LECTURE 11

# Core JavaScript Concepts (ES6+)

Functions, closures, and objects — the rules that make JavaScript feel different from every language you already know.

CSC336 Web Technologies

OVERVIEW

# In This Lecture

1. JavaScript's role on the web, and how to embed it in a page
2. var, let, and const — plus primitive vs. reference types
3. Operators, type coercion, and == vs. ===
4. Scope, hoisting, and the temporal dead zone
5. Three ways to write a function, and ES6 parameter/destructuring features
6. How this is determined, and controlling it with call, apply, bind
7. Closures — what they are, why they matter, and common pitfalls
8. Objects, arrays, template literals, and ES6 modules

# The Three Layers of a Web Page

| Layer | Language | Job |
|---|---|---|
| **Structure** | HTML | What content exists on the page |
| **Presentation** | CSS | How the content looks |
| **Behavior** | JavaScript | How the page reacts and changes over time |

*JavaScript runs client-side, inside the browser, on the user's own computer — not on the server.*

# Three Ways to Add JavaScript

```
<!-- 1. Inline: avoid -- hard to maintain -->
<button onclick="alert('Hello!')">Click me</button>

<!-- 2. Internal: inside a <script> tag -->
<script>
  console.log("Hello from an internal script");
</script>

<!-- 3. External: linked .js file -- the standard approach -->
<script src="app.js"></script>
```

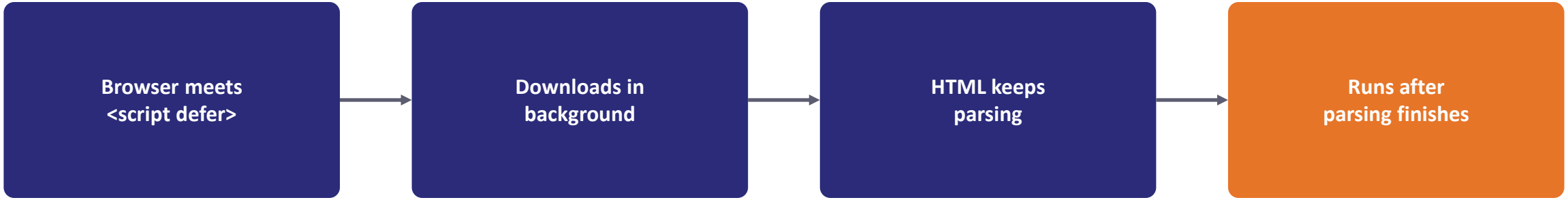
External scripts keep HTML and JS in separate files, and the browser can cache the .js file across page loads.

# defer vs. async

```
<script src="app.js" defer></script>
<script src="analytics.js" async></script>
```

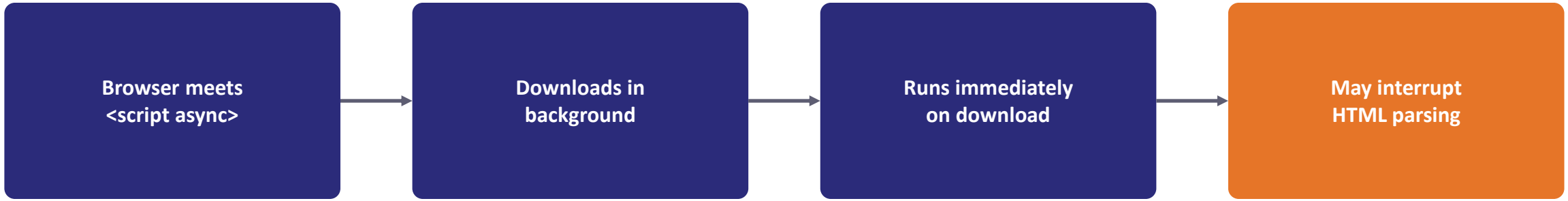
By default a <script> tag stops HTML parsing, downloads, and runs immediately -- defer and async change that.

# defer: Downloads in the Background




Multiple defer scripts run in the order they appear. Use defer for scripts that need the DOM to exist -- most of the time.

# async: Runs the Moment It Arrives




Scripts can run out of order. Use async only for independent scripts that don't touch the page, like analytics trackers.

# Rule of Thumb

**TIP**

Put defer on almost every script you write. Reach for async only for scripts that do not depend on the DOM and do not need to run in a specific order.

# var, let, and const

```
var oldStyle = "avoid me";    // function-scoped, redeclarable -- legacy
let counter = 0;              // block-scoped, can be reassigned
const PI = 3.14159;           // block-scoped, cannot be reassigned
```

Use const by default. Use let when the value must change. Avoid var -- it predates modern JS and has confusing scoping rules.

# const Does Not Mean "Unchangeable Value"

**WARNING**

const only prevents reassigning the variable itself. If the value is an object or array, its contents can still be changed.

# const Locks the Binding, Not the Contents

```
const person = { name: "Ali" };
person.name = "Sara"; // allowed -- we didn't reassign `person`
// person = {};        // NOT allowed -- this would throw an error
```

# Primitive vs. Reference Types

| | Primitive types | Reference types |
|---|---|---|
| **Examples** | string, number, boolean, undefined, null, symbol, bigint | object, array, function |
| **Stored as** | The actual value | A pointer to a location in memory |
| **Copied by** | Value (a real, independent copy) | Reference (both point to the same data) |

# Copy by Value vs. Copy by Reference

```
// Primitives copy by value
let a = 5;
let b = a; // b gets its own copy of 5
b = 10;
console.log(a); // 5 -- unaffected

// Objects copy by reference
let obj1 = { value: 5 };
let obj2 = obj1; // obj2 points to the SAME object as obj1
obj2.value = 10;
console.log(obj1.value); // 10 -- obj1 changed too!
```

This distinction explains a huge number of beginner bugs -- keep it in mind whenever you assign one variable to another.

# Type Coercion

```
console.log("5" + 3);    // "53"  -- number coerced to string, then joined
console.log("5" - 3);    // 2     -- string coerced to number
console.log(1 + true);   // 2     -- true coerced to 1
console.log("10" == 10);  // true  -- coercion happens before comparing
console.log("10" === 10); // false -- no coercion, types differ
```

== ("loose equality") converts both sides to a common type first. === ("strict equality") compares value and type, with no conversion.

# Always Prefer === and !==

**WARNING**

Loose equality produces surprising results ("" == 0 is true, null == undefined is true). Using strict equality by default avoids an entire category of bugs.

# if / else and for

```
let hour = 14;
if (hour < 12) {
  console.log("Good morning");
} else if (hour < 18) {
  console.log("Good afternoon");
} else {
  console.log("Good evening");
}

for (let i = 0; i < 3; i++) {
  console.log("Iteration", i);
}
```

# switch

```
let day = "Mon";
switch (day) {
  case "Sat":
  case "Sun":
    console.log("Weekend");
    break;
  default:
    console.log("Weekday");
}
```

switch compares with === internally, and each case needs a break -- otherwise execution "falls through" into the next case.

# Three Kinds of Scope

1. Global scope: declared outside any function or block; visible everywhere
2. Function scope: var is visible anywhere inside the function it was declared in, even inside nested blocks
3. Block scope: let and const are visible only inside the { } block where they were declared

# var Leaks Out of a Block; let Does Not

```
function demo() {
  if (true) {
    var fnScoped = "I leak out of the if-block";
    let blockScoped = "I stay inside the if-block";
  }
  console.log(fnScoped);    // works
  console.log(blockScoped); // ReferenceError
}
```

# Hoisting and the Temporal Dead Zone

```
console.log(x); // undefined -- not an error, it's "hoisted"
var x = 5;

console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 5;
```

var is hoisted and initialized with undefined. let/const are hoisted but NOT initialized -- the gap is the temporal dead zone (TDZ).

# Why the TDZ Is a Good Thing

NOTE

The TDZ turns a silent bug (accidentally using a variable before it has a real value) into a loud, immediate error, which makes mistakes far easier to catch.

# Three Ways to Write a Function

```
// 1. Function declaration -- hoisted, callable before it appears
function add(a, b) {
  return a + b;
}

// 2. Function expression -- NOT hoisted the same way
const subtract = function (a, b) {
  return a - b;
};

// 3. Arrow function -- shorter, ES6 (2015)
const multiply = (a, b) => { return a * b; };
const square = x => x * x; // single-expression shortcut
```

# Which One Should You Use?

**TIP**

Use function declarations for top-level, named functions -- they are easy to read and are hoisted. Use arrow functions for short, inline callbacks.

# Default and Rest Parameters

```
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}
greet();          // "Hello, Guest!"
greet("Ayesha"); // "Hello, Ayesha!"

function sum(...numbers) {
  return numbers.reduce((total, n) => total + n, 0);
}
sum(1, 2, 3, 4); // 10
```

# The Spread Operator

```
const nums = [1, 2, 3];
console.log(Math.max(...nums)); // same as Math.max(1, 2, 3)

const arr1 = [1, 2];
const arr2 = [3, 4];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4]

const defaults = { color: "blue", size: "M" };
const custom = { ...defaults, size: "L" }; // { color: "blue", size: "L" }
```

# Rest vs. Spread -- Same Dots, Opposite Direction

**NOTE**

Rest gathers many values into one array (used in a function's parameter list). Spread expands one array or object into many values (used when calling a function or building a new array/object).

# Unpacking Arrays and Objects

```
// Array destructuring -- position matters
const coordinates = [10, 20];
const [x, y] = coordinates;

// Object destructuring -- name matters, order doesn't
const student = { name: "Bilal", age: 21 };
const { name, age } = student;

// Renaming and default values while destructuring
const { name: studentName, gpa = 0 } = student;
```

# this Depends on How You Call It

```
const car = {
  brand: "Toyota",
  describe: function () {
    console.log(`This is a ${this.brand}`);
  },
};
car.describe(); // "This is a Toyota" -- called as car.describe()

const detached = car.describe;
detached(); // "This is a undefined" -- `this` is no longer `car`!
```

# call, apply, and bind

```
function describe(city) {
  console.log(`${this.brand} is from ${city}`);
}
const car = { brand: "Honda" };

describe.call(car, "Tokyo");    // pass `this` and args individually
describe.apply(car, ["Tokyo"]); // pass `this` and args as an array
const bound = describe.bind(car); // returns a NEW function, `this` locked in
bound("Tokyo");
```

bind does NOT call the function -- it returns a new function permanently bound to thisArg, useful when passing a method as a callback.

# Arrow Functions and this

**WARNING**

Arrow functions do not have their own this. Instead they use this from the surrounding ("lexical") scope where they were defined -- ideal for callbacks inside methods.

# Why Arrow Callbacks Are Ideal Here

```
const timer = {
  seconds: 0,
  start: function () {
    setInterval(() => {
      this.seconds++; // `this` is `timer`, inherited from `start`
      console.log(this.seconds);
    }, 1000);
  },
};
```

A regular function() here would have its OWN this -- not timer -- and this.seconds++ would fail.

# A Function That Remembers

```
function makeCounter() {
  let count = 0; // "enclosed" by the returned function
  return function () {
    count++;
    return count;
  };
}

const counter1 = makeCounter();
console.log(counter1()); // 1
console.log(counter1()); // 2

const counter2 = makeCounter(); // a completely separate `count`
console.log(counter2()); // 1
```

Each call to makeCounter() creates a brand-new count and a brand-new inner function with a private reference to it.

# Common Uses of Closures

1. Data privacy: count above cannot be accessed except through the returned function -- a simple form of encapsulation
2. Function factories: creating specialized functions, like makeCounter above
3. Callbacks that need context: event handlers and setTimeout callbacks often rely on closures to remember values

# A Classic Loop Pitfall

```
// BUGGY: prints 3, 3, 3 -- `var` is function-scoped, all three
// callbacks share the SAME `i`, whose final value is 3.
for (var i = 1; i <= 3; i++) {
  setTimeout(() => console.log(i), 100);
}

// FIXED: prints 1, 2, 3 -- `let` creates a NEW `i` per iteration.
for (let j = 1; j <= 3; j++) {
  setTimeout(() => console.log(j), 100);
}
```

This is one of the strongest practical reasons to prefer let over var.

# Grouping and Listing Data

```
const book = {
  title: "Eloquent JavaScript",
  year: 2024,
  tags: ["javascript", "programming"],
};
console.log(book.title);       // dot notation
console.log(book["year"]);     // bracket notation -- for dynamic keys
book.pages = 472;              // add a new property

const numbers = [10, 20, 30];
numbers.push(40);              // add to the end
console.log(numbers.length);   // 4
```

# Backtick Strings

```
const name = "Hina";
const score = 92;

// Old way
console.log("Hello " + name + ", your score is " + score + "%.");

// Template literal
console.log(`Hello ${name}, your score is ${score}%.`);

const multiLine = `Line one
Line two`;
```

Template literals embed expressions with ${} and let strings span multiple lines without special characters.

# export and import

```
// mathUtils.js
export function add(a, b) { return a + b; }
export const PI = 3.14159;
export default function multiply(a, b) { return a * b; }

// main.js
import multiply, { add, PI } from "./mathUtils.js";
console.log(add(2, 3));      // 5
console.log(multiply(2, 3)); // 6
```

A module's variables and functions are private by default -- you must explicitly export what other files can use.

# Modules Are Deferred Automatically

**NOTE**

Add type="module" to the script tag to use modules in the browser: <script type="module" src="main.js"></script>. Module scripts behave like defer by default -- they don't block HTML parsing.

KEY TAKEAWAYS

# Lecture 11 in Seven Points

1. JavaScript adds behavior to a page; use defer for scripts needing the DOM, async for independent ones.
2. Prefer const by default, let when reassignment is needed, and avoid var.
3. Primitives copy by value; objects and arrays copy by reference -- a frequent source of beginner bugs.
4. Use === / !== instead of == / != to avoid unexpected type coercion.
5. let/const are block-scoped and sit in the temporal dead zone; var is function-scoped and hoisted as undefined.
6. this depends on how a function is called; call/apply/bind control it explicitly, arrow functions inherit it lexically.
7. A closure lets an inner function remember its outer function's variables after that function has returned.