1
}

test()
console.log(x) // prints 1
```

"use strict"

The literal expression `“use strict”`, introduced in ES5, **prevents the use of undeclared variables.**

```
“use strict”
x = 1 // ReferenceError: x is not defined
```

Using "var"

Before 2015 (ES6), using var to declare variables was the standard in Javascript

```
function test() {
    var x = 9
    console.log(x) // prints 9
}
```

A variable declared with var is **function-scoped**

Problems with var

In modern Javascript, using var is often **too permissive** and causes unintentional behavior:

```
function test() {
    if (true) {
        var x = 9
    }
    console.log(x) // prints 9
}
```

Because of function-scoping, variables persist beyond internal blocks of code

```
function test2() {
    var x = 0
    var x = 9
    console.log(x) // prints 9
}
```

Variables declared with var can be redeclared

let and const

ES6 introduced the statements let and const, which are improvements from using var

Block-scoped:
variables exist only in the {} block they are defined in

```
function test() {
    if (true) {
        let x = 9
    }
    console.log(x) // ReferenceError
}
```

Cannot be redeclared:
Redeclaration causes an error

```
function test2() {
    let x = 2
    let x = 3 // SyntaxError
}
```

Differences between let and const

let and const mostly differ by their mutability:

const is **immutable**:

you cannot reassign a const variable, though it can be changed

```
function test() {
    const arr = [1, 4, 2]
    arr.push(8) // allowed
    arr = [5, 7] //  TypeError
}
```

let is **mutable**:

You can reassign items declared with let

```
function test2() {
    let count = 0
    for (let i = 0; i < 10; i++) {
        count++;
    }
    console.log(count)
}
```

Best practices for variable declaration

- By default, you should use **const** if intend not to re-assign the variable in the current scope.
- If you intend to reassign, use **let**.
- Some legacy code may require you to utilize **var**, but these cases are rare.
- Avoid leaving undeclared variables, use strict mode if applicable.

Functions

How do you write functions in JavaScript?

What are some unique ways they can be used?

Functions

Functions are reusable blocks of code that perform tasks.

Functions in JavaScript are **first-class objects**. They can be:

- Assigned as variables
- Stored in data structures (arrays, objects)
- Passed as arguments
- Returned from other functions

Higher-order functions

Declaring Functions

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

Function Declaration (hoisted)

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

Function Expression/ Anonymous Function

```
const add = (a, b) => {
     return a + b;
}
```

Arrow Function

Calling Functions and Arguments

You can call a named function using its name followed by the defined number of arguments in the declaration or expression.

```
add(34, 56)
```

Arguments of primitive data types (number, string, boolean, null, undefined) are **pass by value**.

- Changes to the parameter inside the function do not affect the original value

Arguments of objects, arrays, and functions are partially **pass by reference**.

- Changes to the object inside the function affect the original object, but reassignments do not affect the original object.

Argument Examples

```
function double(arg) {
    arg = arg * 2
}

let x = 20;
double(x)
console.log(x) // prints 20
```

Primitives: Pass By Value

```
function editName(file, newName) {
    file.fileName = newName;
}

let newFile = {
    fileName: ""
}
editName(newFile, "hello_world.js")
console.log(newFile.fileName) //
prints hello_world.js
```

Non-primitives: Pass By Reference

Functions Hoisting

Functions declared using function declaration are **hoisted**.

This means within the scope where the function is defined, the **function declaration is moved to the top**.

You can call them before you declare them in the code!

Function Declaration

```
add(34, 56) // Allowed

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

Function Expression

```
add(34, 56) // Error: add is not a
function

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

Higher Order Functions: Intro

Functions can be specified as arguments for other functions. This allows you to use that argument to call them from within the other function.

```
function compute(a, b, operation) {
  return operation(a, b);
}

function multiply(x, y) {
  return x * y;
}

console.log(compute(5,6,multiply))
```

Live Coding!